
AI and Co-Simulation Driven Resource Management in Fog Computing Environments

Shreshth Tuli

2023

Department of Computing
Imperial College London

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy of Imperial College London

This page intentionally left blank.

Copyright

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose. When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

This page intentionally left blank.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing that is the outcome of work done in collaboration with others, except as specified in the text and acknowledgements.

Shreshth Tuli

April 2023

This page intentionally left blank.

Acknowledgements

The contributions of this thesis have been funded by the President's PhD Scholarship at Imperial College London. I would like to express my sincere gratitude to my supervisors, Dr. Giuliano Casale and Prof. Nicholas R. Jennings, for the continuous support, encouragement and guidance along my PhD journey. I thank my examiners, Prof. Peter Pietzuch and Dr. Blesson Varghese for their valuable suggestions on improving this thesis. I would also like to thank the members of the Quality of Service Research (QORE) Lab for their support and helpful discussions throughout. A special thanks to Amani El-Kholy and the PhD team for handling the logistics and making it the smoothest journey.

I would also like to immensely thank my parents, Rajesh Tuli and Poonam Tuli, and my sister, Megha Tuli, for their pastoral care, motivation and keeping my spirits up. I would like to thank my brother, Shikhar Tuli, for the insightful discussions and collaborating on the ideas presented in this work. Finally, I would like to thank all my co-authors for helpful tips and advice.

This page intentionally left blank.

“To lose patience is to lose the battle.”

- *Gandhi*

This page intentionally left blank.

Abstract

The title of this thesis, *AI and Co-Simulation Driven Resource Management in Fog Computing Environments*, covers three main aspects of the work we present here. First, we discuss on *what* we do, which is making resource management decisions. Second, we discuss *where* we take such decisions, which are Fog computing environments. Third, we discuss *how* we take these discussions, that is, through Artificial Intelligence (AI) and Co-Simulation based methods.

We consider the Fog computing paradigm, which consists of an infrastructure consisting of heterogeneous distributed computational devices. Fog computing is an emerging paradigm encompassing a diverse spectrum of compute nodes and is considered the future of computing. These may range from resource-abundant cloud virtual machines to resource-limited compute hardware close to the user. As cloud nodes may be at a geographically distant location and a multi-hop distance from the users, they tend to offer high communication latency to users. On the other hand, we have devices at a few hops from the user also referred to as the *edge* of the network, provide reduced latency but tight compute/memory constraints. In this work, we aim to improve service quality by making intelligent resource management decisions for Fog environments. This is hard with highly dynamic modern applications and volatile resource characteristics of systems.

To make intelligent decisions, we consider that a Fog environment consists of broker and worker nodes. The former make resource management decisions, such as when to provision workers, where to place incoming tasks, and how to prevent and recover from failures to ensure service resilience. The latter is where finite-running tasks are executed to return the results to the users. In our setup, we assume that users interact with the Fog environment through gateway devices, such as smartphones or tablets, and send or receive data through sensors and actuators, such as microphones or cameras. The objective is to improve metrics of interest, such as energy consumption, the average response time of tasks, the fraction of violation of task deadlines and the operational cost of the environment. We refer to all these as Quality of Service (QoS) metrics and optimizing such metrics are crucial for both end-users and service providers.

In order to make intelligent decisions, we develop novel data-driven strategies. We leverage and improve upon AI-based methods for their speed and accuracy and the ability of such approaches to identify patterns in data that are hard to encode manually. However, AI methods are typically oblivious to the system characteristics. To overcome this, we encode system knowledge in a co-simulator, a digital replica of the physical infrastructure. Such a simulator allows us to quickly generate additional out-of-distribution data for robust model training, run performance tests of decisions of interest from an AI model and perform long-term QoS estimation to eschew myopic decision-making. Such advances allow us to provide significantly higher QoS compared to prior methods for resource management in fog environments.

This page intentionally left blank.

Preface

The work carried out during this Ph.D. led to the following publications, for which I am the first author. I thank my collaborators for their insights.

Main Contributions

1. - [Shreshth Tuli](#) and Giuliano Casale. *Optimizing the Performance of Fog Computing Environments Using AI and Co-Simulation*. International Conference on Performance Engineering 2022 (Tutorial Paper).
2. Ch. 5 [Shreshth Tuli](#). *SplitPlace: Intelligent Placement of Split Neural Nets in Mobile Edge Environments*. ACM SIGMETRICS Poster, 2021. [link](#). **First Place, Gold Medal - Student Research Competition Poster** [link](#)
3. Ch. 8 [Shreshth Tuli](#), Shikhar Tuli, Giuliano Casale and Nicholas R. Jennings. *Generative Optimization Networks for Memory Efficient Data Generation*. NeurIPS 2021 - Workshop on Machine Learning for Systems. [acc. rate: 9.2%]. [link](#).
4. Ch. 7 [Shreshth Tuli](#), Giuliano Casale and Nicholas R. Jennings. *PreGAN: Preemptive Migration Prediction Network for Proactive Fault-Tolerant Edge Computing*. IEEE International Conference on Computer Communications (INFOCOM) 2022. [acc. rate: 19%]. [link](#). **Best Paper Award** [link](#).
5. - [Shreshth Tuli](#), Giuliano Casale and Nicholas R. Jennings. *TranAD: Deep Transformer Networks for Anomaly Detection in Multivariate Time Series Data*. International Conference on Very Large Data Bases (VLDB) 2022. [link](#).
6. Ch. 8 [Shreshth Tuli](#), Giuliano Casale and Nicholas R. Jennings. *CAROL: Confidence-Aware Resilience Model for Edge Federations*. IEEE International Conference on Dependable Systems and Networks (DSN) 2022. [acc. rate: 18.7%]. [link](#). **Best Paper Award**.
7. Ch. 5 [Shreshth Tuli](#), Giuliano Casale and Nicholas R. Jennings. *Learning to Dynamically Select Cost Optimal Schedulers in Cloud Computing Environments*. ACM SIGMETRICS Poster, 2022.
8. - [Shreshth Tuli](#), Giuliano Casale and Nicholas R. Jennings. *MetaNet: Automated Dynamic Selection of Scheduling Policies in Cloud Computing Environments*. IEEE International Conference on Cloud Computing (CLOUD) 2022. [link](#).
9. - [Shreshth Tuli](#), Sukhpal Singh Gill, Giuliano Casale, and Nicholas R. Jennings. *iThermoFog: IoT Fog based automatic thermal profile creation for cloud data centers using artificial intelligence techniques*. Internet Technology Letters 3, no. 5 (2020): e198. [link](#).

10. Ch. 3 [Shreshth Tuli](#), Shivananda Poojara, Satish N. Srirama, Giuliano Casale, and Nicholas R. Jennings. *COSCO: Container Orchestration using Co-Simulation and Gradient Based Optimization for Fog Computing Environments*. IEEE Transactions on Parallel and Distributed Systems 33.1 (2022): 101-116. [link](#).
11. - [Shreshth Tuli](#), Sukhpal Singh Gill, Minxian Xu, Peter Garraghan, Rami Bahsoon, Schahram Dustdar, Rizos Sakellariou, Omer Rana, Rajkumar Buyya, Giuliano Casale and Nicholas R. Jennings. *HUNTER: AI based Holistic Resource Management for Sustainable Cloud Computing*. Journal of Systems and Software (2021): 111-124. [link](#).
12. - [Shreshth Tuli](#), Sukhpal Singh Gill, Peter Garraghan, Rajkumar Buyya, Giuliano Casale and Nicholas R. Jennings. *START: Straggler Prediction and Mitigation Technique for Cloud Computing Environments using Encoder LSTM Networks*. IEEE Transactions on Service Computing (2021). [link](#).
13. Ch. 6 [Shreshth Tuli](#), Giuliano Casale and Nicholas R. Jennings. *MCDS: AI Augmented Workflow Scheduling in Mobile Edge Cloud Computing Systems*. IEEE Transactions on Parallel and Distributed Systems (Special Issue on PDC for AI). [link](#).
14. Ch. 6 [Shreshth Tuli](#), Giuliano Casale and Nicholas R. Jennings. *GOSH: Task Scheduling using Deep Surrogate Models in Fog Computing Environments*. IEEE Transactions on Parallel and Distributed Systems (Special Issue on PDC for AI). [link](#).
15. Ch. 5 [Shreshth Tuli](#), Giuliano Casale and Nicholas R. Jennings. *SplitPlace: AI Augmented Splitting and Placement of Large-Scale Neural Networks in Mobile Edge Environments*. IEEE Transactions on Mobile Computing 2022. [link](#).
16. Ch. 8 [Shreshth Tuli](#), Giuliano Casale and Nicholas R. Jennings. *DRAGON: Decentralized Fault Remediation in Edge Federations*. IEEE Transactions on Network and Service Management 2022. [link](#).
17. - [Shreshth Tuli](#), Giuliano Casale and Nicholas R. Jennings. *SimTune: Simulator Tuning to Bridge the Reality Gap for Resource Management in Edge Computing*. Nature Scientific Reports 2022. [link](#).
18. Ch. 7 [Shreshth Tuli](#), Giuliano Casale, Ludmila Cherkasova and Nicholas R. Jennings. *DeepFT: Self-Supervised Deep Learning based Surrogate Models for Fault-Tolerant Edge Computing*. IEEE International Conference on Computer Communications (INFOCOM) 2023. [acc. rate: 19%]. [link](#).
19. Ch. 2 [Shreshth Tuli](#), et al.. *AI Augmented Fog Computing: New Trends and Challenges*. Journal of Network and Computer Applications 2022. [link](#).

Under Review / Submission

The work below is under submission or review as of 14 April 2023.

20. Ch. 4 [Shreshth Tuli](#), Giuliano Casale and Nicholas R. Jennings. *CILP: Co-simulation based Imitation Learner for Dynamic Resource Provisioning in Cloud Computing*. IEEE Transactions on Network and Service Management 2023.
21. Ch. 7 [Shreshth Tuli](#), Giuliano Casale and Nicholas R. Jennings. *PreGAN+: Semi-Supervised Fault Prediction and Preemptive Migration in Dynamic Mobile Edge Environments*. IEEE Transactions on Mobile Computing 2022.
22. Ch. 9 [Shreshth Tuli](#), Giuliano Casale and Nicholas R. Jennings. *SciNet: Co-Design in Resource Management of Distributed Computing Environments*. IEEE Transactions on Computers 2022.

Supplementary Contributions

During the Ph.D. candidature, I have also contributed to the following collaborative works (this thesis does not claim them as its contributions):

23. [Shreshth Tuli](#), Rajas Bansal, Rohan Paul and Mausam. *TANGO: Commonsense Generalization in Predicting Tool Interactions for Mobile Manipulators*. International Joint Conference on Artificial Intelligence (IJCAI), 2021. [acc. rate: 13%]. [link](#).
24. Shreya Sharma, Jigyasa Gupta, [Shreshth Tuli](#), Rohan Paul and Mausam. *GoalNet: Inferring Conjunctive Goal Predicates from Human Plan Demonstrations for Robot Instruction Following*. International Conference on Automated Planning and Scheduling (ICAPS) 2022 - Planning and Reinforcement Learning Workshop. [link](#).
25. Shikhar Tuli, Bhishma Dedhia, [Shreshth Tuli](#) and Niraj K. Jha. *FlexiBERT: Expanding the Flexibility of Transformer Architectures and Exploring a Heterogeneous Design Space*. Journal of AI Research 2022. [link](#).
26. [Shreshth Tuli](#), Matthew R. Wilkinson and Chris Kettell. *RadNet: Incident Prediction in Spatio-Temporal Road Graph Networks Using Traffic Forecasting*. International Joint Conference on Artificial Intelligence (IJCAI) 2022 - Workshop on Artificial Intelligence for Time Series Analysis. [link](#).
27. [Shreshth Tuli](#), Kinga Bojarczuk, Natalija Gucevska, Mark Harman, Xiao-Yu Wang, Graham Wright. *Simulation-Driven Automated End-to-End Test and Oracle Inference*. International Conference on Software Engineering (SEIP Track) 2023. [link](#).
28. Sundas Iftikhar, Uttkarsh Raj, [Shreshth Tuli](#), Muhammed Golec, Deepraj Chowdhury, Sukhpal Singh Gill and Steve Uhlig. *TESCO: Multiple Simulations Based AI-Augmented Fog Computing for QoS Optimization*. IEEE Conference on Scalable Computing and Communication (ScalCom) 2022.
29. [Shreshth Tuli](#), Rajas Bansal, Rohan Paul and Mausam. *TOOLTANGO: Common sense Generalization in Predicting Sequential Tool Interactions for Robot Plan Synthesis*. Journal of AI Research 2022. [link](#).

30. Mustafa Chasmai, [Shreshth Tuli](#), Mausam and Rohan Paul. *Learning Generalizable Task Policies for Embodied Agents in Domains with Rich Inter-object Interactions*. Conference on Robot Learning (CoRL) - Workshop on Long-Horizon Planning 2022.

Contents

1	Introduction	33
1.1	Motivation	34
1.2	Research Objectives	36
1.3	Contributions	37
2	Background and Related Work	41
2.1	System Background	43
2.1.1	Fog Computing	43
2.1.2	Application Scenarios	45
2.1.3	Optimization Parameters	45
2.2	AI Background	46
2.2.1	Neural Networks	46
2.2.2	Loss Functions and Model Training	50
2.2.3	Neural Network Splitting	51
2.3	Literature Review of Resource Management Methods	51
2.3.1	AI Augmented Deployment	52
2.3.2	AI Augmented Scheduling	57
2.3.3	AI Augmented Dependability	61
2.3.4	Limitations	66
2.4	Summary	67
3	COSCO Framework and GOBI Optimization	69
3.1	System Model and Problem Formulation	70
3.1.1	System Model	70
3.1.2	Workload Model	71
3.1.3	Problem Formulation	72
3.2	The GOBI Scheduler	72
3.2.1	Objective Function	73
3.2.2	Input Parameters	73
3.2.3	Model Training	74
3.2.4	Scheduling	75
3.3	The COSCO Framework	76

3.3.1	Simulator	77
3.3.2	Framework	77
3.3.3	Model Interface	78
3.4	Extending GOBI to GOBI*	78
3.4.1	Leveraging Co-Simulations	78
3.4.2	Convergence of GOBI and GOBI*	80
3.5	Performance Evaluation	81
3.5.1	Experimental Setup	81
3.5.2	Model Training and Assumptions	82
3.5.3	Workloads	83
3.5.4	Baseline Models	84
3.5.5	Evaluation Metrics	84
3.5.6	Results	84
3.5.7	Comparing GOBI and GOBI*	88
3.6	Summary	88
4	AI Augmented Resource Provisioning	89
4.1	System Model and Problem Formulation	90
4.1.1	System Model	90
4.1.2	Co-Simulator Based Predictions	91
4.1.3	Problem Formulation	91
4.2	The CILP Provisioner	92
4.2.1	Neural Model	92
4.2.2	Two-phase inference	94
4.2.3	Model Training	95
4.2.4	Provisioning in Practice	96
4.3	Performance Evaluation	96
4.3.1	Setup	96
4.3.2	Testbed	97
4.3.3	Results and Ablation Analysis	98
4.4	Summary	99
5	AI Augmented Application Deployment	101
5.1	System Model and Problem Formulation	103
5.1.1	Workload Model	104
5.1.2	Split Nets Placement	105
5.1.3	Problem Formulation	106
5.2	SplitPlace Policy	106
5.2.1	Multi-Armed Bandit Decision Module	107
5.2.2	Reinforcement Learning based Placement Module	110
5.2.3	SplitPlace Algorithm	111

5.2.4	Implementation	112
5.3	Performance Evaluation	112
5.3.1	Experiment Setup	113
5.3.2	Workloads	113
5.3.3	MAB Training	114
5.3.4	Performance Metrics	115
5.3.5	Baselines and Ablated Models	116
5.3.6	Results and Ablation Analysis	118
5.3.7	Sensitivity to λ	119
5.3.8	Sensitivity with α, β	120
5.3.9	Constraining Environments	122
5.3.10	Constraining Workloads	124
5.3.11	Comparing Edge with Cloud	125
5.3.12	Comparing the Impact of Splitting and Placement Decisions on Response Time	126
5.4	Summary	127
6	AI Augmented Task and Workflow Scheduling	129
6.1	Extending GOBI to Stochastic Neural Models and Higher Order Optimization	131
6.1.1	System Model	131
6.1.2	NPN as Heteroscedastic Surrogate Model	132
6.1.3	LCB Exploration	134
6.1.4	Second-Order Optimization	135
6.1.5	The GOSH and GOSH* Schedulers	136
6.1.6	Model Training	137
6.1.7	Dynamic Exploration Weight	137
6.1.8	Scheduling	138
6.2	Extending GOBI to Long-Term Optimization	139
6.2.1	Workload Model	139
6.2.2	Problem Formulation	139
6.2.3	Deep Surrogate Model	140
6.2.4	Workflow Constraint Graph Encoding	140
6.2.5	Scheduling Decision Encoding	141
6.2.6	Time Series Window Encoding	141
6.2.7	Estimating the Objective Score	142
6.2.8	Monte Carlo Learning	142
6.2.9	Building the Monte Carlo Tree using Co-Simulations	143
6.2.10	Model Training	144
6.2.11	Scheduling using the Trained Model	145
6.3	Performance Evaluation	146
6.3.1	Evaluation Setup	146

6.3.2	Workloads	147
6.3.3	Model Training and Assumptions	148
6.3.4	Baselines	149
6.3.5	Results for GOSH	151
6.3.6	Results for MCDS	154
6.4	Summary	160
7	AI Augmented Fault-Tolerance	163
7.1	System Model and Problem Formulation	166
7.1.1	System Model	166
7.1.2	Formulating a Supervised Problem	167
7.1.3	Formulating an Unsupervised Problem	167
7.2	PreGAN and PreGAN+ Supervised Models	168
7.2.1	PreGAN+ Model	170
7.2.2	Offline FPE Training in PreGAN and PreGAN+	171
7.2.3	Online FPE Tuning in PreGAN+	172
7.2.4	Online GAN Training using Co-Simulation	174
7.2.5	Online Inference	175
7.3	DeepFT Unsupervised Model	176
7.3.1	Scheduling and Online Model Fine-Tuning using Co-Simulation	181
7.4	Performance Evaluation	181
7.4.1	Evaluation Setup	181
7.4.2	Evaluation Metrics	182
7.4.3	Model Training and Assumptions	184
7.4.4	Baselines	187
7.4.5	Results for PreGAN and PreGAN+	187
7.4.6	Results for DeepFT	191
7.5	Summary	195
8	AI Augmented Service Resilience	197
8.1	System Model and Problem Formulation	200
8.1.1	System Model	200
8.1.2	Problem Formulation	201
8.2	Generative Optimization Networks	202
8.3	DRAGON Based Worker Resilience	203
8.3.1	Optimizing GONs for fault-detection	203
8.3.2	Offline Model Training	206
8.3.3	Decentralized Fault Tolerance using GON and Co-Simulations	206
8.4	CAROL Based Broker Resilience	210
8.4.1	Node-Shifts	210
8.4.2	Confidence-Aware Model	211

8.4.3	Finding optimal edge topology	212
8.4.4	Confidence Aware Fine-Tuning	213
8.4.5	GON Network	214
8.4.6	Objective function	215
8.5	Performance Evaluation	215
8.5.1	Workloads	215
8.5.2	Fault Injection Module	216
8.5.3	Broker Failure Detection and Recovery	216
8.5.4	Fog Time Series Anomaly Detection (FTSAD) Dataset	216
8.5.5	Implementation and Training	217
8.5.6	Results for DRAGON and DRAGON+	218
8.5.7	Results for CAROL	223
8.6	Summary	226
9	AI Augmented Concurrent Resource Management	227
9.1	System Model and Problem Formulation	228
9.2	The SciNet Model	231
9.2.1	Neural Model and Two-Phase Inference	231
9.2.2	Co-Simulated Oracle	234
9.2.3	Training Recipe	234
9.3	Performance Evaluation	237
9.3.1	Setup	237
9.3.2	Comparison of Training Time and Demand Prediction Loss	239
9.3.3	Comparison of QoS Metrics	240
9.3.4	Ablation Analysis	241
9.3.5	Sensitivity Analysis	241
9.4	Summary	242
10	Conclusions and Future Work	243
10.1	Summary of Contributions	243
10.2	Future Directions	244
	Bibliography	249

This page intentionally left blank.

List of Figures

1.1	Motivation for low compute time of resource management.	35
1.2	Motivation for low memory footprint of resource management.	35
1.3	Research objectives and corresponding chapters. The objectives are in the order of development of fog computing environments.	37
2.1	AI <i>on</i> and <i>for</i> the Fog continuum.	42
2.2	Architecture of a Feed-forward Network.	46
2.3	Architectures of LSTM and GRU Networks.	47
2.4	Architecture of a Graph Neural Network.	48
2.5	Architecture of a Transformer Network.	49
2.6	Architecture of a Generative Adversarial Network.	50
2.7	Overview of layer and semantic splitting strategies	51
2.8	A brief taxonomy of AI methods for Fog systems.	52
3.1	System model	71
3.2	Dynamic Task Workload Model	72
3.3	COSCO architecture	77
3.4	Iteration of GOBI* approach at the start of interval I_t	79
3.5	Convergence plots for GOBI*.	81
3.6	Comparison of GOBI and GOBI* against baselines on framework with 10 hosts	85
3.7	Comparison of GOBI and GOBI* against baselines on simulator with 50 hosts	86
4.1	<i>CILP</i> neural model.	92
4.2	Top level design of the interleaved co-simulation and prediction in <i>CILP</i>	94
4.3	Comparison of QoS parameters (averaged over intervals) of <i>CILP</i> against baselines and ablated models.	97
5.1	Comparison of Layer and Semantic Splits.	102
5.2	SplitPlace System Model	104
5.3	SplitPlace decision and placement problems.	105
5.4	MAB decision workflow.	107
5.5	MAB training curves	114

5.6	Comparison of SplitPlace against baselines and ablated models on physical setup with 50 edge workers	116
5.7	Additional results comparing SplitPlace with baselines and ablated models.	117
5.8	Sensitivity of λ on all models.	120
5.9	Sensitivity of α, β on ablated SplitPlace models.	121
5.10	Fraction of layer-split decisions of the SplitPlace model with increasing λ	121
5.11	Fraction of layer-split decisions of the SplitPlace model with increasing α	122
5.12	Comparison of SplitPlace against baselines and ablated models in Compute, Network and Memory constrained environments.	123
5.13	Distribution of response time in Normal and Compute, Network and Memory constrained environments.	124
5.14	Distribution of SLA violations in Normal and Compute, Network and Memory constrained environments.	125
5.15	Comparison of SplitPlace against baselines and ablated models in settings with only MNIST, FashionMNIST and CIFAR100 workloads.	125
5.16	Distribution of response time in Normal and MNIST only, FashionMNIST only, CIFAR100 only workloads.	126
5.17	Comparing the response time and SLA violation rates for SplitPlace for edge and cloud setups.	126
5.18	Comparing the response time across splitting and placement decisions.	127
6.1	Heteroscedastic Surrogate for Response Time.	132
6.2	LCB metric calculation in GOSH.	135
6.3	Deep Surrogate Model used in the MCDS Scheduler.	140
6.4	Building the Monte Carlo Tree.	142
6.5	Choosing best action and updating the Monte Carlo Tree after simulations.	144
6.6	Comparison of GOSH and GOSH* against baselines and ablated models on physical setup with 10 hosts	150
6.7	Sensitivity Analysis for k	153
6.8	Comparison of the adaptability of various models.	154
6.9	Comparison of the performance with increasing uncertainty.	154
6.10	Comparison of the scheduling overhead of various models.	155
6.11	Comparison of MCDS against baselines on physical setup with 10 hosts.	156
6.12	Comparison of MCDS against baselines on simulated setup with 50 hosts.	157
6.13	Scalability Analysis for MCDS and baseline methods with the number of tasks in each workflow as 10, 50, 100, 500 and 1000.	159
6.14	Sensitivity Analysis for ablated models with number of Monte Carlo iterations (the ψ parameter in our formulation).	159
7.1	The PreGAN Model.	168
7.2	The PreGAN+ Model. Generator and Discriminator models are same as in Figure 7.1.	170
7.3	The DeepFT Surrogate Model.	176

7.4	Visualization of attention weights, truncated with the CPU utilization of four hosts in the system.	180
7.5	Raspberry Pi Cluster as evaluation platform.	182
7.6	Visualization of fault prediction, classification and migration decisions with execution intervals. .	183
7.7	Training curves for the Fault Prototype Encoder.	185
7.8	Fault Prediction using PreGAN and PreGAN+ models with Ground Truth labels.	185
7.9	t-SNE plot of prototypes generated for the test set by the PreGAN+ model.	185
7.10	Confusion matrices for fault classification on the test set.	186
7.11	Training of the GAN network.	186
7.12	Comparison of QoS parameters of PreGAN against baselines.	188
7.13	Sensitivity Analysis for all models with λ (parameter of the discrete Poisson distribution). . . .	189
7.14	Sensitivity Analysis for PreGAN+ with ξ (fraction of labeled data).	190
7.15	Convergence plots for the DeepFT model.	191
7.16	Visualization of the optimization loop for a sample scheduling decision.	192
7.17	Comparison of QoS parameters of DeepFT against baselines.	193
7.18	Sensitivity Analysis for all models with λ (parameter of the Poisson distribution).	194
8.1	Edge Federation Model with 16 edge nodes.	200
8.2	The GON framework.	202
8.3	Neural network used in DRAGON.	204
8.4	Possible node-shifts in CAROL. Nodes marked in red are the high consumption brokers that break-down.	210
8.5	Visualizing the confidence scores and POT threshold values for 1000 scheduling intervals in CAROL.	213
8.6	Neural network used in CAROL.	214
8.7	Training the GON model.	217
8.8	Comparison of QoS parameters of DRAGON against baselines. DRAGON and DRAGON+ trained on complete FTSAD-55 data and tested on AIoTBench based workloads.	220
8.9	Comparison of QoS parameters of DRAGON against baselines for each application type in AIoT- Bench workloads.	220
8.10	Ablation Analysis of different model components of DRAGON.	221
8.11	Sensitivity Analysis of performance of DRAGON and DRAGON+ with window size.	222
8.12	Sensitivity Analysis of performance of DRAGON and DRAGON+ with AdaHessian learning rate.	223
8.13	Comparison of CAROL with baseline methods and ablated models.	224
8.14	Sensitivity Analysis.	225
9.1	Neural network as imitation learning model in SciNet.	231
9.2	Imitation Learning pipeline.	236
9.3	Comparison of training times and test loss.	239
9.4	Comparison of QoS parameters (averaged over intervals) of SciNet against baselines and ablated models.	240

This page intentionally left blank.

List of Tables

2.1	Summary of state-of-the-art methods for AI augmented deployment. Color scheme as per Figure 2.8.	53
2.2	Summary of state-of-the-art methods for AI augmented scheduling. Color scheme as per Figure 2.1.	57
2.3	Summary of state-of-the-art methods for AI augmented dependability. Color scheme as per Figure 2.1.	61
3.1	Host characteristics of Azure fog environment.	82
3.2	Comparison between GOBI and GOBI*	88
4.1	Comparing utilization ratio r_t , cost ϕ_t and QoS score $\hat{Q}_t$ for each interval across all models.	96
4.2	Sensitivity analysis with the weight of the execution cost in the objective function (γ). All standard deviations are <0.05 .	99
5.1	Edge Worker characteristics of Azure Edge Environment.	112
5.2	Comparison of SplitPlace with baseline and ablated models.	118
6.1	Comparison of GOSH and GOSH* against baselines and ablated models on simulator with 50 hosts.	151
6.2	Loss values of various models on test data. The values are averaged over the five-fold cross-validation runs.	152
6.3	Ablation Analysis. Units: Energy - $KW-Hr$, Response Time - scheduling intervals, SLA Violations - percentage, Scheduling Time (cumulative) - seconds.	158
7.1	Performance scores of PreGAN and PreGAN+ and the baseline methods with standard deviation. The best scores are shown in bold.	188
7.2	Comparison of the load on broker node.	191
7.3	Performance scores of DeepFT and the baseline methods with standard deviation. The best scores are shown in bold.	192
8.1	Dataset Statistics	217
8.2	Comparison of detection and diagnosis scores with standard deviations. Results reported on the test set with 80:20 train-test split.	219

9.1	Comparison of QoS parameters (averaged over intervals) of SciNet against baselines and ablated models.	239
9.2	Sensitivity analysis with hyperparameters (γ, ξ, ζ) . All standard deviations are <0.05	241

List of Symbols

I_t	t^{th} scheduling interval
A_t	Active tasks in I_t
W_{t-1}	Waiting tasks at the start of I_t
L_t	Tasks leaving at the end of I_t
N_t	New tasks received at the start of I_t
H	Set of hosts in the Resource Layer with static infrastructure
h_i	i^{th} host in an enumeration of H
a_j^t	j^{th} task in an enumeration of A_t
$\mathcal{U}(h_i^t)$	Utilization metrics of host h_i in I_t
$\mathcal{U}(a_j^t)$	Utilization metrics of task a_j^t in I_t
$Y_t = N_t \cup W_{t-1} \cup A_{t-1} \setminus L_t$	Scheduler input at the start of I_t
$\mathcal{D}(Y_t)$ or $\mathcal{D}$ or S_t	Scheduling decision at start of I_t
$\hat{\mathcal{D}}$	Feasible sub-set of scheduling decision $\mathcal{D}$
$\bar{\mathcal{D}}$	Scheduling decision of GOBI in GOBI* loop
$\mathcal{O}(P_t)$	Objective value at the end of I_t
$\mathcal{S}$	Execution of an interval on simulator
$\mathcal{F}$	Execution of an interval on physical setup
x_t	System state at the start of I_t
W_t	Sliding window at I_t of system states of length k
x_i	i -th state in W_t such that $i \in \{t - k + 1, \dots, t\}$
$\hat{W}_t$	Predicted reconstruction of W_t
f_t	Fault score as the square of the L2 distance between W_t and $\hat{W}_t$
P_t	Prototype vector for fault classification
c_i	i -th class in the set of faults $i \in \{1, \dots, j\}$
c_0	No fault class or No-Anomaly-Prototype (NAP) class
$\mathcal{H}_t$	Set of available hosts in I_t with dynamic infrastructure
$\mathcal{A}$	Application types supported by the cloud platform
A_t^j	Application type of workload $n_t^j \in N_t$ as a one-hot vector
D_t^j	Deployment decision of workload $n_t^j \in N_t$
$\{n_t^j\}_{D_t^j}$	Set of serverless tasks created for workload $n_t^j \in N_t$ based on D_t^j
O_t^k	One-hot vector for the index of $w_t^k \in \{n_t^j\}_{D_t^j}$ for workload $n^j \in N_t$

A_t^k	Application of the workload corresponding to $w_t^k \in W_t$
W_t^k	Feature vector of $w_t^k \in W_t$
$\mathcal{T}_t^i$	Set of tasks placed on host $h_t^i \in \mathcal{H}_t$
H_t^i	Feature vector of host $h_t^i \in \mathcal{H}_t$
$\mathcal{V}$	Available set of VM types
V_t	VM types of all hosts $\mathcal{H}_t$
L_t^i	Provisioning decision for host $h_t^i \in \mathcal{H}_t$
L_t	Set of all provisioning decisions in I_t
D_t^j	Deployment decision for workload $n_t^j \in N_t$
D_t	Set of all deployment decisions in I_t
P_t^k	Placement decision for task $w_t^k \in W_t$
P_t	Set of all placement decisions in I_t
f^{il}	Imitation Learner
f^{sim}	Co-Simulator as a callable function

Software Availability

All code, datasets and result reproduction scripts developed as part of this thesis have been made open-source under BSD 3.0 or CC-BY 4.0 licenses. A list of repository links for each published work is given below.

COSCO	Ch. 3	Code: https://github.com/imperial-qore/COSCO Docker images: https://hub.docker.com/u/shreshthtuli Training datasets: https://doi.org/10.5281/zenodo.4897944
SplitPlace	Ch. 5	Code: https://github.com/imperial-qore/SplitPlace
GOSH	Ch. 6	Code: https://github.com/imperial-qore/COSCO/tree/gosh
MCDS	Ch. 6	Code: https://github.com/imperial-qore/COSCO/tree/workflow
PreGAN	Ch. 7	Code: https://github.com/imperial-qore/PreGAN
PreGAN+	Ch. 7	Code: https://github.com/imperial-qore/PreGANPlus
DeepFT	Ch. 7	Code: https://github.com/imperial-qore/DeepFT
GON	Ch. 8	Code: https://github.com/imperial-qore/GON
DRAGON	Ch. 8	Code: https://github.com/imperial-qore/DRAGON
CAROL	Ch. 8	Code: https://github.com/imperial-qore/CAROL

This page intentionally left blank.

Chapter 1

Introduction

In the past decade, the evolution of our digital lives has accelerated across multiple facets, including efficient computation [Gill et al., 2019], communication [Shi et al., 2020] and transportation [Nguyen et al., 2021], making our lives simpler and more convenient. This evolution has been driven by several factors such as the rising concern for climate change and sustainable computing [Perera et al., 2017], the expected end of Moore’s law for silicon-based compute systems [Theis and Wong, 2017] and the recent lifestyle-changing pandemics [Ndiaye et al., 2020]. With changing user demands and application scenarios, novel techniques are required to fuel further growth for high-fidelity and scalable computation.

As the world population increases, the ability to scale services without compromising performance is critical. To achieve this, increasing the capacity of centralized servers increases costs and provides asymmetric service quality to users that are geographically spread across the globe [Hwang et al., 2013]. To provide region-targeted services and reduce overall service latency, distributed computing paradigms have become popular. Here, computational nodes may be present close to the users, *i.e.*, the *edge* of the network or in geographically distant datacenters, *i.e.*, the cloud servers. Recently, a paradigm that has come to the forefront of research is Fog computing. This is because Fog computing devices connect Edge and Cloud resources, giving rise to a paradigm that holistically considers the entire continuum of resources from Edge to the Cloud, often referred to as *Fog Continuum* in literature [Bittencourt et al., 2018]. We consider the Fog computing paradigm in our work for two main reasons. First, research that develops ways to manage Fog environments tends to also apply to purely Edge or Cloud paradigms. Second, literature suggests that Fog environments are the future of computing and require further research to provide highly-efficient and mission-critical services [Firdhous et al., 2014].

This thesis tackles the challenges of delivering efficient, reliable and large-scale computing using effective *resource management* in fog environments. This entails developing algorithms to decide, for instance, when to provision or disable compute nodes, how to select the type of workload to execute, where to place incoming applications on the infrastructures or run remediation steps to mitigate the effects of system failures. As such systems become more complex and less predictable, these algorithms need to adapt to the prevailing circumstances, non-stationary workloads and dynamic user demands. Literature has shown evidence to suggest that data-driven methods such as those that utilize Artificial Intelligence (AI) are able to extract hard-to-encode patterns from data and adapt to changing environments [Nayeri et al., 2021]. This means that AI based methods are increasingly coming to the fore in the domain of resource management. AI enables us

to build intelligent policies to automate the process of making various decisions for large-scale computational infrastructures. Typically AI is based on data-driven models that require several execution traces to *learn* the system behaviors and characteristics. However, generating such data is time consuming on a physical platform. To this end, we investigate the idea of coupled simulation, where a simulated replica of the infrastructure can be used to run discrete event based emulations of resource management decisions to quickly generate additional data required for training AI models [Tao et al., 2019]. Here, coupling means that for every change in the physical nodes, the corresponding change is also reflected in the simulated nodes in real-time.

However, the problem of effective resource management in Fog environments is hard. When deploying AI-based applications on the Fog continuum or leveraging AI to manage running applications, novel resource management issues arise corresponding to the dependability of systems service quality. Recently, application demands have shifted from either high-accuracy or low-latency to both of these together, termed as HALL (high-accuracy and low-latency) service delivery [Gill et al., 2019]. Given the prevalence and demand of AI based workloads, serving them on a public cloud with tight bounds of latency, throughput and cost is becoming increasingly challenging [Gunasekaran et al., 2020]. Specifically, to solve the problem of providing HALL services, recent work proposes to integrate large-scale deep learning models with edge-devices [Gunasekaran et al., 2020, Laskaridis et al., 2020, Liang et al., 2020]. However, even the most recent approaches either provide a low latency mode or a high-accuracy mode [Laskaridis et al., 2020, Gunasekaran et al., 2020] and struggle to provide the benefits of both modes at the same time. Another key challenge is that mobile edge devices face severe limitations in terms of computational and memory resources as they rely on low power energy sources like batteries, solar or other energy scavenging methods [Abbas et al., 2017, Mao et al., 2016]. This is not only because of the requirement of low cost, but also the need for mobility in such nodes [Khanna et al., 2020]. Further, in fog environments with unreliable compute nodes, physical machines may break-down or resource contention may arise leading to delays in service delivery. To this end, AI and co-simulation can facilitate decision making in highly non-stationary scenarios or in large-scale settings where a simulated replica may give critical insights on how decisions make affect Quality of Service (QoS) metrics. As part of this thesis, we explore the latest trends in the domain of AI and co-simulation augmented resource management and the challenges it presents to deliver upon the promise of improving QoS of existing and next-generation infrastructures.

1.1 Motivation

The objective of this thesis is to efficiently solve resource management problems in large-scale distributed computing environments. To do this, we focus on two main aspects of resource management: low decision time and low memory footprint. We motivate these aspects through two examples.

Motivating Example 1 (low decision time). Prior work demonstrates that for Fog environments, the decision time increases exponentially with the number of fog devices [Beloglazov and Buyya, 2012, Han et al., 2018]. This can significantly impact SLA violations as modern workloads require methods to make resource management decisions quickly. This is because modern workloads seldom follow stationary distributions in terms of their computational demands. This further exacerbates the resource management problem in Fog environments. To demonstrate this, we present a motivating example where we test three approaches on a simulated platform [Gupta et al., 2017] (see Figure 1.1). We test the popular heuristic based approach LR-

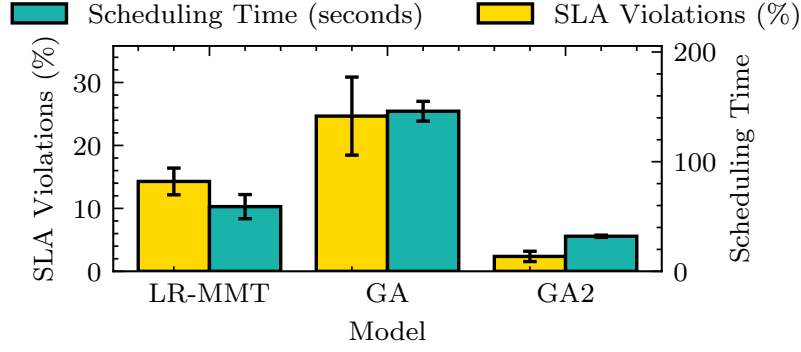


Figure 1.1: Motivation for low compute time of resource management.

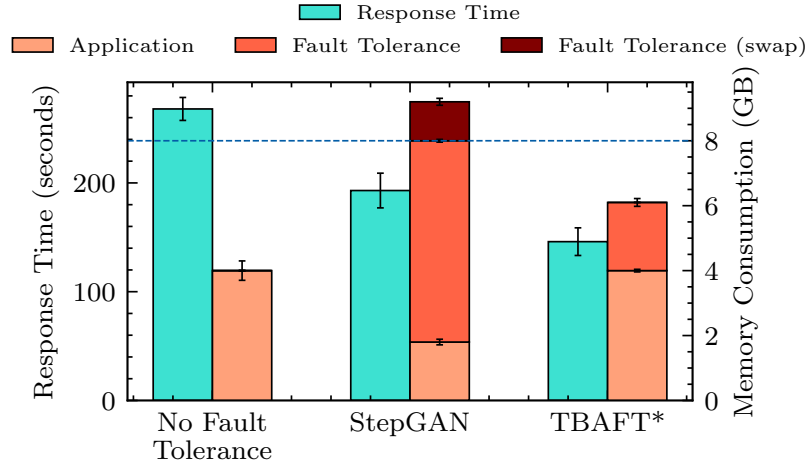


Figure 1.2: Motivation for low memory footprint of resource management.

MMT [Beloglazov and Buyya, 2012] and the evolutionary approach using genetic algorithm (GA) [Han et al., 2018] with 50 simulated virtual machines (details in Section 3.5.1). The former represents the broad range of heuristic-based approaches that can quickly generate a scheduling decision, but cannot adapt. The latter can quickly adapt but have high scheduling overheads. As in [Han et al., 2018], we provide the GA the utilization metrics of all hosts and tasks and aim to minimize the average response time. We assume that the container allocation time is negligible and the total response time of a fog task is the sum of the scheduling and execution times. We consider volatile workloads as in Chapter 3. For the approaches considered, the scheduling time can range from 100 to 150 seconds with SLA violation rate of up to 24%. However, if we could bring down the scheduling times to 10 seconds with the same decisions as GA (referred to as GA2), the SLA violation rate goes down to $\sim 2\%$. This shows how decision time is a key factor that limits the extent to which SLA can be optimized, thus highlighting the potential for an approach that can respond quickly in volatile scenarios and has low scheduling overheads, leveraging the fast and scalable execution times of AI methods. We develop novel ideas to reduce scheduling overheads and improve service quality in this thesis.

Motivating Example 2 (low memory footprint). Other prior works demonstrate that by increasing the number of Fog devices and the amount of operational data, the resource management methods tend to be more sophisticated, subsequently causing higher memory consumption [Audibert et al., 2020, He et al., 2020, Hu et al., 2021]. This load on the limited working memory of Fog devices also affects the functioning of the applications running on edge systems. To demonstrate this, we run simple experiments on four resource-limited

Fog devices (details in Section 7.4.1). Each node has 8 GB RAM, acts as its broker and runs a fault-tolerance model (details in Chapter 8). Figure 1.2 shows the results corresponding to three different cases, presenting average response time in blue and memory consumption in red. The first case is when we do not run any fault tolerance model. All the memory of edge nodes is utilized by the running applications in this case. In the second case we run a popular deep learning based StepGAN fault-tolerance model [Feng et al., 2021], which has a large memory overhead of 7.4 GB. Even though it reduces the average response time by 28%, the model consumes 1.2 GB of slow virtual memory (swap space), limiting its optimization capability. In the third case, we run the heuristic-based approach TBAFT [Rawat et al., 2021], but use pre-computed fault labels from StepGAN (denoted by TBAFT*). This shows the performance in a case when we have accuracy of the deep learning model with the memory-efficiency of a heuristic approach. In this case, as no swap space is utilized, the average response time is much lower, giving 24% improvement compared to the StepGAN case. This example highlights the potential for an approach that is as memory-efficient as heuristic approaches, but that does not compromise on performance, matching fault detection scores with those of state-of-the-art deep learning techniques. We develop memory-efficient resource management solutions in this thesis.

1.2 Research Objectives

A vast amount of literature exists that aims to improve existing fog systems using AI across a wide range of domains, such as efficient resource provisioning, application deployment, task placement, and service management. However, prior work faces several limitations, such as the inability to generalize in unseen scenarios, tendency to fail in large-scale settings and poor performance in highly volatile settings. Further details are mentioned in Chapter 2. To address the limitations of the prior work, the fundamental hypothesis that we propose and empirically evidence in this work is succinctly phrased as follows:

Co-Simulation and Deep Learning can deliver major enhancements to QoS of Fog computing environments

Our primary focus lies at the intersection of the three fields of AI, Co-Simulation and Fog Computing, particularly for resource management decision making to optimize Fog system performance. To give evidence for the above hypothesis, we categorize the resource management problem into three scopes and identify critical research objectives within these scopes. Our objectives are specific to Fog computing and not general distributed systems due to the assumptions in Fog environments of having user mobility, full environment observability, finite run-time of tasks and the presence of resource-limited nodes at the edge and resource-abundant resources in the cloud. Thus, our objectives are to leverage AI and Co-Simulation to solve:

1. **Deployment** deals with intelligent resource provisioning and versioning of workloads on Fog infrastructures to optimize QoS [Luo et al., 2021, Calheiros et al., 2014]. This objective is divided into two sub-goals.
 - (a) The first sub-goal is to proactively predict the required resources for seamless task execution and minimize operational costs incurred by service providers and users [Luo et al., 2021, 2020].
 - (b) The second sub-goal is to be able to efficiently deploy disparate applications to optimize system QoS [Shao and Zhang, 2020, Eshratifar et al., 2019].

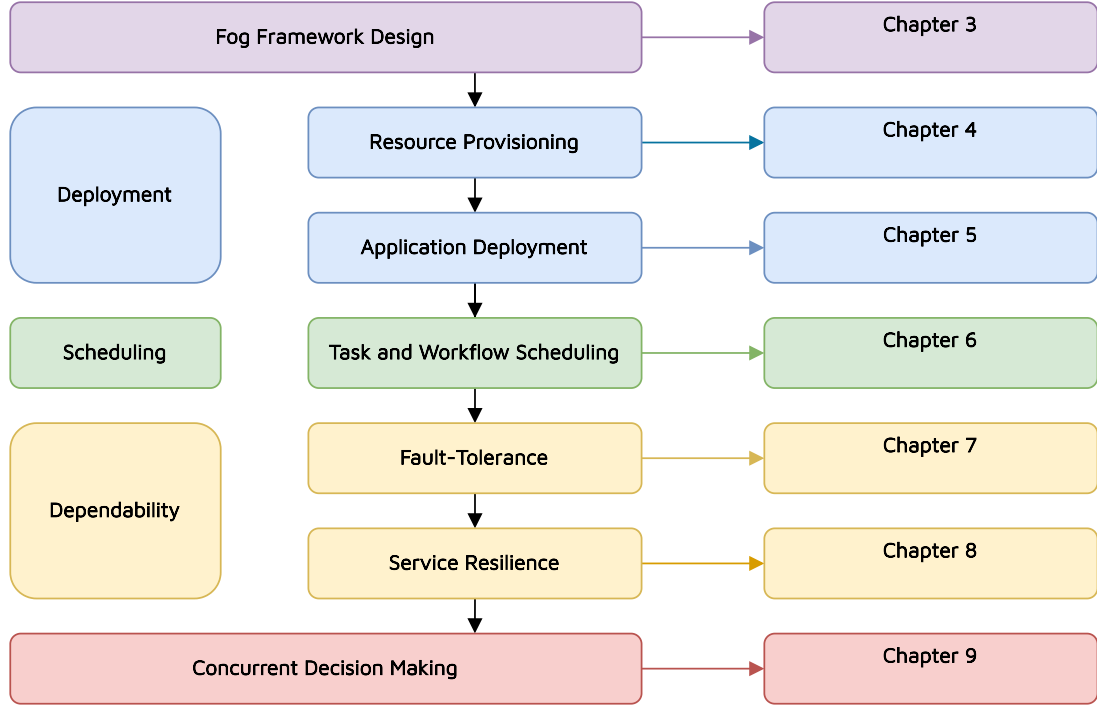


Figure 1.3: Research objectives and corresponding chapters. The objectives are in the order of development of fog computing environments.

2. **Scheduling** entails the placement of deployed tasks on the provisioned resources as part of objectives 1(a) and 1(b) [Basu et al., 2019, Kadota et al., 2018, Matrouk and Alatoun, 2021]. Schedulers should be able to efficiently place tasks on heterogeneous distributed resources and adapt to volatile workloads and dynamic setups [Xiong et al., 2020].
3. **Dependability** aims at securing, preventing and recovering from failures the deployed and scheduled workloads in a fog environment [Du et al., 2020, Bakhshi et al., 2019]. This is divided into two sub-goals.
 - (a) The first sub-goal is to be able to augment scheduling decisions by predicting and preemptively migrating running tasks from those hosts at which there may be resource contention.
 - (b) Even with fault-tolerance, system nodes may still fail. To ensure availability, we proactively predict node failures and balance load by updating the fog network architecture.

Another aspect we consider is to make decisions along disparate aspects concurrently for more informed resource management. We refer to this as *concurrent decision making*. We now map the above research objectives to the specific technical chapters in this thesis.

1.3 Contributions

Based on the defined scopes and the thesis of this work, we provide discussions on the technical contributions and experimental evidence to validate the above claim. A summary is given in Figure 1.3. The chapter order follows the *deployment order* of Fog environments, wherein we first provision resources, then deploy applications on these resources, schedule the tasks corresponding to the deployed applications, run fault-tolerance and resilience mechanisms.

1. **Chapter 3** presents a fog computing framework, namely COSCO, that we utilize and extend throughout this thesis. We also present an idea of gradient-based optimization (GOBI) that enables us to search the decision space using deep neural network based surrogate models to maximize system QoS scores.
2. **Chapter 4** presents an approach, namely CILP, to efficiently leverage co-simulation and an imitation learning model and develop a state-of-the-art resource provisioning model. This chapter aims to achieve objective 1(a).
3. **Chapter 5** throws light on a technique, namely SplitPlace, to optimize QoS settings when deploying workloads on distributed compute nodes on a provisioned infrastructure. This chapter aims to address objective 1(b).
4. **Chapter 6** presents two solutions, namely GOSH and MCDS, to leverage higher-order and long-term optimization to schedule tasks within provisioned resources and deployed workloads. The presented approaches achieve state-of-the-art QoS scores and achieve objective 2.
5. **Chapter 7** shows two approaches, PreGAN and DeepFT, for fault-tolerant computing. The former for supervised fault-detection and classification and the latter for unsupervised settings. These solutions are for objective 3(a).
6. **Chapter 8** presents to techniques, DRAGON and CAROL, for service resilience in fog computing environments. The former for worker resilience and the latter for broker resilience. Both are to address objective 3(b).
7. **Chapter 9** demonstrates how to concurrently take decisions across the different scopes, *i.e.*, provisioning, deployment and scheduling.

The key technical advance we propose in this thesis is the synergistic use of data-driven AI models and co-simulation to solve resource management problems efficiently. Specifically, we build upon the idea of using deep neural networks as surrogate models that approximate performance metrics of a fog system. As highlighted by the motivating example (see Figure 1.1), prediction accuracy and decision making time have significant impact on system QoS. We use a co-simulator to expose such models to scenarios unseen at training. Unlike prior work that trains an AI model on a static training dataset, data generated by a co-simulator allows us to have a robust training procedure wherein the models are familiar with the performance metrics in settings that might arise at test time. This allows our methods to generalize to previously unseen environment settings, possibly with workloads or host characteristics different from training time. Further, this enables us to periodically fine-tune AI models in non-stationary settings using additional co-simulated data based on the latest system trends.

Additionally, we use gradients with respect to decisions to optimize in the decision space. Unlike prior work that uses gradient-free strategies, a gradient-based strategy enables us to have a goal-directed search, furnishing lower decision times than prior work. This enables us to reduce the time to enact decisions and consequently minimize the overall response time of workloads (see Chapters 3, 5 and 6). Further, in specific problems, we use co-simulators to test several candidate decisions and generate near-optimal solutions to train imitation learning AI models. In contrast to prior work that uses multi-step optimization strategies that converge after several iterations, neural network based imitation learning models allow us to predict actions in a single inference step.

This allows us to forgo searching in the decision space using surrogate models and, instead, directly predict decisions to reduce decision time further and improve system QoS (see Chapters 4 and 9).

Furthermore, we train generative models using co-simulators and generate decisions based on specific system characteristics, for instance, fault-types in Chapter 7. Utilizing the several advances in generative modeling allows us to inject additional knowledge, such as the specific component (CPU, RAM or Disk) that might cause contention and system slowdown. Generative models allow us to make more informed decisions for apt remediation in dynamic systems. We build few-shot fault classification models for informed decision generation, which can be tuned using limited data, allowing us to adapt them quickly in volatile scenarios. This is a significant advance over prior methods that leverage fixed fault classification methods and are oblivious to changes in the fog system. Moreover, we modify adversarial generative models using gradient optimization to forgo having one of the two neural networks typically in such models (see Chapter 8). This allows us to significantly reduce memory overheads and make decision making feasible in resource limited settings. Additional advances in leveraging graph based neural networks allow us to capture system trends more accurately than prior work, giving more precise fault predictions and, consequently, more dependable fog systems.

We finally summarize the contributions, conclude and provide future research directions in Chapter 10. The technical advances discussed in these chapters highlight the impact of the novel AI and Co-simulation models developed on resource management problems to improve the QoS of Fog systems.

All contributions in this thesis can be utilized by Edge, Fog and Cloud computing service providers as well as users. The service providers can utilize the state-of-the-art provisioning and deployment solutions to reduce operational costs and energy consumption of their infrastructure. Service providers can also utilize the fault-tolerance and service resilience mechanisms to ensure robust service delivery. Examples of such providers include Microsoft Azure, Google Cloud Platform, Amazon Web Services and Netflix. The industries that leverage the computational resources of such service providers can leverage the developed scheduling solutions to reduce running costs and minimize the impact on SLA violations for their running services. Example application scenarios include smart-city development, healthcare and surveillance.

This page intentionally left blank.

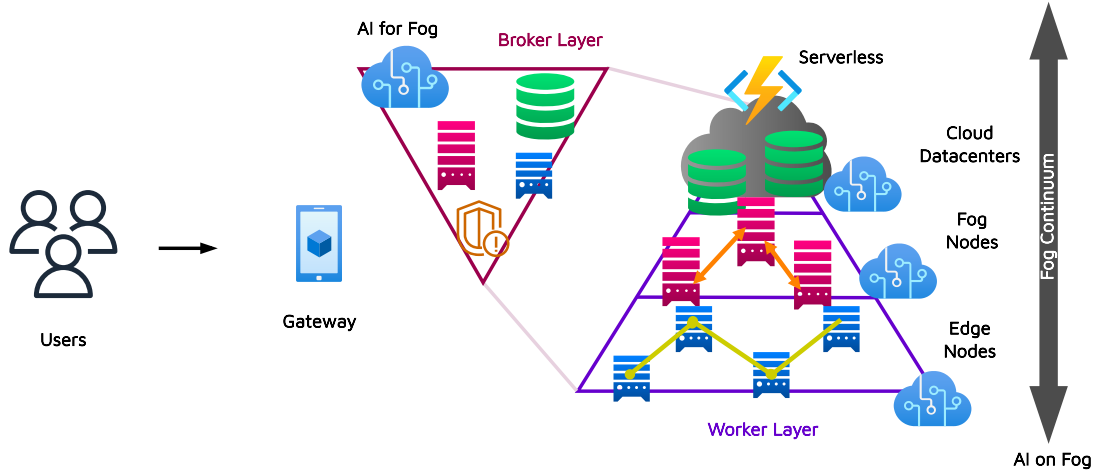
Chapter 2

Background and Related Work

In this chapter, we present the various computing paradigms that form the Fog continuum, service architectures and parameters offered from the systems aspect for AI methods to exploit and optimize the overall QoS. These details allow us to build the context and assumptions required for the subsequent chapters. We also provide various simulators and physical frameworks for AI *on* and *for* Fog computing research as described in Chapter 1. We demonstrate their merits and demerits, and justify how the COSCO framework (presented in Chapter 3), which provides AI and co-simulation primitives, is critical for our work. Finally, we provide details on how AI has been used for resource management in fog environments in Section 2.3 and their limitations to establish the research gap. We present a literature review, providing the directions of works for the objectives established in Chapter 1.

A typical Fog environment consists of two computational layers: broker and worker (see Figure 2.1). The worker layer consists of generic compute nodes that execute incoming applications by processing incoming data from the users and return the results *via* gateway devices. The broker layer consists of compute nodes that monitor and manage the back-end infrastructure, including the worker nodes. This includes deciding where to deploy/place incoming applications as tasks or migrate running tasks to optimize system performance. AI-based augmentation of Fog systems has traditionally been in two major directions. First, where AI models have replaced conventional applications, for instance Deep Neural Networks (DNNs) have replaced prior methods [Shi et al., 2020, Park et al., 2018, Amini et al., 2020], giving fast, scalable and accurate results. This entails augmenting the workloads that are run *on* the Fog worker nodes, and hence we call this domain *AI on Fog*. AI on Fog has been a key driving factor many AI based practical deployments, such as self-driving cars, smart-cities and automated surveillance systems [Wang et al., 2020]. The other domain is where the AI models are used to determine optimal workload placement, service level schedules and fault remediation steps. This augments the resource management services at the broker layer *for* decision making, and hence we call this domain *AI for Fog*. This domain has been crucial for efficient resource management for modern distributed services such as Netflix and cloud platforms [Varghese and Buyya, 2018]. We now elucidate the challenges presented by each paradigm.

AI on Fog. This domain is primarily concerned with the applications running on the worker layer of a Fog system. As modern applications have turned heavily dependent on AI-based models, specifically those that utilize deep learning, we observe that DNNs are becoming the backbone of many industrial tasks and activities [Gill et al., 2019]. As the computational capabilities of devices have improved, new deep learning

Figure 2.1: AI *on* and *for* the Fog continuum.

models have been proposed to provide improved performance [Zhu et al., 2018, Li et al., 2019]. Moreover, many recent DNN models have been incorporated with mobile edge computing to give low latency services with improved accuracy compared to shallow networks, particularly in complex tasks like image segmentation, high frame-rate gaming and traffic surveillance that require latency in the order of 10-100 milliseconds [Khanna et al., 2020]. The performance of such neural models reflects directly on the reliability of application domains like self-driving cars, healthcare and manufacturing [Gill et al., 2019, Kraemer et al., 2017]. The integration of such AI models with various computational systems has led to the rise of EdgeAI services, *i.e.*, applications that utilize AI to process data at the edge. To provide high accuracy, neural models are becoming increasingly demanding in terms of data and compute power, resulting in many challenging problems. To accommodate these increasing demands, such massive models are often hosted as web services deployed on the public Cloud [Zhang and Zhang, 2017]. On the other hand, mobile edge devices in typical Fog deployments face severe limitations in terms of computational and memory resources as they rely on low power energy sources like batteries, solar, or other energy scavenging methods [Mao et al., 2016]. This is not only because of the requirement of low cost, but also the need for mobility in such nodes [Khanna et al., 2020]. In such systems, it is possible to handle the processing limitations of massive AI models by effective preemption and prolonged job execution. However, memory bottlenecks are much harder to solve as shown in prior work [Shao and Zhang, 2020]. In a practical distributed edge environment where storage spaces are typically mapped to a network-attached-media, a large virtual memory imposes high network bandwidth overheads that make performing large-scale distributed computations hard [Laskaridis et al., 2020]. Thus, as part of this chapter, we explore various methods developed to efficiently deploy and manage AI-based applications on Fog infrastructures by possibly decomposing DNNs and running distributed training and inference [Li et al., 2020].

AI for Fog. This domain is primarily concerned with resource management level decision making at the broker layer of a Fog system. The problem of efficiently managing Fog resources is hard [Gill et al., 2019]. One of the challenges we face in such a system is the heterogeneity of resources across the Edge and Cloud [Li et al., 2020, Kaur et al., 2020, Hosseinalipour et al., 2020]. Another challenge in industrial settings is to deliver low latencies for time-critical applications, for instance, healthcare, robotics and smart-cities. These challenges are exacerbated by modern-day applications, wherein the workloads are highly dynamic and host machines having volatile resource capabilities. Furthermore, as applications become more demanding and privacy-sensitive, Fog

devices have become more prone to breakdowns, malicious attacks and intrusions [Zhang et al., 2019]. This entails taking recovery steps required to deal with the diverse effects of system faults, such as network packet drops, memory errors or disk failures requiring different remediation steps. So far, the industrial and research landscape of Fog resource management has been dominated by heuristics and classical optimization-based methods. Such approaches have low scheduling times and work well for general cases, but due to steady-state or stationarity assumptions, they provide poor performance in non-stationary heterogeneous environments with dynamic workloads [Mahmud et al., 2018]. To address these challenges, various AI methods have been recently proposed that utilize adaptive schemes based on evolutionary methods and reinforcement learning. These methods adapt to changing scenarios and offer promising avenues for dynamic optimization [Fox et al., 2019]. For accurate and scalable modeling of the Fog environment, such methods use deep learning-based local search or learning models that approximate an objective function such as energy consumption or response time [Liu and Wang, 2020, Basu et al., 2019]. However, the most accurate AI methods typically have high decision times or resource footprints, making it hard to deploy them in budget or resource-constrained settings. Thus, we also explore various advancements in AI methods for efficient resource management in Fog systems [Li et al., 2020].

Simulation-Driven AI. We highlight in this thesis that using AI in tandem with a co-simulator (or digital-twin or simulator) enables us to solve several challenges such as quick testing, generating additional data and exploring unseen scenarios, for improved accuracy and optimize decision making. To test out a decision in a physical environment, we need to enact the decision and check how it affects the QoS metrics of interest such as energy consumption, response time or operational costs. However, in many cases, the decisions that we test may not be optimal and in online decision making systems, this would affect the overall service quality. To avoid this, we can create a simulated replica of the cyber-physical infrastructure that emulates the effects of resource management decisions on the simulated platform. This gives us two major benefits: (1) reduced decision testing time, and (2) robust decision making. Using a discrete event-based simulator allows us to simulate actions much faster than actual execution in the environment. This enables us to generate insights and quickly test candidate decisions. For instance, if we need to check the effects of a decision after, say ten minutes, we need not wait for this duration in a simulated environment, thanks to recent event-driven simulations [Mansouri et al., 2020]. Further, some decisions may lead to unforeseen system failures that could lead to pernicious systems downtimes. Validating certain decisions by enacting them in a simulated environment allows us to eschew making such decisions in the practice.

2.1 System Background

We now provide some background on Fog computing and the related computational paradigms, the constraints they impose and the challenges they present that we tackle in subsequent chapters.

2.1.1 Fog Computing

We now describe the computing paradigms of Cloud, Edge and serverless. We mention their merits and limitations to motivate the need for a continuum of resources.

Cloud Computing. The Cloud computing paradigm consists of an inter-connected and virtualized pool of resources (computing, storage, network, etc.) that can be dynamically provisioned on-demand, as per user

specifications and with minimal management effort [Buyya et al., 2009] (see top tier in Figure 2.1). Cloud resources may be publicly accessible or privately deployed. Traditionally, workloads are run in Cloud nodes as distinct virtual machines (VMs), allowing Cloud providers to migrate running workloads from one Cloud node to another for load balancing and tuning various QoS parameters. A significant challenge in the Cloud paradigm is that Cloud datacentres are located multiple hops away from the IoT devices, which increases the data transmission time between the devices and the Cloud instances hosting the applications. To overcome these limitations of Cloud computing, a new paradigm called Edge computing was introduced to meet the service requirements of large-scale IoT applications.

Edge Computing. Recently, Edge computing [Satyanarayanan, 2017] has grown dramatically. The network edge, defined as the computational layer that resides closest to the end-user, is where most data sources are present. Edge computing follows the data gravity principle, *i.e.*, it moves the computational resources close to the data sources or the network edge (see bottom tier in Figure 2.1). This leads to a multitude of benefits [Hu et al., 2017]. First, it offers low response times, possibly in milliseconds, crucial for time-critical tasks such as flight control, healthcare, autonomous cars and gaming [Li et al., 2019, Gill et al., 2019]. Second, it allows us to build reliable systems where service resilience is provided at the node level, allowing other compute devices to act as backups and ameliorating performance degradation by reducing service downtimes using failover and fallback mechanisms [Bagchi et al., 2019]. A major challenge at the Edge is that devices have limited computational capabilities and therefore suffering significantly under stress. There also exists a vast amount of devices in an IoT system, giving rise to bandwidth contentions [Belcastro et al., 2021].

Serverless Computing. Serverless computing emerged as a solution for the complexity of Cloud and Edge computing that hides server usage and runs user codes on-demand automatically with high scalability at the function level, such that the users are only billed for the code execution time [Castro et al., 2019]. It is agnostic to the specific set of resources we utilize, Edge or Cloud. Platforms and architectures have been recently proposed in the literature to extend serverless capabilities to Edge computing [Javadi et al., 2020, Cicconetti et al., 2020]. In serverless, the applications use precisely the amount of resources needed at any one point in time and charge accordingly, making the costs proportional to the exact resource usage [Hendrickson et al., 2016]. Even though the tight integration in serverless makes it friendly for user, it also makes it hard for developers to optimize QoS when running serverless applications due to the lack of data management in serverless. Unlike containers and VMs that allow independent monitoring of each running service, serverless frameworks abstract out the active functions in the system, reducing the viability of tuning them for performance optimization.

Fog Continuum. Fog is a parallel and distributed computing paradigm introduced by CISCO in 2012 as an interface between the Cloud and Edge computing systems to support latency-critical and resource-hungry application services by providing an interface between the computation and storage offered by Cloud and Edge [Bonomi et al., 2012]. Fog introduces a hierarchical architecture with an intermediate layer between end-users and Cloud datacentres which utilizes computational, storage, and networking resources that reside within the path connecting users to the Cloud [Mahmud et al., 2018]. These resources known as Fog nodes include gateways, switches, routers, nano datacentres, Cloudlets, etc. Unlike traditional fog or mist platforms, *fog continuum* is an umbrella term that includes edge only, cloud only and hybrid edge-fog-cloud resources. As Fog resources are distributed, heterogeneous, and resource-constrained compared to Cloud data centers, efficient resource provisioning and application placement algorithms is vital.

2.1.2 Application Scenarios

Fog computing environments are popularly used in several application scenarios, some of which we list below.

Smart Cities. Modern cities can be managed more efficiently using Fog computing. For instance, the traffic-signalling systems can be made congestion aware using cameras and compute nodes at traffic junctions that predict vehicular flow and adapt signal timings autonomously [Haouari et al., 2018]. Further, buildings can be made smarter by managing the thermostats and light systems based on the room occupancy.

Healthcare. Advances in IoT sensors allow the integration of sophisticated body sensors with wearables. The data from such wearables can be utilized to diagnose ailments and predict the spread of infectious diseases, such as COVID-19. The wearables are also applied to the patients in hospital to give continuous information about their vitals, blood glucose levels and many more. Fog computing is useful to these wearables as it ensure data to be delivered without delay during emergency cases [O'herrin et al., 2004].

Surveillance. Shopping malls and public places usually are installed with surveillance camera to provide video images on public behaviours. To prevent latency, fog computing is essential in helping identifying abnormalities in crowd patterns and immediately alert authorities on the situation [Chen et al., 2016].

2.1.3 Optimization Parameters

We now describe the various QoS parameters of a Fog system that we expect an AI-based resource manager to optimize for ideal system performance.

Response Time. This parameter indicates the service delivery time. Within distributed Fog environments, the response time of a service depends on multiple parameters such as data transmission time, propagation time, processing time, and service deployment time [Mahmud et al., 2020]. In Cloud and Edge environments, consumers and providers negotiate these QoS parameters to establish a Service Level Agreement (SLA) [Buyya et al., 2009]. SLAs are critical in deadline-oriented tasks such as flight management systems, self-driving car networks and gaming.

Cost. The cost of using Cloud and Edge environments depends on the type of service used by the consumer and the pricing model (*i.e.*, on-demand, reserved or spot pricing) employed by the service provider [Mahmud et al., 2020]. Thus, Fog application placement aims to reach a tradeoff between cost and response time, to minimize the cost of deployment while satisfying deadline requirements of the applications [Deng et al., 2020].

Energy. IoT is highly scalable, with a large number of sensors generating a significant amount of data for processing. This results in higher energy consumption and carbon footprint in Cloud datacenters during data transmitting and processing [Oma et al., 2018]. Fog continuum, with its distributed architecture, has the potential to achieve higher energy efficiency by relying on low-power edge nodes when possible [Gill et al., 2019, Mahmud et al., 2020], but is limited by the energy and computation capacity of the Fog nodes.

Reliability. Reliability of Fog systems is quantitatively defined as a probability measure of how frequently a system delivers the services it has been designed for. Edge and Fog nodes are prone to different types of failures, including hardware failures, software failures, network failures and resource overflow [Bagchi et al., 2019, Carvalho et al., 2021]. Even a small failure probability per node is cascaded by the presence of a large number of interconnected nodes. Therefore, reliable Fog systems must be implemented that has a low failure rate, and when it does fail, it recovers quickly.

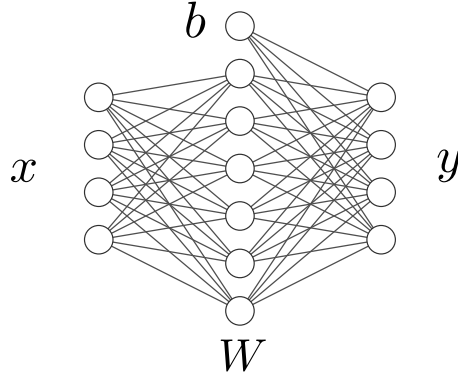


Figure 2.2: Architecture of a Feed-forward Network.

Accuracy. We use accuracy as a general term to highlight the performance of an AI or ML service in terms of the closeness of model output to the true or expected outputs. This can include classification performance, detection accuracy or prediction error. Some application use-cases, such as healthcare, require extremely accurate results. On the other hand, other scenarios, for instance autonomous systems, need near real-time inference. AI models have distinct performance and inference times.

2.2 AI Background

In the rest of the thesis we extensively leverage certain concepts of AI. Specifically, we use DNNs as function approximators that map an input x to an output $y = f(x; \theta)$ via a sequence of transformations. Here, θ corresponds to the weights or parameters of the neural network. There are different types of neural networks that we describe next.

2.2.1 Neural Networks

There are several types of neural networks. However, we only discuss here the ones used in this thesis.

Feed-forward Networks. This is the most basic form of neural network layer that takes an input vector x and uses parameters: weight W and bias b to generate an output as shown in Figure 2.2

$$y = W \cdot x + b. \quad (2.1)$$

We stack multiple such layers to create a *deep* neural network. As a result of stacking several of these layers deep neural networks of sufficient capacity are able to approximate functions of arbitrary complexity, specifically predicting QoS metrics in Chapter 3 and action probabilities for resource management in Chapter 4. In order to achieve this, however, one needs to include non-linear activation functions between the linear layers. Some common activation functions that we use in this thesis are defined below:

$$\begin{aligned} \text{ReLU}(x) &= \max(0, x), \\ \text{Sigmoid}(x) &= \frac{1}{1 + e^x}, \\ \text{Tanh}(x) &= 2 \cdot \text{Sigmoid}(2x) - 1. \end{aligned} \quad (2.2)$$

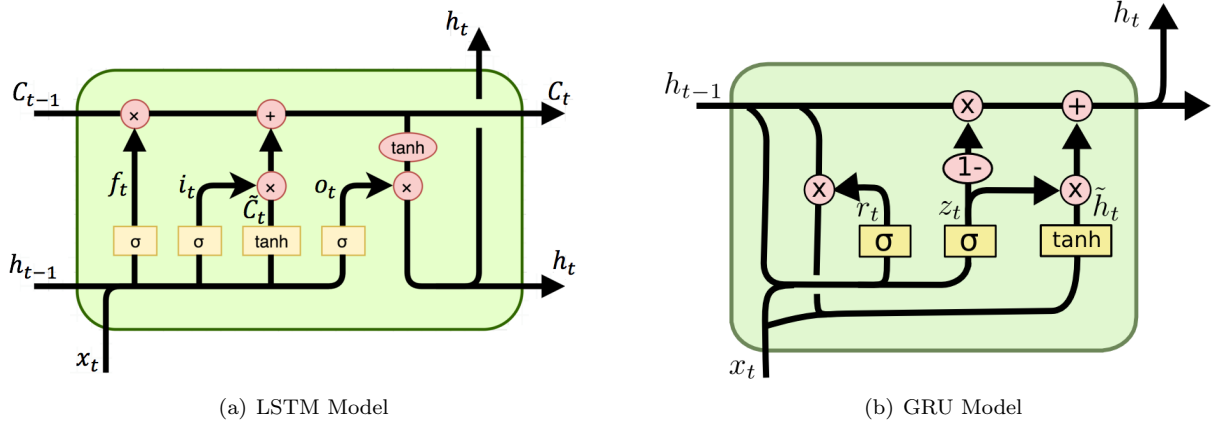


Figure 2.3: Architectures of LSTM and GRU Networks.

The ReLU function is a popular function with low compute overhead and unbounded output, ideal for training large neural models as it does not face the issue of *vanishing gradients* when the gradient updates tend to zero and the weights do not get updated [Goodfellow et al., 2016]. The Sigmoid function has the output range between 0 and 1, whereas, Tanh has between -1 and 1. Such functions are used when we need a bounded output of the neural model. We repeatedly use feed-forward networks in Chapters 3-9.

Recurrent Neural Networks. For a sequence of input, possibly the time-series of states of a fog environment, we may rely on recurrent neural models. Popular models include Long-Short-Term-Memory (LSTM) [Hochreiter and Schmidhuber, 1997] and Gated-Recurrent-Unit (GRU) [Li et al., 2015]. An illustration is shown in Figure 2.3¹. The LSTM cell generates a *hidden-state* h_t for each input x_t , where t denotes the timestamp of the time-series data.

$$\begin{aligned}
 f_t &= \text{Sigmoid}(W_f x_t + b_f), i_t = \text{Sigmoid}(W_i x_t + b_i), \\
 o_t &= \text{Sigmoid}(W_o x_t + b_o), \tilde{c}_t = \text{Sigmoid}(W_c x_t + b_c), \\
 c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, h_t = o_t \odot \text{Sigmoid}(c_t),
 \end{aligned} \tag{2.3}$$

where $\odot$ denotes element-wise multiplication or Hadamard product. Here c_t captures the long-term memory over the time-series data. f_t denotes the output of the *forget gate*, that is a weight to the long-term memory. i_t is the output of the *input gate* that combines the updated long-term memory $\tilde{c}_t$. Finally, h_t is generated by the *output gate* that combines the long-term memory and inferred input o_t .

A GRU is a simplification of this reducing the parameter set of the neural model. It uses only *reset* and *update* gates.

$$\begin{aligned}
 z_t &= \text{Sigmoid}(W_z x_t + b_z), r_t = \text{Sigmoid}(W_r x_t + b_r), \\
 \tilde{h}_t &= \text{Tanh}(W_h x_t + b_h), h_t = z_t \odot \tilde{h}_t + (1 - z_t) \odot h_{t-1}.
 \end{aligned} \tag{2.4}$$

We typically use LSTMs to capture longer-range patterns, whereas GRU when there is a requirement of low memory footprint or faster-to-tune model. We utilize LSTMs in Chapter 3. We also utilize GRU within graph neural networks as we describe next.

¹Images taken from <http://dprogrammer.org/rnn-lstm-gru>.

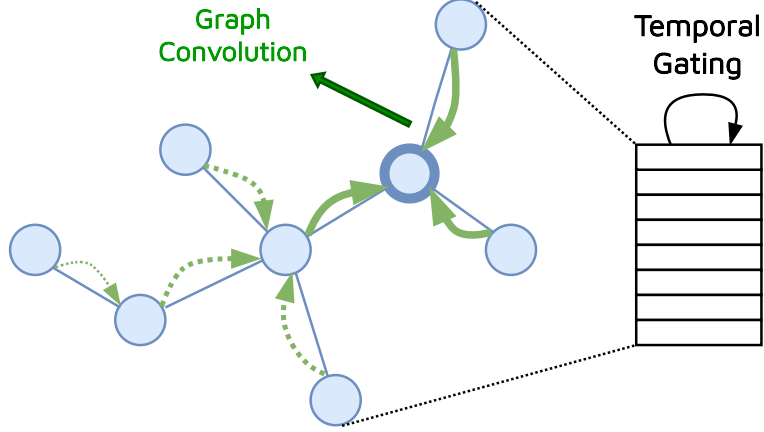


Figure 2.4: Architecture of a Graph Neural Network.

Graph Neural Networks. These takes graphs as an inputs. An architectural representation of GNNs is given in Figure 2.4. Here, the nodes of the graph have feature vectors that are denotes by e_i for node i . The feature vectors are first encoded using a linear layer. The encoded vectors are then used to perform graph convolutions. For each node i , such convolutions are using the neighbouring nodes of i denoted by $n(i)$. As we perform multiple convolutions, we encapsulate the convolution updates using temporal gating (typically using a GRU). Such *graph-to-graph* updates are formally denoted as as:

$$\begin{aligned} r_i^0 &= \text{Tanh}(W e_i + b), \\ x_i^k &= \sum_{j \in n(i)} W^k r_j^{k-1}, \\ r_i^k &= \text{GRU}(r_i^{k-1}, x_i^k). \end{aligned} \quad (2.5)$$

The gating using GRU allows the model to capture long-range dependencies across nodes in the graph. This model is referred to as Gated-Graph Convolution Network [Ruiz et al., 2020]. Another approach for inferring patterns in graph based inputs is Graph Attention Network (GAT) [Veličković et al., 2018].

$$\begin{aligned} a_i &= \text{softmax}(W \cdot e_i + b), \\ x_i &= \sum_{j \in n(i)} a_i^j \cdot W^k \cdot r_j^{k-1}. \end{aligned} \quad (2.6)$$

Compared to GGCN, a GAT allows us to infer over graphs with lower parameter set size, albeit with a poorer prediction accuracy for large-scale graphs. We use graph neural models in Chapters 4, 6-9.

Transformers. Transformers are popular deep learning models that have been used in various natural language and vision processing tasks [Vaswani et al., 2017]. An architectural representation is shown in Figure 2.5. It has two parts: encoder and decoder. The encoder uses this to create an encoded representation of the input window w which is a sliding-window of a time-series data. A multivariate sequence like w is transformed first into a matrix form with modality m . We define scaled-dot product attention [Vaswani et al., 2017] of three matrices Q (query), K (key) and V (value):

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{m}}\right) V. \quad (2.7)$$

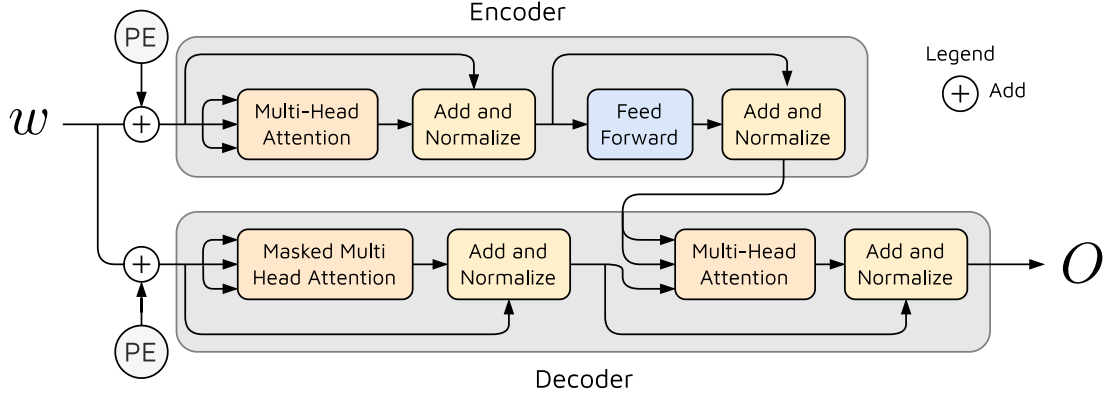


Figure 2.5: Architecture of a Transformer Network.

For input matrices Q , K and V , we apply Multi-Head Self Attention [Vaswani et al., 2017] by first passing it through h (number of heads) feed-forward layers to get Q_i , K_i and V_i for $i \in \{1, \dots, h\}$, and then applying scaled-dot product attention as

$$\text{MultiHeadAtt}(Q, K, V) = \text{Concat}(H_1, \dots, H_h), \quad (2.8)$$

where $H_i = \text{Attention}(Q_i, K_i, V_i)$.

Multi-Head Attention allows the model to jointly attend to information from different representation sub-spaces at different positions. In addition, we use position encoding of the input matrices as defined in [Vaswani et al., 2017]. We then apply position encoding and obtain the input for the encoder, say I_1 . The first performs the following operations

$$\begin{aligned} I_1^1 &= \text{LayerNorm}(I_1 + \text{MultiHeadAtt}(I_1, I_1, I_1)), \\ I_1^2 &= \text{LayerNorm}(I_1^1 + \text{FeedForward}(I_1^1)). \end{aligned} \quad (2.9)$$

Here, $\text{MultiHeadAtt}(I_1, I_1, I_1)$ denotes the multi-head self attention operation for the input matrix I_1 . Also, LayerNorm normalizes the output of the layer. For the decoder, we apply position encoding to the input window w to get I_2 . We modify the self-attention in the encoder to mask the data at subsequent positions. This is done to prevent the decoder from looking at the datapoints for future timestamp. The decoder performs the following operations

$$\begin{aligned} I_2^1 &= \text{Mask}(\text{MultiHeadAtt}(I_2, I_2, I_2)), \\ I_2^2 &= \text{LayerNorm}(I_2 + I_2^1), \\ O &= \text{LayerNorm}(I_2^2 + \text{MultiHeadAtt}(I_1^2, I_1^2, I_1^2)). \end{aligned} \quad (2.10)$$

The final output then becomes O . Unlike recurrent models, this does not require iterative prediction, saving training and inference time. We use Transformer models in Chapter 7.

Generative Adversarial Networks. A class of neural networks is generative models. One of the most popular such model is Generative Adversarial Network (GAN) [Goodfellow et al., 2014]. Such models learn a distribution $p_G(x)$ approximating the real data distribution $p_{\text{data}}(x)$ by using an adversarial minimax formulation. A typical GAN consists of two neural networks G and D as shown in Figure 2.6. In lieu of assigning a density to each

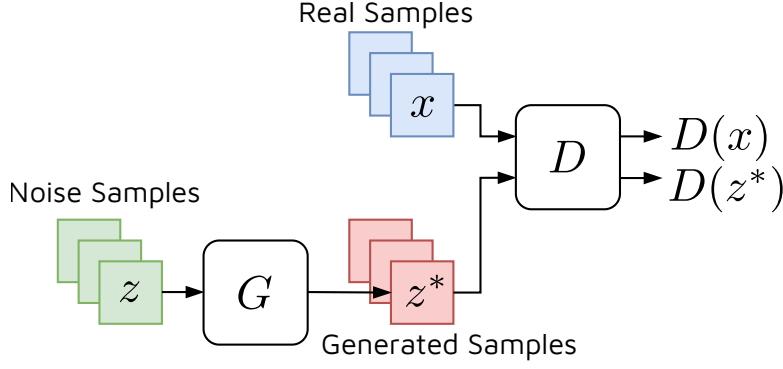


Figure 2.6: Architecture of a Generative Adversarial Network.

data point in the distribution, the generator learns a non-linear function $G(z; \theta)$ from a noise distribution $p(z)$ to the data space. It tries to generate a *realistic* data sample. On the contrary, discriminator aims to distinguish between real samples from the true distribution $p_{\text{data}}(x)$ and fake samples from the generator $p_G(x)$. This is done by optimizing the following minimax objective function:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)} \log D(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D(G(z))). \quad (2.11)$$

We utilize GANs in Chapter 7 and also extend GAN architectures to reduce their memory footprint for memory-limited resource management in Chapter 8.

2.2.2 Loss Functions and Model Training

In cases where we use neural networks as surrogate models, the output of the network $y = f(x; \theta)$ typically should be close to the ground-truth output $\hat{y}$. To train the model, we use the deviation of y and $\hat{y}$ as a signal to update the weights of the model θ . The deviation of the two values is generated using a *loss function*. A common loss function that we use is Mean-Squared-Error that is denoted as

$$\mathcal{L}(y, \hat{y}) = \frac{1}{N} \|y - \hat{y}\|^2, \quad (2.12)$$

where N is the dimensionality of the output vectors. Using the deviation of y from a ground-truth output $\hat{y}$, we can update the parameters of the neural model θ using gradient-based update

$$\theta \leftarrow \theta - \nabla_{\theta}(\hat{y} - f(x; \theta)). \quad (2.13)$$

We iteratively update the parameters till the update steps are negligible. This equation is called *stochastic gradient descent* in literature [Russell and Norvig, 2009]. More advanced versions have been implemented, such as Adam [Kingma and Ba, 2014] and Adahessian [Yao et al., 2021] optimizers. Such methods use techniques such as momentum based parameter updates and higher-order gradients to avoid getting stuck in local minima and converge in few iterations.

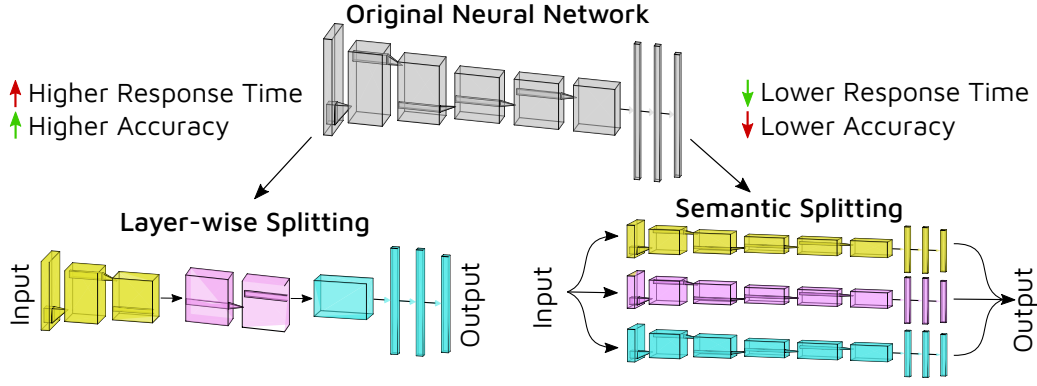


Figure 2.7: Overview of layer and semantic splitting strategies

2.2.3 Neural Network Splitting

Now that we have introduced neural networks, we now discuss some ways in which large-scale neural networks based workloads may be deployed in memory-constrained fog devices. The key idea that we use in subsequent chapters and heavily in Chapter 5 is neural network splitting. Complete neural models with the input batch can seldom be accommodated in the random-access-memory (RAM) of edge devices. Thus, ideas like model compression or splitting are required to make inference over large-scale neural networks plausible in such environments. Frameworks that aim at achieving this must maintain a careful balance between accuracy requirements and response times for different user tasks. For such a framework, real-time analysis of the incoming tasks is required for quick decision making of task placement. This requires robust algorithms to seamlessly integrate different paradigms and meet the user's service level agreements.

In this thesis, we leverage the only two available splitting schemes for neural networks: layer and semantic splitting [Yu et al., 2021, Kim et al., 2017]. An overview of these two strategies is shown in Figure 2.7. Semantic splitting divides the network weights into a hierarchy of multiple groups that use a different set of features (different colored models in Figure 2.7). Here, the neural network is split based on the data semantics, producing a tree structured model that has no connection among branches of the tree, allowing parallelization of input analysis [Kim et al., 2017]. Due to limited information sharing among the neural network fragments, the semantic splitting scheme gives lower accuracy in general. Semantic splitting requires a separate training procedure where publicly available pre-trained models cannot be used. This is because a pre-trained standard neural network can be split layer wise without affecting output semantics. For semantic splitting we would need to first split the neural network based on data semantics and re-train the model. However, semantic splitting provides parallel task processing and hence lower inference times, more suitable for mission-critical tasks like healthcare and surveillance. Layer-wise splitting divides the network into groups of layers for sequential processing of the task input, shown as different colored models in Figure 2.7.

2.3 Literature Review of Resource Management Methods

Recent research in AI has shown some promise in the direction of improving QoS of Fog systems, thanks to their higher inference speeds and accuracy compared to classical techniques [Liang et al., 2020]. AI research for Fog systems has spanned diverse categories including (1) classical AI that covers informed and uninformed search methods, (2) machine learning that encompasses unsupervised, supervised and semi-supervised methods,

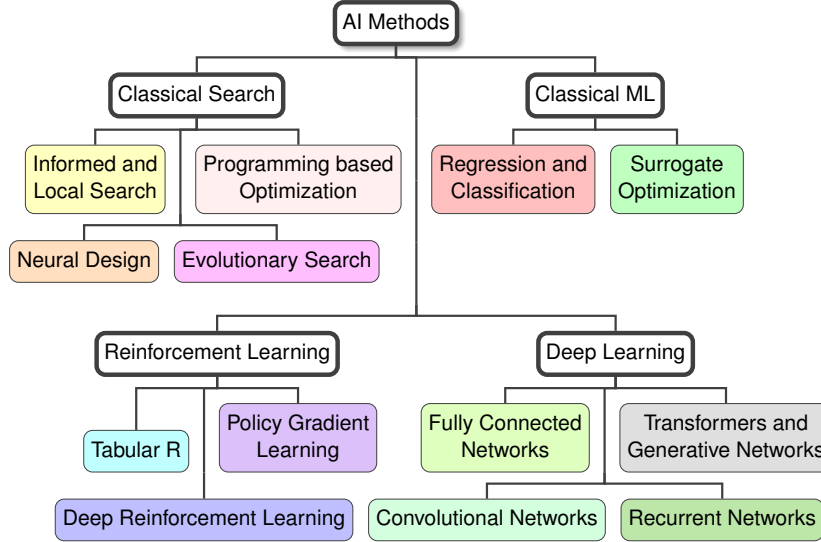


Figure 2.8: A brief taxonomy of AI methods for Fog systems.

(3) reinforcement learning (RL) that includes tabular and deep reinforcement methods, and (4) deep learning that uses deep neural networks as function approximators to model complex relationships across data in Fog systems [Russell and Norvig, 2009, Goodfellow et al., 2016]. A brief taxonomy from Russell and Norvig [2009] is presented in Figure 2.8. Although, considering the assumption that the environment state is fully known, resource management decision making is similar to RL. However, unlike RL, we do not consider the Markovian assumption to hold for Fog environments. Further, unlike RL, we also assume the presence of offline data (supervised or unsupervised) available for model training. We now discuss the state-of-the-art approaches categorized as per the three scopes of deployment, scheduling and dependability.

2.3.1 AI Augmented Deployment

A summary of all AI augmented deployment methods is presented in Table 2.1. Here, a *benchmark* corresponds to the workloads used to train and test the presented methods. *Infra.* column represents whether the methods utilize Edge (E) or Cloud (C) or both (E+C). Asterisk with framework/simulator means that the respective chapters utilize a modified version of the base platforms.

2.3.1.1 AI Augmented DNN Deployment

Running training or inference procedures for AI/ML models is computationally expensive. Given that Edge devices tend to have limited compute resources that are usually shared across multiple running applications, it is essential to develop resource-efficient training and inference mechanisms to ensure short training times and resource load. Several solutions have been proposed in the past to address this.

AutoML. For instance, several AutoML (automated machine learning) techniques run search in the space of neural network models, *i.e.*, NAS, to find out the optimal DNN model to execute a task in a given system [Xia et al., 2019, Zhou et al., 2019, Li et al., 2019, Zhao et al., 2021]. These methods can be run to find the optimal DNN architecture for a given set of constraints such as training or inference times, memory footprint, computational requirements, etc.

Model Pruning. Another direction is to take existing AI models and prune their parameters to reduce overall

Table 2.1: Summary of state-of-the-art methods for AI augmented deployment. Color scheme as per Figure 2.8.

Decision Type	Category	Ref.	Method	Infra.	Benchmark	Framework / Simulator	Merits	Limitations
DNN Deployment	AutoML	[406, 203, 437]	Policy Gradient Learning	E	AIoTBench	F: Custom	Fast Execution	Generalizability
	Model Pruning	[170, 224]	Search	E+C	AIoTBench	F: Custom	Cost Efficient	Accuracy
		[319, 422, 150]	Neural Design	E+C	AIoTBench	-	Generalizable	Overhead
		[439]	Search	E+C	AIoTBench	F: Custom	Cost Efficient	Overhead
	Gradient Pruning	[313, 231]	Search	E	AIoTBench	F: Custom	Low Overhead	Accuracy
	Low	[292, 126, 353]	Neural Design	E+C	Shakespeare	S: Custom	Low Overhead	Accuracy
		[71, 157]	Neural Design	E+C	-	-	Generalizable	Scalability
	Precision	[190, 154]	Search	E+C	AIoTBench*	F: Custom	Memory Efficient	Generalizability
	Layer Splitting	[172, 430, 142]	Search	E+C	AIoTBench*	S: Bighouse	Energy Efficient	Generalizability
	& Early Exits	[50]	Linear Programming	E	-	-	Fast Execution	Scalability
		[247, 350, 108]	Neural Design	E+C	AIoTBench*	-	Low Overhead	Accuracy
		[423]	Policy Gradient Learning	C	AIoTBench*	F: AWS IoT	Memory Efficient	Generalizability
	Splitting	[173]	Search	C	AIoTBench*	F: AWS IoT	Generalizable	Overhead
		[62, 146, 180]	Neural Design	E+C	AIoTBench*	F: Custom	High Accuracy	Generalizability
		[355]	DQN + Multi Armed Bandits	E+C	AIoTBench*	F: COSCO	High QoS	Overhead
Resource Provisioning	Demand Prediction	[153]	Linear Regression	E+C	-	S: Custom	Fast Execution	Accuracy
		[444]	Support Vector Regression	C	Google Cluster	S: Custom	Memory Efficient	Scalability
		[58, 229, 230]	Gaussian Process Regression	E+C	Azure2017/19	S: Custom	High Accuracy	Scalability
		[348]	Modular Regression	E+C	-	-	Scalable	Accuracy
		[330, 49]	ARIMA	C	Wikimedia	S: Custom	High Accuracy	Scalability
		[411]	Decision Regression Tree	E	Bikeshare	-	Fast Execution	Only Univariate
		[32, 167]	CNN	E+C	Custom	-	High Accuracy	Interpretability
		[270, 420]	LSTM	C	Custom	S: Custom	High Accuracy	Interpretability
		[12]	Ant Colony Optimization	E+C	Bitbrain	S: Cloudsim	High util. ratio	Scalability
		[229, 230]	Bayesian Optimization	E+C	Azure2017/19	S: Custom	High util. ratio	Scalability
	Decision Optimization	[444, 64]	Particle Swarm Optimization	C	Google Cluster	S: Custom	Memory Efficient	Scalability
		[20]	Genetic Algorithm	C	-	S: Cloudsim	High util. ratio	Execution Time
		[402]	Deep Surrogate Optimization	E+C	Yahoo Webscope	S: Custom	Cost Efficient	Generalizability
	Hybrid Provisioning	[338]	DNN + Dynamic Prog.	E+C	-	-	High util. ratio	Scalability
		[199]	FCN + Multi Armed Bandits	C	Custom	F: Kafka	Fast & Scalable	Exposure Bias
		[59]	FCN + Monte Carlo Tree Search	C	-	-	High QoS	Execution Time
		[413, 37, 309]	Deep Q Learning	E+C	Google Cluster	S: Custom	Scalable	Interpretability
		[412, 65, 60]	Policy Gradient Learning	E+C	-	S: iFogSim	Low Energy	Scalability

local training cost, for instance, PruneFL [Jiang et al., 2019]. Model pruning is a commonly used strategy in ML that intelligently cuts away parts of the model architecture without compromising the quality of model inference [Shao and Zhang, 2020, Yu et al., 2020, Huang et al., 2020]. By pruning out parts of the model, the computational resources required to train or run inference on the model are reduced, making it more amenable to Edge and Fog systems. Similar model pruning works are dependent on the DNN design. For instance, SlimGAN [Hou et al., 2021], Gradient Origin Networks [Bond-Taylor and Willcocks, 2021] and similar GAN slimming techniques [Wang et al., 2020]. Similar works exist that perform model pruning for other DNN types, for instance, CNNs [Liu et al., 2017]. Pruning reduces both the memory and compute requirements of models. Similarly methods like BBNNet utilize multiple techniques together, such as model pruning and data compression [Zhou et al., 2021]. BBNNet decides the optimal pruning and compression parameters using local search-based techniques.

Gradient Pruning. In FL systems, all worker nodes need to send gradient updates of their models over the network, which generally translates to up to gigabytes of data depending on the size of the model. To avoid bandwidth contentions, several works discuss solutions to prune the gradient updates improving memory and network efficiency in federated setups [Yang et al., 2020]. Some approaches, such as Sparse Ternary Compression (STC) [Sattler et al., 2019] and Deep Gradient Compression (DGC) [Luo et al., 2021], employ compression mechanisms to reduce the communication bandwidth required for distributed training or inference. Other methods, like FedPq [Reisizadeh et al., 2020], perform periodic aggregation and quantization to reduce communication frequency. A similar approach, CMFL [Luping et al., 2019], intelligently decides which model updates give the maximum boost in performance and only sends the top-performing gradient updates. Another

work, FedBoost [Hamer et al., 2020], uses *ensemble training* to boost model training efficiency and offload only a small part of the ensemble to the Edge devices with predefined intervals to reduce communication overheads. Another work, FEDL [Tran et al., 2019], theoretically demonstrates the relationship between convergence rate and energy consumption of an FL system and formulates the computation and communication models as a non-convex optimization to optimize the distribution of federated networks.

Low-Precision. Energy efficiency is also one of the major concerns when designing efficient ML training algorithms due to FL parties generally being battery-powered devices. Deep learning is inherently very power-consuming due to the large amounts of computation that need to be performed. As such, there has been a relatively large amount of work in energy efficiency by discrete quantization and using low-precision hardware architectures [Coelho et al., 2021, Gong et al., 2019, Jain et al., 2018, Langroudi et al., 2019,]. This not only reduces the computational overheads, but also gives significant gains in terms of memory and energy footprints [Gong et al., 2019, Hou et al., 2021]. The level of precision cannot be changed dynamically at test time as changing precision requires re-training the models. Thus, these decisions either need to be performed at the setup time or multiple models need to be trained of different precision levels, at the cost of higher training time, to provide control to the resource manager to trade-off between accuracy and memory footprint. Other optimization methods, such as DeepX [Lane et al., 2016], focus on the development of deep learning models on mobile devices by runtime control of the memory to reduce the layer-wise operations, such that only the most important operations use larger bytes. Further, it efficiently identifies unit blocks of a DNN architecture and allocates them to local or remote memory depending on access frequency, improving memory footprint. A similar method, FloatPIM [Imani et al., 2019] provides an interface between software and hardware by using Processing in-memory (PIM) to reduce memory usage.

Layer Splitting and Early-Exits. For typical DNNs, it is possible to run inference without performing operations across all layers. Methods, such as Neurosurgeon [Kang et al., 2017], Scission [Lockhart et al., 2020], Couper [Hsu et al., 2019], FedApt [Wu et al., 2022], Musical-Chairs [Hadidi et al., 2018], CONTINUER [Majeed et al., 2022] and DeepSlicing [Zhang et al., 2021], decide the optimal layer partitions of a neural network using grid-search at run time to maximize system QoS. Others, like SplitComp [Callegaro et al., 2020] model the problem of deciding the optimal splitting strategy as a Markov Process and leverage Linear Programming to converge to the optimal splitting strategy. Many of such approaches utilize ideas similar to those in auto-parallelization strategies for model training. These include parallelizing the various model components (such as layers) and datasets. Further, to reach the best tradeoff between model accuracy and processing delay, many early-exit strategies have been proposed when the inference is performed only through a few layers instead of the entire DNN [Li et al., 2019, Pacheco et al., 2021, Wang et al., 2019]. Most work in this category aims at segregating these network splits into different devices based on their computational performance [Matsubara et al., 2019, Teerapittayanon et al., 2017, Goli et al., 2020, Zhang et al., 2021, Kang et al., 2017]. Thus, fast and localized inference using shallow portions of DL models can allow quick inference, possibly at the cost of poorer resulting accuracy. This gives a tradeoff between result fidelity and response time. Several works have been proposed to leverage this tradeoff for multi-objective optimization, especially to reduce the frequency of SLA violations [Mahmud et al., 2018, Yang et al., 2020]. Other recent methods aim at exploiting the resource heterogeneity in the same network layer by splitting and placing DNNs based on user demands and host capabilities [Gunasekaran et al., 2020]. Such methods can split DNNs and choose from different

architectural choices to reach the maximum accuracy while agreeing to the latency constraints. Other works aim at accelerating the model run-times by appropriate scheduling of a variety of DNN models on Edge-clusters [Liang et al., 2020]. Another method, Gillis, uses a hybrid model, wherein it employs either model-compression or layer-splitting based on the application SLA demands [Yu et al., 2021]. The decision is taken using a reinforcement-learning model which continuously adapts in dynamic scenarios. It is a serverless-based model serving system that automatically partitions a large model across multiple serverless functions for faster inference and reduced memory footprint per function. The Gillis method employs two model partitioning algorithms that respectively achieve latency optimal serving and cost-optimal serving with SLA compliance. However, this method cannot jointly optimize both latency and SLA. Moreover, it does not consider the mobility of devices or users and hence is ineffective in efficiently managing large DNNs in mobile Edge environments.

Splitting. Two types of splitting strategies exist: data splitting and semantic model splitting. Data splitting splits the input data batch across multiple instances of the neural networks for parallel inference. Data splitting allows reducing the response time of inference over input data, at the cost of higher network overheads [Kaplunovich and Yesha, 2020]. Semantic model splitting divides the network weights into a hierarchy of multiple groups that use a different set of features. Here, the neural network is split based on the data semantics, producing a tree structured model that has no connection among branches of the tree, allowing parallelization of input analysis [Kim et al., 2017]. Due to limited information sharing among the neural network fragments, the semantic splitting scheme gives lower accuracy in general compared to unsplit networks. Semantic splitting requires a separate training procedure where publicly available pre-trained models cannot be used. This is because a pre-trained standard neural network can be split layer-wise without affecting output semantics. For semantic splitting, we would need to first split the neural network based on data semantics and re-train the model. However, semantic splitting provides parallel task processing and hence lower inference times, more suitable for mission-critical tasks like healthcare and surveillance. Examples include ThriftyEdge [Chen et al., 2018], CLIO [Huang et al., 2020] and SplitNet [Kim et al., 2017, Ushakov et al., 2018].

2.3.1.2 AI Augmented Resource Provisioning

Systematic resource provisioning is central to cost and resource-efficient computation in Fog systems. Bootstrapping resources, such as Cloud VMs or Edge nodes is time-consuming for latency-critical tasks; a key challenge is to predict future workload demands to provision resources to optimize QoS. Resource management is a key aspect of resource provisioning, which instantiates and deallocates resources based on dynamic workload demands. Most prior work aims to automate resource provisioning to optimize various performance measures such as energy consumption, cost, and task response time [Perera et al., 2017, Levy et al., 2020]. However, this problem is challenging due to the non-stationary utilization characteristics of most workloads [Ebadifard and Babamir, 2021], requiring methods to dynamically adapt their provisioning policies. Most dynamic resource provisioning methods decouple the provisioning problem into two stages: demand prediction and decision optimization [Luo et al., 2020]. This is commonly referred to as the *predict+optimize* framework in literature. Thus, we divide prior approaches based on their decision type.

Demand Prediction. Methods that forecast demands at a future state of a Fog system need data corresponding to historical workload demands on the same system. Several methods have been proposed that leverage a forecasting model. For instance, a class of methods utilize regression models such as Linear Regression

(LR) [Hyndman and Athanasopoulos, 2018], Support Vector Regression (SVR) [Zhu et al., 2016], Gaussian Process Regression [Chen and Wang, 2018, Luo et al., 2020, 2021] or modular regression (Prophet) [Taylor and Letham, 2018]. Others utilize auto-regressive models such as AutoARIMA [Singh et al., 2019, Calheiros et al., 2014] or other ML models like Regression Tree (RT) [Xu et al., 2020], time series decomposition (TS-Dec) [Hyndman and Athanasopoulos, 2018] or unobserved component model (UCM) [Durbin and Koopman, 2012] based forecasting. Recent works utilize DNNs to perform forecasting, for instance, using LSTM neural networks [Ouhamme et al., 2021], CNNs [Bega et al., 2019], convolutional wavelet neural networks [Jeddi and Sharifian, 2019] or LSTM based GANs [Yazdani and Sharifian, 2021].

Decision Optimization. Using a demand prediction model, several previous works optimize the provisioning decision to minimize execution costs or maximize the utilization ratio. Conventional methods often use evolutionary search strategies such as Ant Colony Optimization (ACO) [Aliyu et al., 2020], which has been shown to exhibit state-of-the-art QoS scores in recent work [Luo et al., 2021]. Others use Bayesian Optimization (BO) [Luo et al., 2020, 2021], Particle Swarm Optimization (PSO) [Zhu et al., 2016, Chen et al., 2020] or Genetic Algorithms [Asghari et al., 2021].

Hybrid Provisioning. Other methods, such as Decision-NN, combine the prediction and optimization steps by modifying the loss function to train neural networks in conjunction with the optimization algorithm [Wilder et al., 2019]. This method uses a neural network as a surrogate model to directly predict optimization objectives and uses the concept of neural network inversion, wherein the method evaluates gradients of the objective function with respect to inputs and runs optimization in the input space. However, continuous relaxation of the discrete optimization problem used in this work has been shown to adversely impact performance [Luo et al., 2020]. A similar method, Semi-Direct, utilizes dynamic programming to find the optimal provisioning decision [Stuckey et al., 2020], but offers limited scalability with workload size. Similarly, Narya [Levy et al., 2020] is built for mitigating VM interruptions in Cloud machines, but can be straightforwardly extended to resource provisioning. It uses a neural network as a surrogate model with a multi-armed bandit model to decide provisioning actions. However, it faces the problem of exposure bias, *i.e.*, the neural model is biased to the trends in the training data and is unable to forecast in unseen cases.

Reactive Provisioning Recently, RL based methods have been proposed for reactive provisioning. For instance, Intelligent Resource Allocation Framework (iRAF) [Chen et al., 2019] solves the complex resource allocation problem for the collaborative mobile Edge computing (CoMEC) network using Deep Reinforcement Learning (DRL) with a multi-task objective formulation. It makes resource allocation decisions based on network states and other task characteristics such as the computing capability of devices, network quality, resource utilization, and latency requirements. iRAF automatically takes into account the network environment and makes resource allocation decisions to maximize the performance over latency and power consumption. It uses self-play training where the agent becomes its own teacher and learns over time in a self-supervised learning manner. Specifically, it uses a fully connected network (FCN) with Monte Carlo Tree Search (MCTS) to optimize the provisioning decision. Some other works, such as DDRM [Chen et al., 2020], focus on the integration of IoT and industrial manufacturing systems (IIoT). Another similar work [Baek and Kaddoum, 2020] focuses on optimizing the Fog nodes by selecting the suitable nodes and proper resource management while guaranteeing the QoS requirements of the users. It designs a joint task offloading and resource allocation control for heterogeneous service tasks in multi-fog nodes systems. It applies a deep recurrent Q-network (DRQN) approach

Table 2.2: Summary of state-of-the-art methods for AI augmented scheduling. Color scheme as per Figure 2.1.

Decision Type	Category	Ref.	Method	Infra.	Benchmark	Framework / Simulator	Merits	Limitations
Bag-of-Tasks Scheduling	Maxweight Scheduling	[187]	Maxweight	E+C	-	S: Custom	Fast Execution	Resp. Time
		[222]	Maxweight	E+C	Tutoring	S: Custom	Fast Execution	Resp. Time
	Surrogate Modeling	[169]	DNN + Linear Programming	E	-	S: Custom	Energy Efficient	Accuracy
		[127]	DNN + GA	E+C	-	-	Generalizable	Overhead
		[359]	GMM + GA	E+C	Bitbrain	S: iFogSim	Fast Execution	Overhead
		[364]	DNN + Gradient Opt.	E+C	DeFog	F: COSCO	High QoS	Interpretability
		[363]	GNN + Gradient Opt.	C	DeFog	F: COSCO	High QoS	Exposure Bias
		[160, 46]	Gaussian Process Regression	C	Google Cluster	S: Custom	Fast Execution	Overhead
	Stochastic Modeling	[273, 166]	Robust Search	C	Custom	-	Energy Efficient	Overhead
		[11]	DBN + Search	E+C	-	S: Custom	Fast Execution	Accuracy
		[366]	NPN + Gradient Opt.	E+C	DeFog	F: COSCO	High QoS	Interpretability
		[346, 205]	Deep Q Learning	E+C	Crawdad	S: Cloudsim	Generalizable	Scalability
	Reinforcement Learning	[382]	Minimax Q Learning	C	Traffic	S: Clousim	Generalizable	Scalability
		[325]	Policy Gradient Learning	E	-	S: Custom	High QoS	Adaptability
		[360, 66]	Policy Gradient Learning	E	Bitbrain	S: iFogSim	High QoS	Adaptability
	Co-Simulation	[364, 366]	FCN + Co-Simulation	E+C	DeFog	F: COSCO	Generalizable	Overhead
Workflow Scheduling	Meta-Heuristic Methods	[390]	Particle Swarm Optimization	C	WfCommons	F: AWS IoT	Generalizable	Low QoS
		[148]	Ant Colony Optimization	E+C	WfCommons	F: AWS IoT	Generalizable	Low QoS
		[102]	Ant Mating Optimization	E+C	WfCommons	S: Custom	Energy Efficient	Overhead
	Surrogate Optimization	[156]	DNN + GA	C	WfCommons	-	Generalizable	Overhead
		[282]	DNN + GA	C	WfCommons	S: Cloudsim	Cost Efficient	Overhead
		[367]	DNN + Gradient Opt.	E+C	WfCommons	F: COSCO	High QoS	Interpretability
	Game Theory	[395]	Attack-Defense Model	C	WfCommons	F: Openstack	Cost Efficient	Overhead
	Reinforcement Learning	[393]	Deep Q Learning	C	WfCommons	F: AWS IoT	Generalizable	Scalability
		[174]	Deep Q Learning	C	WfCommons	S: Clousim*	Generalizable	Scalability
		[103]	Policy Gradient Learning	E+C	-	-	High QoS	Adaptability
		[145]	Policy Gradient Learning	C	WfCommons	F: Custom	High QoS	Adaptability
	Other	[93]	LSTM	E+C	Custom	-	Generalizable	Scalability
		[392]	Support Vector Regression	E	-	S: Custom	Fast Execution	Scalability
		[13]	Gradient Descent Search	E+C	Custom	S: CloudSim	High QoS	Overhead

to approximate the optimal value functions and applies an adjusted exploration-exploitation method to make the optimization process more efficient. Similarly, ReCARL [Xu et al., 2020] focuses on Cloud Radio Access Networks (CRANs). It proposes a resource allocation scheme in CRANs to improve the objective of power consumption and SLA violations of wireless users over a long time period. It uses two DRL agents: ReCARL-Basic (requiring limited training) and ReCARL-Hybrid (requiring deep learning training). It has been evaluated via extensive simulation to demonstrate that ReCARL achieves significant power savings in highly dynamic settings while meeting user SLA demands. Similarly, Deep Elastic Resource Provisioning (DERP) [Bitsakos et al., 2018] uses Deep-Q learning to optimize provisioning decisions with utilization ratio as a reward for the DRL agent. Unlike Q-learning based agents that utilize a neural network to predict the expected reward for each action [Sami et al., 2021], recent methods also use neural networks to approximate the optimal policy. Such approaches are called policy gradient methods and include works by Xu et al. [2020] and Chen et al. [2021].

2.3.2 AI Augmented Scheduling

A summary of recent AI-augmented scheduling methods is presented in Table 2.2.

2.3.2.1 AI Augmented Scheduling of Bag-Of-Tasks

QoS-aware placement of IoT applications requires reaching a tradeoff among multiple QoS parameters such as response time, cost and energy. In the *bag-of-task* workload model, each task can be independently scheduled. **MaxWeight-Scheduling.** Over the years, many scheduling approaches have turned to utilize MaxWeight based techniques due to its theoretical guarantees and the ability to reduce the frequency of resource contention [Liu et al., 2020, Krishnasamy et al., 2018]. For instance, the pessimistic-optimistic online dispatch approach, POND, is a variant of the MaxWeight approach [Liu et al., 2020]. POND formulates the scheduling

problem as a constrained optimization objective with unknown dispatch, arrival and reward distributions, such that each Fog node has a virtual queue to track violation counts. It uses an Upper-Confidence Bound (UCB) based exploration strategy [Auer et al., 2002] with the final decisions being made with the MaxWeight weights as the expected reward value of each scheduling decision. However, prior work has demonstrated that MaxWeight policies suffer from instability in dynamic workloads, high delays and inefficiency in modeling large-scale Fog networks [Bae et al., 2019, van de Ven et al., 2009, 2013]. MaxWeight schedulers are also known to have high wait times due to their inability to adapt to volatile workload settings [Liu et al., 2020].

Surrogate Modeling. Most classical research in this area employs meta-heuristic algorithms with a DNN or regression model as a surrogate that approximates QoS of a given system state. This is due to their generic formulation and ease of implementation. For instance, prior works have shown that evolutionary-based methods, and generally gradient-free approaches, perform well in dynamic scenarios [Wang et al., 2019, Han et al., 2018, Wang et al., 2020]. Some works use a combination of a DNN surrogate, and classical optimization techniques such as mixed-integer linear programming (MILP) [Jiang et al., 2019]. Evolutionary approaches like genetic algorithms (GA) lie in the domain of gradient-free optimization methods. The GA method schedules workloads using a neural model to approximate the objective value and a genetic algorithm to reach the optimal decision [Han et al., 2018]. Such methods use either analytical models [Wang et al., 2019], Gaussian Mixture Model (GMM) or polynomial approximators [Zhu et al., 2017] or neural networks [Han et al., 2018] to predict system QoS for a given scheduling decision and input Fog state. Typically, such approaches run a search scheme with non-local jumps, due to cross-over and mutation-like operations, to converge towards an optimum. However, gradient-free methods are known to take much longer to converge [Bogolubsky et al., 2016] and are not as scalable [Rios and Sahinidis, 2013] as gradient-based methods. Moreover, non-local jumps can significantly change the scheduling decision, leading to a high number of preemptive task migrations. This entails checkpointing the running task, migrating it to another Fog node and resuming its execution on the new node [Engelmann et al., 2009]. This can give rise to high migration overheads, subsequently increasing the average task response times and SLA violation rates. Furthermore, prior work also establishes that neural approximators can precisely model the gradients of the objective function with respect to input using back-propagation [Nguyen-Thien and Tran-Cong, 1999]. Now, although such works use these gradients with respect to input for solving differential equations, they can also be applied for gradient-based optimizations. However, even with the advantages of scalability and quick convergence to optima, few prior works use gradient-based methods as neural approximators are not consistent with the convexity/concavity requirements of such methods [Nandi et al., 2001]. This problem is alleviated by momentum and annealing in schedulers such as GOBI and GOSH, which we discuss in Chapters 3 and 6. Such methods take the scheduling decision and state of the Fog system as resource utilization characteristics of workloads and Fog nodes and output a QoS estimate. Using backpropagation to input, *i.e.*, fixing the neural network parameters and updating the scheduling decision based on the gradient of DNN output, these methods find the optimal scheduling decisions. Further, continuous approximation of a discrete optimization problem is known to be sub-optimal in some cases [Miranda-Varela and Mezura-Montes, 2018].

Stochastic Modeling: Another type of models that approximate system QoS is stochastic surrogate models. These include Heteroscedastic Gaussian Processes to approximate the distribution of the QoS metrics instead of giving only a static output for a given input state [Jamshidi and Casale, 2016, Bui et al., 2017, Panda et al., 2015]. Similarly, prior works also predict the mean and variance estimates of system QoS based on

historical data to perform robust and safe decision optimization [Panda et al., 2015, Jawad et al., 2018] or use error-based exploration [Jamshidi and Casale, 2016]. Other methods use Deep Belief Networks (DBN) for response-time predictions, which are used to make prompt offloading decisions under mobility and fluctuating resource demands [Alelaiwi, 2019]. Typically, due to the poor modeling accuracy of Gaussian Processes, they cannot perform well in complex environments like heterogeneous Fog environments. Hence, sophisticated models are required to additionally model the stochasticity in the QoS metrics [Jawad et al., 2018, Wu et al., 2020].

Reinforcement Learning models: Recently, reinforcement learning based methods have shown themselves to be robust and versatile to diverse workload characteristics and complex Fog setups [Tang et al., 2018, Basu et al., 2019, Gazori et al., 2019]. Such methods use a Markovian assumption of state which is the scheduling decision at each interval. Based on new observations of reward signals, they explore or exploit their knowledge of the state-space to converge to an optimal decision. Recent methods, such as DQLCM [Tang et al., 2018] and DeepRM [Li et al., 2019], model the container migration problem as a multi-dimensional Markov Decision Process (MDP) and use a deep-reinforcement learning strategy, namely deep Q-Learning to schedule workloads in a heterogeneous Fog environment. Another similar method, SDAEM-MMQ [Wang et al., 2021] uses a stacked denoising autoencoder with minimax Q learning for accurate Q estimates and robust optimization. Policy gradient methods, such as [Sheng et al., 2021], train a DNN to directly predict the optimal scheduling decision instead of Q values. A recent method, Asynchronous Advantage Actor-Critic (A3C), is a policy gradient method that schedules workloads using an actor-critic pair of DNN agents [Tuli et al., 2020]. This approach uses Residual Recurrent Neural Networks (R2N2) to predict the expected reward for each action *i.e.*, scheduling decision and tries to optimize the cumulative reward signal. Another similar method, Multi-Agent Deep Deterministic Policy Gradient (MADDPG) [Cheng et al., 2021] formulates the decision optimization problem as a stochastic game among multiple RL agents to reach to an optimal schedule. Recently, methods such as George [Li et al., 2021] have been proposed that train a policy network through a simulator. However, such methods are still slow to adapt to real-world application scenarios (see Chapter 3). This leads to higher wait times and subsequently high response times and SLA violations, leading to poor scalability with workload or the number of nodes.

Coupled Optimization. Finally, coupled or symbiotic simulation and model-based control have long been used in the modeling and optimization of distributed systems [Onggo et al., 2021, Bosmans et al., 2019, Onggo et al., 2018]. Many prior works have used hybrid simulation models to optimize decision-making in dynamic systems [Mustafee et al., 2015, Onggo et al., 2018]. To achieve this, they monitor, analyze, plan and execute decisions using previous knowledge-base corpora (MAPE-k) [Gill et al., 2019]. However, such works use this to facilitate search methods and not generate additional data to aid the decision-making of an AI model. A similar work [Onggo et al., 2018], presents the notion of *symbiotic simulation* that aims to feed in the resource characteristics related data to a co-simulated model for optimizing resource management related decisions using ML techniques. Another similar work, EDSS [Onggo et al., 2021], uses a co-simulator to estimate the effects of various resource scheduling decisions from an ML model and choose the one with highest QoS. Running a co-simulator gives another estimate of system QoS, solving two problems at once: the problem of exposure bias to training data as well as the data saturation problem. The former arises due to the surrogate models being trained on a set of pre-collected execution traces, wherein the system characteristics might be different from those at test time. The latter arises due to the limited diversity in training data such that even with more datapoints, the performance of the DNN does not improve.

2.3.2.2 AI Augmented Scheduling of Workflows

Workflow like applications typically have precedence constraints of the form of a DAG that must be adhered to when scheduling such applications. These workloads could be of the form of a layer or semantic split neural models (see Section 2.3.1.1) or other scientific workflow applications [Gill et al., 2019].

Meta-Heuristic methods. This class of methods leverages high-level problem independent algorithms to find the optimal scheduling decision for the workflows. Most state-of-the-art approaches belong to this category. Among these, many use variants of the PSO optimization technique [Wang et al., 2020]. One such technique is the immune-based particle swarm optimization (IMPSON) method [Wang et al., 2020]. It uses candidate affinity to prevent poor candidates from being discarded in subsequent iterations, allowing it to surpass other PSO-based methods in terms of execution costs and average response time. Other techniques, categorized commonly as list scheduling, use metrics like earliest finish time, critical path, and dynamic resource utilization levels [Adhikari et al., 2019]. However, list scheduling performs poorly in settings with non-preemptable jobs and heterogeneous requirements or machines [Adhikari et al., 2019]. Others include ACO, such as [Huang et al., 2020]. Such a technique starts with several random or heuristically initialized candidate solutions. Each candidate is iteratively optimized, moving it slightly in the state-space where the optimization objective tends to increase. Such methods aim to reach a balance between makespan-service spread, makespan-energy and makespan-cost, respectively. Further, novel bio-inspired meta-heuristic algorithms are also introduced to solve Fog application scheduling simultaneously considering multiple objectives. For instance, the Ant Mating Optimization (AMO) [Ghanavati et al., 2020], aims to minimize the total system makespan and energy consumption for Fog task scheduling.

Surrogate Optimization. Other recent methods use genetic algorithms to optimize the scheduling decision, again using a DNN as a surrogate model [Ismayilov and Topcuoglu, 2020, Pham and Fahringer, 2020]. Again, due to non-local jumps in the search space, such methods typically lead to better QoS estimates, at the cost of higher task migration overheads. Recent techniques, such as ESVR [Pham and Fahringer, 2020], initialize its candidate population using the Heterogeneous Earliest Finish Time (HEFT) heuristic and optimize using the crossover-mutation scheme. To account for volatility in the system, ESVR continuously fine-tunes the neural network surrogate using the latest workload traces and host characteristics [Pham and Fahringer, 2020]. A similar technique is DNSGA [Ismayilov and Topcuoglu, 2020] that uses a multi-objective optimization method that uses a Pareto Optimal Front (POF) aware approach that prevents the set of candidates from converging to the same optima [Ismayilov and Topcuoglu, 2020]. Prior work shows that these two methods out-perform previously proposed genetic algorithms-based techniques [Ismayilov and Topcuoglu, 2020, Pham and Fahringer, 2020]. However, in the case of long-running workflows, having a short-term QoS estimate is detrimental to the system performance as it leads to myopic optimization. To tackle this, recent methods, such as Monte-Carlo Deep Surrogate (MCDS) [Tuli et al., 2022], trains a DNN to generate long-term QoS estimates by running multiple Monte-Carlo simulations on a co-simulator. This not only helps in long-term optimization, but also facilitates stable learning.

Game-Theoretic Modeling. Another recently proposed workflow scheduling model, namely Closure, uses an attack-defense game-theoretic formulation [Wang et al., 2020]. Unlike other schemes that assume mostly homogeneous resources, Closure has been shown to efficiently manage heterogeneous devices by calculating the Nash Equilibrium of the attack-defense game model. This is crucial in Edge-cloud environments where there are contrasting resource capacities of Edge and Cloud.

Table 2.3: Summary of state-of-the-art methods for AI augmented dependability. Color scheme as per Figure 2.1.

Decision Type	Category	Ref.	Method	Infra.	Benchmark	Framework / Simulator	Merits	Limitations
Fault Detection and Prediction	Unsupervised Reconstruction Models	[151, 339, 275]	LSTM	E+C	SMD	S: Custom	Generalizable	Overhead
		[426, 433]	ConvLSTM	E+C	Custom	S: Custom	High Accuracy	Scalability
		[446]	DNN + GMM	E+C	Custom	S: Custom	High Accuracy	Overhead
		[436, 80]	Graph Neural Network	E+C	SWaT	S: Custom	High Accuracy	Overhead
	Generative Models	[202, 94]	Generative Adversarial Nets	E+C	SWaT	-	Fast and Scalable	Exposure Bias
		[21, 100]	VAE	C	-	F: Custom	Memory Efficient	Scalability
		[134, 107]	GNN + LSTM + VAE	C	HDFS	F: Custom	High Accuracy	Execution Time
		[68]	LSTM + VAE	C	-	-	Generalizable	Overhead
		[147]	Transformers	E+C	HDFS	F: Openstack	Fast Execution	Generalizability
		[365]	Neural Design	E	SMD	F: COSCO	Memory Efficient	Overhead
	Clustering Methods	[403]	Few Shot Learning	C	-	F: Openstack	Fast Execution	Accuracy
		[200]	Fuzzy Clustering	E+C	-	-	Fast Execution	Accuracy
		[144]	VAE + Fuzzy Clustering	E+C	Custom	S: Custom	Fast Execution	Accuracy
		[344]	Q Learning	E+C	MHealth	-	Generalizable	Scalability
Fault Remediation	Fault-Tolerant Scheduling	[343]	Deep Surrogate Optimization	E+C	MHealth	S: iFogSim	Cost Efficient	Interpretability
		[220]	Particle Swarm Optimization	C	-	S: Cloudsim*	Memory Efficient	Scalability
		[312]	Crow Search	C	Custom	S: Cloudsim	Energy Efficient	Scalability
		[388]	Deep Q Learning	E+C	-	S: Custom	High Throughput	Scalability
	Load Balancing	[368]	Generative Adversarial Nets	E	DeFog	F: COSCO	High QoS	Data Hungry
		[161]	Particle Swarm Optimization	E+C	-	S: iFogSim	Fast Execution	Scalability
		[175]	Grey Wolf Optimization	E+C	WfCommons	S: iFogSim*	Cost Efficient	Scalability
		[242]	DNN + Search	C	Google Cluster	S: Cloudsim	High Util. Ratio	Interpretability
	Scaling	[15]	Fuzzy Logic + Search	C	-	F: Openstack	Low Overhead	Low QoS
		[92]	RNN	E+C	Custom	S: iFogSim	Low Overhead	Scalability
		[4]	Decision Tree Regression	E+C	Custom	F: Custom	Fast Execution	Scalability
		[91]	NAR Network	E+C	T-Drive	S: iFogSim*	Cost Efficient	Scalability
		[201]	DNN + ARIMA	E+C	Custom	-	Cost Efficient	Scalability
		[256]	Fuzzy Logic + Search	E+C	-	S: Cloudsim*	Fast and Scalable	Scalability
	Straggler Analysis	[18]	Dynamic Bayesian Network	E	SETI	-	Fast Execution	Interpretability
		[16]	Bayesian Neural Network	E	Custom	-	Fast Execution	Interpretability
		[17]	Dynamic Bayesian Network	E	SETI	-	Fast Execution	Interpretability
		[362]	VAE + LSTM	C	PlanetLab	S: Cloudsim	High QoS	Overhead

Reinforcement Learning. Some methods constrain the action space of the MDP formulation to exclude scheduling decisions that violate the precedence constraints set by the incoming workloads. These include removing infeasible actions from the action set at each state of the MDP [Wang et al., 2019] for Deep Q Networks (DQN) or masking the policy likelihood scores in policy gradient methods [Hu et al., 2019, Ghosal et al., 2020]. For instance, DQ-HEFT [Kaur et al., 2020] superimposes the task order over the reward function to ensure that the Q-learning model converges to an optimal scheduling decision.

Other. Many prior works utilize other augmentation strategies in tandem with AI. For instance, some works optimize the Fog network. Examples include [Jalali et al., 2019, 2017], which introduce cognitive Edge gateways that use machine learning (regression and ensemble models) to automatically learn the best allocation for each task based on the Fog environment status and performance requirements of the tasks. Similarly, other methods [Wang et al., 2020] propose an intelligent task offloading algorithm to synergistically run them on Edge and Cloud platforms. Their dynamic switching algorithm groups applications using a support vector machine-based approach to improve the performance in terms of delay and energy consumption. Other methods such as gradient descent search [Alsurdeh et al., 2021] has been adopted for hybrid workflow scheduling in Edge and cloud computing to optimize execution time and monetary cost.

2.3.3 AI Augmented Dependability

In this work, we focus on the aspect of maintaining Fog systems using resource management techniques, particularly concerned with fault tolerance, resilience and remediation. Resilience is crucial when utilizing AI for

resource management, as corrupted computation from failed nodes can lead to ML systems having erroneous behavior. Such errors can be fatal in some scenarios, such as autonomous driving and medical predictions. We measure system resilience with three metrics.

1. *Resource Contention*: Stressful workloads tend to overwhelm the resource capacities of the Edge or Cloud nodes, leading to competition among workloads for resources. This competition can cause failures due to inefficient resource scheduling, resulting in outages. The most common way of mitigating this is by proactive resource provisioning that ensures sufficient resources are available for incoming workloads before they arrive. However, it is crucial to eschew the over-provisioning of resources to avoid system under-utilization or resource wastage in Fog systems.
2. *Service Availability*: It is possible that the node performing a crucial computation fails due to hardware or software faults. This disrupts the service provided to the user and is a critical metric to measure system reliability. It is usually addressed by having multiple Fog nodes involved in the processing of the same application so that one can take over if the other fails. This resilience concept is known as hot-standby, where the backup resources are called fallback nodes [Zhao et al., 2020]. However, this metric trades off with energy and cost as application replication leads to redundant computations, leading to inefficiency.
3. *Security and Privacy*: Fog systems must also ensure data resilience to ensure that data is not compromised by malicious attacks [Zhang et al., 2019]. This includes data integrity, *i.e.*, resilience to data corruption, and data confidentiality, *i.e.*, sensitive data remains hidden from malicious entities. To avoid data corruption and stealing, technologies such as encryption [Bonawitz et al., 2017], differential privacy [Abadi et al., 2016], detection [Preuveneers et al., 2018] are used.

A summary of resilience methods for Fog systems is presented in Table 2.3.

2.3.3.1 AI Augmented Fault-Detection and Prediction

Several machine learning algorithms have been proposed for fault detection and prediction in Edge Cloud environments. They have proposed a framework that includes time-series data collection and data pre-processing components for the training of DNNs.

Unsupervised Reconstruction Models. Majority of prior work proposes reconstruction-based methods that predominantly aim to encapsulate the temporal trends and predict the time-series system data in an unsupervised fashion, then use the deviation of the prediction with the ground-truth data as anomaly scores. In such methods, the time-series system data may correspond to utilization characteristics of the running workloads in a Fog system. One such method, LSTM-NDT [Hundman et al., 2018], relies on an LSTM to forecast data for the next timestamp. This work also proposes a non-parametric dynamic error thresholding (NDT) strategy to set a threshold for anomaly labeling using moving averages of the error sequence. A similar work, Omnianomaly [Su et al., 2019], uses a stochastic recurrent neural network (similar to an LSTM-Variational Autoencoder [Park et al., 2018]) and a planar normalizing flow to generate reconstruction probabilities. It also proposes an adjusted Peak-Over-Threshold (POT) method for automated anomaly threshold selection that outperforms the previously used NDT approach. This work led to a significant performance leap compared to the prior art, but at the expense of high training times. The Multi-Scale Convolutional Recursive Encoder

Decoder (MSCRED) [Zhang et al., 2019] converts an input sequence window into a normalized two-dimensional image and then passes it through a ConvLSTM layer. This method is able to capture more complex inter-modal correlations and temporal information, however is unable to generalize to settings with insufficient training data. The CAE-M [Zhang et al., 2021] uses a convolutional autoencoding memory network, similar to MSCRED. It passes the time-series through a CNN with the output being processed by bidirectional LSTMs to capture long-term temporal trends. Such recurrent neural network-based models have been shown to have high computation costs and low scalability for high dimensional datasets [Audibert et al., 2020]. The DAGMM [Zong et al., 2018] method uses a deep autoencoding Gaussian mixture model for dimension reduction in the feature space and recurrent networks for temporal modeling. This work predicts an output using a mixture of Gaussians, where the parameters of each Gaussian are given by a deep neural model. However, it still is slow and unable to explicitly utilize inter-modal correlations [Deng and Hooi, 2021]. The Graph Deviation Network (GDN) approach learns a graph of relationships between data modes and uses attention-based forecasting and deviation scoring to output anomaly scores. MTAD-GAT [Zhao et al., 2020] uses a graph-attention network to model both feature and temporal correlations and pass it through a lightweight Gated-Recurrent-Unit (GRU) network that aids detection without severe overheads. Traditionally, attention operations perform input compression using convex combinations where the weights are determined using neural networks.

Generative Models. Prior methods have used simulators to generate situations when the model is likely to fail. For instance, companies like Netflix have used chaos engineering to randomly induce artificial failures to build confidence in the system’s capability to withstand turbulent conditions in production [Basiri et al., 2016]. However, when randomly generating such situations, finding outliers may be time-consuming and require several compute hours, rendering them ineffective in volatile settings considered in this thesis. More recent works such as USAD [Audibert et al., 2020], MAD-GAN [Li et al., 2019] and openGauss [Li et al., 2021] do not use resource-hungry recurrent models, but only attention-based network architectures to improve training speeds. The USAD method uses an autoencoder with two decoders with an adversarial game-style training framework. This is one of the first works that focus on low overheads by using a simple autoencoder and can achieve a several-fold reduction in training times compared to the prior art. The MAD-GAN [Li et al., 2019] uses an LSTM based GAN model to model the time-series distribution using generators. This work uses not only the prediction error, but also the discriminator loss in the anomaly scores. The openGauss approach uses a tree-based LSTM that has lower memory and computational footprint and allows capturing temporal trends even with noisy data. However, due to the small window as an input and the use of simple or no recurrent models, the latest models are unable to capture long-term dependencies effectively. A recently proposed HitAnomaly [Huang et al., 2020] method uses vanilla transformers as encoder-decoder networks, but is only applicable to natural-language log data and not appropriate for generic continuous time-series data as inputs. Other methods, such as TopoMAD [He et al., 2020], use a topology-aware neural network that is composed of a Long-Short-Term-Memory (LSTM) and a variational autoencoder (VAE) to detect faults. However, the reconstruction error is only obtained for the latest state, limiting them to using reactive fault recovery policies. Similar methods use slight variations of LSTM networks with either dropout layers [Girish and Rao, 2021], causal Bayesian networks [Gan et al., 2020] or recurrent autoencoders [Chouliaras and Sotiriadis, 2021]. A GAN-based approach that uses a stepwise training process, StepGAN [Feng et al., 2021], converts the input time-series into matrices and executes convolution operations to capture temporal trends. However,

such techniques are not agnostic to the number of hosts or workloads as they assume a maximum limit of the active tasks in the system. Moreover, even though more accurate than heuristic-based approaches, deep learning models such as deep autoencoders, GANs and recurrent networks are adaptive and accurate, but have a high memory footprint that adversely affects system performance. To resolve this, some works have been proposed that have low memory footprint, such as SlimGANs [Hou et al., 2021].

Clustering Models. Very recent works also propose a few-shot learning method for fault detection [Won and Kim, 2021]. Other recent methods utilize deep neural networks to execute fuzzy clustering [Hu et al., 2021, Li et al., 2018]. For instance, the Adaptive Weighted Gath-Geva (AWGG) [Hu et al., 2021] clustering method is an unsupervised model that detects faults using stacked sparse autoencoders to reduce detection times. Such methods train using supervised labels and do not present a mechanism to recover from faults once detected, and hence can not be used to develop end-to-end fault tolerance in Fog systems.

2.3.3.2 AI Augmented Fault Remediation

When Edge servers fail or are unavailable, optimal migration of the running tasks is crucial. However, it is also important to ensure that the task placement and scheduling procedures are fault-aware and aim to minimize system faults to minimize the overheads of running remediation strategies.

Fault-Aware Scheduling. Recently, several resilience models have been proposed that leverage AI methods like RL, surrogate or reconstruction modeling. Many of these methods run proactive scheduling and task placement steps to avoid faults in a future state. An RL based approach is Load Balancing and Optimization Strategy (LBOS) [Talaat et al., 2020] that allocates the resources using RL. The reward of the RL agent is calculated as a weighted average of multiple QoS metrics to avoid system contention by balancing the load across multiple compute nodes. The values of the weights are determined using genetic algorithms. LBOS observes the network traffic constantly, gathers the statistics about the load on each Edge server, manages the arriving user requests and uses dynamic resource allocation to assign them to available Edge nodes. However, RL approaches are known to be slow to adapt in dynamic settings (see Chapter 3). Most other approaches use neural networks as a surrogate model. For instance, Effective Load Balancing Strategy (ELBS) [Talaat et al., 2019] is a recent framework that offers an execution environment for IoT applications and creates an interconnect among Cloud and Edge servers. The ELBS method uses the priority scores to proactively allocate tasks to Edge nodes or worker nodes as brokers to avoid system failures. It uses a fuzzy inference system to calculate the priority scores of different tasks based on three fuzzy inputs: SLA deadline, user-defined priority, and estimated task processing time. The priority values are generated by a neural network acting in the capacity of a surrogate of QoS scores. The Proactive Coordinated Fault Tolerance (PCFT) [Liu et al., 2016] method uses Particle Swarm Optimization (PSO) to reduce the overall transmission overhead, network consumption and total execution time for a set of tasks. This method first predicts faults in the running host machines by anticipating resource deterioration and uses PSO to find target hosts for preemptive migration decisions. This approach mainly focuses on reducing transmission overheads in distributed Cloud setups but often fails to improve the I/O performance of the compute nodes. CSAVM [Satpathy et al., 2018] uses another evolutionary Crow Search scheme to take live migration decisions for the task queues. The method is used to optimize the power consumption of a compute setup by preventing unnecessary migrations. DDQP [Wang et al., 2021] uses double deep Q-networks to place services on network nodes.

Load Balancing for Tolerance. Load balancing is a concept that proactively aims to balance the load on different elements of a Fog infrastructure to avoid a faulty future system state. One such work, namely DPSO [Jan et al., 2021], proposes network gateways to host the load balancing logic where they monitor the load across Edge servers and balance the load using evolutionary algorithms. Furthermore, a migration mechanism is also incorporated where application modules are rearranged to achieve a balanced load across Edge servers. Migration is triggered based on a machine learning-based dynamic threshold. Similarly, FOCALB [Kaur and Aron, 2021] proposes a hybrid load balancing algorithm based on Grey Wolf Optimization (GWO) and Ant Colony Optimization (ACO), where energy consumption, execution time and implementation cost of scientific workflows are achieved by uniformly distributing the workload across Fog devices to optimize the Fog resource utilization. Similarly, some methods [Arabnejad et al., 2017] propose a fuzzy logic based weighting scheme to run load-balancing task placement. Other methods, like PEFS [Marahatta et al., 2020], present a prediction-based energy aware fault-tolerant load balancing scheme that uses a neural network to predict faults in the system and run load-balancing strategies to ensure a high resource utilization ratio.

Automatic Scaling. Many methods aim to optimally decide how to scale the Fog infrastructure to avoid or recover from faults in the system. Similar to resource provisioning (Section 2.3.1.2), here too, it is vital to avoid over-provisioning and under-provisioning of limited Fog resources under dynamic workloads. An RNN based method [Etemadi et al., 2021] provides a deep learning based solution to utilize metrics such as resource requests (*i.e.*, CPU, RAM, etc.) and Fog resource status (e.g., CPU efficiency, storage utilization, network traffic, active/inactive resources) to make optimum auto-scaling decisions. Moreover, AI-augmented auto-scaling methods have the potential to support proactive auto-scaling of containers under dynamic workload fluctuations. Similarly, [Abdullah et al., 2020] presents a predictive auto-scaling policy using decision tree regression (DTR) model where a reactive rule-based auto-scaling mechanism is employed to train the proactive model under multiple heterogeneous workloads. Existing works explore the use of AI-augmented workload forecasting [Li et al., 2020, Etemadi et al., 2020] to make proactive auto-scaling decisions within Edge/ Fog environments. Another method, MADRP [Li et al., 2020], uses a hybrid ARIMA and DNN model to forecast the workloads, whereas a nonlinear autoregressive (NAR) neural network is used by [Etemadi et al., 2020] to predict the future demands for the Fog devices. Methods such as FLBFH [Naha et al., 2021], propose a fuzzy logic-based method to handle unpredicted and predicted failures in Fog environments. Such methods predict two failure scores to decide what actions to be undertaken to handle failures for unreliable Fog devices. The first failure score is based on device mobility, device response time and device power availability. This score determines the checkpointing interval as a proactive mechanism for unpredicted failures. The second score is based on CPU utilization, device mobility, device response time, device power availability, device communication. This score is used to decide about preemptive task migration. In some cases where the rate of unpredicted failure is high, the proposed fuzzy-logic mechanism will suggest an application replication. An extension to this work is the Dependency and Topology-aware Failure Resilience (DTFR) algorithm, which considers failure probability, response time and the number of replicas to schedule services on Edge servers [Aral and Brandić, 2021]. DTFR explores the spatio-temporal failure dependency among Edge servers to develop a dynamic method with minimum redundancy to enhance the failure resilience of services. Other works [Aral and Brandić, 2017], continue this trend to estimate the availability level of VMs in Edge Data Centers (EDCs) based on Bayesian Networks. The probabilistic models consider dependencies between different failures such as hardware, software, or network

failures, and power outage. This model is utilized to select VMs that can meet the availability requirements in SLA. Another similar method is the Fuzzy-based Real-Time Auto-scaling (FRAS) [Etemadi et al., 2021] technique that leverages a virtualized environment for the recovery of IoT applications that run on compromised or faulty Edge nodes. Here, FRAS executes each IoT application in a virtual machine (VM) and performs VM autoscaling to improve execution speed and reduce execution costs. The VM autoscaling decisions making involves inference of system QoS using a fuzzy recurrent neural network as a surrogate model.

Straggler Aware Models. Another common performance problem in Fog systems is dealing with straggler tasks that are slow running instances that increase the overall response time. Such tasks can significantly impact the system’s QoS and the SLA. Methods, such as JFP [Aral and Brandic, 2018], exploit failure dependencies between Edge servers to predict the failure probability of given service deployment. JFP evaluates the use of replication in Edge servers based on analyzing historical failure logs of individual servers, modeling temporal dependencies as a Dynamic Bayesian Network (DBN), and predicting the probability at which a certain number of servers fail simultaneously. It also uses two replica scheduling algorithms to optimize failure probability and the cost of redundancy in an Edge computing environment.

2.3.4 Limitations

Our study on related work as well as those in the past [Beloglazov and Buyya, 2012, Nashaat et al., 2019, Samanta et al., 2019] have shown how heuristic based approaches perform well in a variety of scenarios for optimization of diverse QoS parameters. Methods such as *LR-MMT* schedules workloads dynamically based on local regression (LR) to predict which hosts might get overloaded in the near future and select containers/tasks which can be migrated in the minimum migration time (MMT). Both LR and MMT heuristics are used for overload detection and task selection, respectively [Beloglazov and Buyya, 2012, Nashaat et al., 2019]. However, such heuristic based approaches fail to model stochastic environments with dynamic workloads [Tuli et al., 2020]. Other prior works have shown that evolutionary based methods, and generally gradient-free approaches, perform well in dynamic scenarios [Wang et al., 2019, Han et al., 2018, Wang et al., 2020]. Evolutionary approaches like genetic algorithms (GA) lie in the domain of gradient-free optimization methods. The GA based resource management methods make decisions using a neural model to approximate the objective value and a genetic-algorithm to reach the optimal decision [Han et al., 2018]. However, gradient-free methods have a number of drawbacks. They take much longer to converge [Bogolubsky et al., 2016] and are not as scalable [Rios and Sahinidis, 2013] as gradient-based methods. Moreover, non-local jumps can lead to significant change in the scheduling decision which can lead to a high number of migrations. Over the years, MaxWeight scheduling has gained popularity for its theoretical guarantees and ability to reduce resource contention [Liu et al., 2020, Krishnasamy et al., 2018]. However, MaxWeight policies can exhibit poor delay performance, instability in dynamic workloads and spatial inefficiency [Bae et al., 2019, van de Ven et al., 2009, 2013]. Recently, reinforcement learning based methods have shown themselves to be robust and versatile to diverse workload characteristics and complex edge setups [Tang et al., 2018, Basu et al., 2019, Gazori et al., 2019]. However, such methods are still slow to adapt to real-world application scenarios. This leads to higher wait times and subsequently high response times and SLO violations. Moreover, these methods do not scale well [Nandi et al., 2001]. Finally, coupled or symbiotic simulation and model based control have long been used in the modelling and optimization of distributed systems [Tuli et al., 2019, Bosmans et al., 2019, Onggo et al., 2018]. Many prior works have used hybrid

simulation models to optimize decision making in dynamic systems. To achieve this, they monitor, analyze, plan and execute decisions using previous knowledge-base corpora (MAPE-k) [Gill et al., 2019]. However, such works use this to facilitate search methods and not to generate additional data to aid the decision making of an AI model.

2.4 Summary

This chapter shows how the shift in technological landscape to heterogeneous fog continuums has given rise to new research avenues. The shift to contemporary AI based workloads with increased resource requirements and tight service response deadlines makes the management of fog resources challenging. Although several works have been proposed for efficiently managing resources across the three scopes of deployment, scheduling and dependability, they face several limitations. The technical contributions of this thesis aim to address these limitations. Specifically, the surrogate based models with gradient-based optimization facilitate quick decision making, thanks to the goal directed search (more details in Chapters 3-6). Further, co-simulation allows us to generate additional out-of-distribution data to facilitate robust training and test out candidate decisions for more informed planning (more details in Chapters 7-9).

This page intentionally left blank.

Chapter 3

COSCO Framework and GOBI Optimization

This chapter describes the novel COSCO framework and the gradient-based optimization approaches using surrogate models that we utilize throughout the rest of the thesis for the objectives described in Chapter 1.

Existing Solutions. In Chapter 2 we discuss several approaches for resource optimization in fog environments, particularly to optimize scheduling decisions. We show that gradient-based optimization methods outperform gradient-free methods, thanks to their fast and accurate decision updates. In light of this, we formulate a gradient-based optimization algorithm (**GOBI**) which calculates the gradients of neural networks with respect to input for optimizing QoS parameters using advanced gradient-based optimization strategies. We establish by experiments that our approach provides a *faster and more scalable optimization strategy* for fog scheduling compared to the state-of-the-art methods.

New Insights. However, simply using gradient-based optimization is not sufficient as data driven neural models can sometimes saturate [Rakitienskaia and Engelbrecht, 2015]. This is when feeding more data to the neural model does not improve the performance. Further optimization of QoS in such cases is hard and more intelligent schemes are required. Coupled-simulation (also referred as co-simulation or symbiotic simulation) and execution of tasks have been shown to be a promising approach for quickly obtaining an estimate of QoS parameters in the near future [Gomes et al., 2018, Rehman et al., 2019, Onggo et al., 2018]. Specifically, coupled-simulation allows us to run a simulator in the background with the scheduling algorithms to facilitate decision making. However, prior works use this to aid search methods and not to generate more data to facilitate decision making of an AI model. The latter requires development of a new interface between the scheduler and simulator. Hence, we develop **COSCO: Coupled Simulation and Container Orchestration Framework** to leverage simulated results to yield better QoS. The COSCO framework is the *first* to allow a single or multi-step simulation of container migration decisions in fog environments. It enables a scheduler to get an estimate of the QoS parameters at the end of a future interval for **better prediction and subsequently optimum schedules**. Container migration refers to the process of moving an application between different physical or virtual hosts and resuming computation on the target host. This allows us to run the GOBI approach, simulate the schedule (with a predicted workload model) and provide the objective values to another neural approximator which can then better approximate the objective function and lead to improved performance. We call this novel

optimization loop **GOBI***. The *interactive training dynamic* between GOBI and GOBI* aids the latter to converge quickly and adapt to volatile environments (more details in Section 3.4).

Research Contributions. The key contributions of this chapter are:

- We present a novel framework, COSCO, which is the first to allow *coupled simulation and container orchestration* in fog environments.
- We propose a gradient based back-propagation approach (GOBI) for *fast and scalable scheduling* and show that it outperforms the state-of-the-art schedulers.
- We also propose an extended approach (GOBI*) that leverages simulation results from COSCO running GOBI’s decision, to provide *improved predictions and scheduling decisions* with lower scheduling overheads.

Validating both GOBI and GOBI* on physical setups with real-world benchmark data shows that, GOBI gives lower performance than GOBI*. However, GOBI is more suitable for resource constrained fog brokers, because of its low computational requirements. GOBI*, on the other hand, is more suitable for setups with critical QoS requirements and powerful fog brokers.

3.1 System Model and Problem Formulation

3.1.1 System Model

We consider a standard distributed and heterogeneous fog computing environment as shown in Figure 3.1. We assume a single fog datacenter with geographically distributed edge and cloud layers as computational nodes. We consider network latency effects for interactions between compute devices in different layers (edge and cloud) and between compute devices and fog broker. We ignore communication latencies among devices in the same layer. Tasks are container instances with inputs being generated by sensors and other IoT devices and results received by actuators which physically manifest the intended actions. These devices constitute the IoT layer and send and receive all data from the fog gateway devices. All management of tasks and hosts is done by a Fog Broker in the management layer. This includes task scheduling, data management, resource monitoring and container orchestration. The work described in this chapter is concerned with improving the scheduler in the Fog Broker by providing an interface between the container orchestration and resource monitoring services with a simulator. Our work proposes discrete-time controllers, that are commonly adopted in the literature [Basu et al., 2019, Mahmud et al., 2018]. The communication between end-users and the fog broker is facilitated by gateway devices. The compute nodes in the fog resource layer, which we refer to as “hosts”, can have diverse computational capabilities. The hosts at the edge of the network are resource-constrained and have low communication latency with the broker and gateway devices. On the other hand, cloud hosts are several hops from the users and have much higher communication latency and are computationally more powerful. We assume that there are a fixed number of host machines in the fog resource layer and denote them as $H = \{h_0, h_1, \dots, h_{N-1}\}$. We denote the collection of time-series utilization metrics, which include CPU, RAM, Disk and Network Bandwidth usage of host h_i as $\mathcal{U}(h_i^t)$, averaged over the time-interval. The collection of maximum capacities of CPU, RAM, Disk and Network Bandwidth with the average communication latency of host h_i is denoted as $\mathcal{C}(h_i)$.

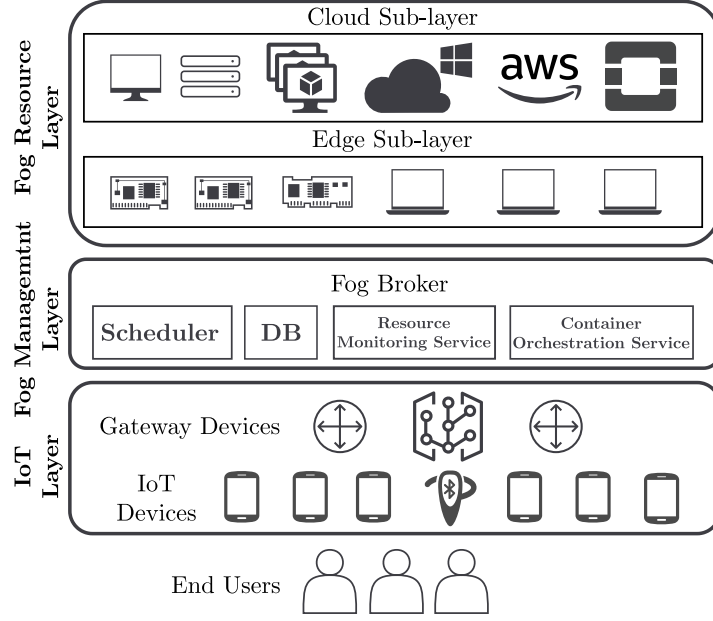


Figure 3.1: System model

3.1.2 Workload Model

We now present our workload model, as summarized in Figure 3.2. We divide a bounded timeline into equal sized scheduling intervals with equal duration Δ . The t -th interval is denoted by I_t and starts at $s(I_t)$, so that $s(I_0) = 0$ and $s(I_t) = s(I_{t-1}) + \Delta \forall t > 0$. In interval I_{t-1} , N_t new tasks are created by the IoT devices. The gateway devices then send batches of these new tasks N_t with their SLO requirements to the Fog Broker. The SLO violations are measured corresponding to the response time metrics of task executions as a fraction of them that exceeds the stipulated deadlines. The set of active tasks in the interval I_t is denoted as $A_t = \{a_0^t, a_1^t, \dots, a_{|A_t|}^t\}$ and consists of $|A_t|$ tasks. Here a_j^t represents an identifier for the j -th task in A_t . The Fog Broker schedules the new tasks to the compute nodes and decides which of the active tasks need to be migrated. At the end of interval I_{t-1} , the set of completed tasks is denoted as L_t , hence the tasks that carry forward to the next interval is the set $A_{t-1} \setminus L_t$. Thus, the broker takes a decision $\mathcal{D}^t = \mathcal{D}(Y_t)$, where Y_t is the union of the new (N_t), active ($A_{t-1} \setminus L_t$) and waiting tasks (W_{t-1}). This decision is a set of allocations and migrations in the form of ordered-pairs of tasks and hosts (assuming no locality constraints). At the first scheduling interval, as there are no active or waiting tasks, the model only takes allocation decision for new tasks (N_0). Here, $\mathcal{D}$ denotes the scheduler such that $\mathcal{D} : Y_t \mapsto Y_t \times H$. Only if the allocation is feasible, *i.e.*, depending on whether the target host can accommodate the scheduled task or not, is the migration/allocation decision executed. Initially, the waiting queue and the set of active and leaving actions are empty. The executed decision is denoted by $\hat{\mathcal{D}}$ and satisfies the following properties

$$\hat{\mathcal{D}}(A_t) \subseteq \mathcal{D}(Y_t), \text{ where} \quad (3.1)$$

$$\forall (a_j^t, h_i) \in \hat{\mathcal{D}}(A_t), \mathcal{U}(a_j^t) + \mathcal{U}(h_i^t) \leq \mathcal{C}(h_i). \quad (3.2)$$

We use the notation $a_j^t \in \hat{\mathcal{D}}(A_t)$ to mean that this task was allocated/migrated. The set of new and waiting tasks $\hat{N}_t \cup \hat{W}_{t-1} \subseteq \hat{\mathcal{D}}(A_t)$, where $\hat{N}_t \subseteq N_t$ and $\hat{W}_{t-1} \subseteq W_{t-1}$, denotes those tasks which could be allocated successfully. The remaining new tasks, *i.e.* $N_t \setminus \hat{N}_t$ are added to get the new wait queue W_t . Similarly, active

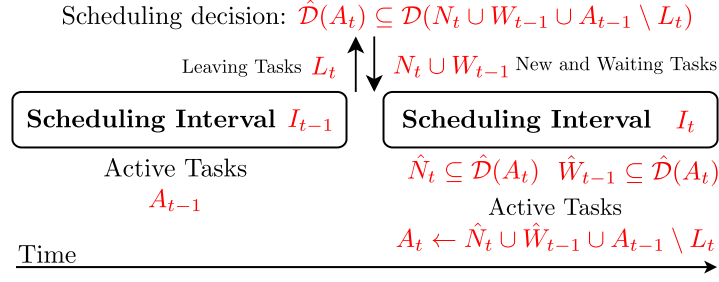


Figure 3.2: Dynamic Task Workload Model

tasks are denoted as $\hat{A}_{t-1} \subseteq A_{t-1}$. Hence, for every interval I_t ,

$$A_t \leftarrow \hat{N}_t \cup \hat{W}_{t-1} \cup A_{t-1} \setminus L_t \quad (3.3)$$

$$W_t \leftarrow (W_{t-1} \setminus \hat{W}_{t-1}) \cup (N_t \setminus \hat{N}_t), \text{ where} \quad (3.4)$$

$$W_{-1} = A_{-1} = L_{-1} = \emptyset. \quad (3.5)$$

We also denote the utilization metrics of an active task $a_j^t \in A_t$ as $\mathcal{U}(a_j^t)$ for the interval I_t . We also utilize metrics such as response time and SLO violation rates as an average across all completed tasks in that interval. We denote a simulator as $\mathcal{S} : \hat{\mathcal{D}}(A_t) \times \{\mathcal{U}(a_j^t) | \forall a_j^t \in A_t\} \mapsto \{\mathcal{U}(h_i^t) | \forall h_i \in H\} \times P$, which takes scheduling decision and container utilization characteristics to give host characteristics and values of QoS parameters. Here $P_t \in P$ is a set of QoS parameters such as energy consumption, response times, SLO violations, etc. Similarly, execution on a physical fog framework is denoted by $\mathcal{F}$. Hence, execution of interval I_t on a physical setup is denoted as

$$(\{\mathcal{U}(h_i^t) | \forall h_i \in H\}, P_t) \equiv \mathcal{F}(\hat{\mathcal{D}}(A_t), \{\mathcal{U}(a_j^t) | \forall a_j^t \in A_t\}). \quad (3.6)$$

3.1.3 Problem Formulation

After execution of tasks A_t in interval I_t , the objective value to minimize is denoted as $\mathcal{O}(P_t)$. $\mathcal{O}(P_t)$ could be a scalar value which combines multiple QoS parameters. To find the optimum schedule, we need to minimize the objective function $\mathcal{O}(P_t)$ over the entire duration of execution. Thus, we need to find the appropriate and feasible decision function $\mathcal{D}$ such that $\sum_t \mathcal{O}(P_t)$ is minimized. This is subject to the constraints that at each scheduling interval the new tasks N_t , waiting tasks W_{t-1} and active tasks from the previous interval $A_{t-1} \setminus L_t$ are allocated using this decision function. The problem can then be concisely formulated as:

$$\begin{aligned} & \underset{\mathcal{D}}{\text{minimize}} && \sum_{t=0}^T \mathcal{O}(P_t) \\ & \text{subject to} && \forall t, Eqs.(3.1) - (3.6). \end{aligned} \quad (3.7)$$

3.2 The GOBI Scheduler

We now present the approach based on Gradient Based Optimization using back-propagation to Input (GOBI). This constitutes the *Scheduler* module of the Fog Broker described in Figure 3.1. Here, we consider that we optimize an objective function $\mathcal{O}(P_t)$ at every interval I_t via taking an optimal action in the form of scheduling

decision $\mathcal{D}$. This $\mathcal{D}$ is then used by the framework/simulator to execute tasks as described in optimization program (3.7).

3.2.1 Objective Function

We will now present how our optimization scheme can be used for taking appropriate scheduling decisions in a fog environment. To optimize the QoS parameters, we consider an objective function that focuses on energy consumption and response time, two of the most crucial metrics for fog environments [Mahmoud et al., 2018, Mutlag et al., 2019]. For interval I_t ,

$$\mathcal{O}(P_t) = \alpha \cdot AEC_t + \beta \cdot ART_t. \quad (3.8)$$

Here AEC and ART ($\in P_t$) are defined as follows.

1. *Average Energy Consumption* (AEC) is defined for any interval as the energy consumption of the infrastructure (which includes all edge and cloud hosts) normalized by the maximum power of the hosts, *i.e.*,

$$AEC_t = \frac{\sum_{h_i \in H} \int_{t=s(I_t)}^{s(I_{t+1})} Power_{h_i}(t) dt}{|A_t| \sum_{h_i \in H} Power_{h_i}^{max} \times (t_{i+1} - t_i)}, \quad (3.9)$$

where $Power_{h_i}(t)$ is the power function of host h_i at instant t , and $Power_{h_i}^{max}$ is maximum possible power of h_i .

2. *Average Response Time* (ART)¹ is defined for an interval I_t as the average response time for all leaving tasks (L_t), normalized by maximum response time until the current interval, as shown below

$$ART_t = \frac{\sum_{l_j^t \in L_t} Response\ Time(l_j^t)}{|L_t| \max_{s \leq t} \max_{l_j^s \in L_s} Response\ Time(l_j^s)}. \quad (3.10)$$

Here, α and β are hyper-parameters set by the industry adopters or the cloud service providers based on the relative importance of the two optimization objectives (energy consumption and response time) for the running services. These values may be obtained using a search strategy before deploying the services. We used grid-search for our experiments.

3.2.2 Input Parameters

In concrete implementations, we can assume a finite maximum number of active tasks, with the upper bound denoted as M . Thus at any interval, $|A_t| \leq M$. Moreover, we consider the utilization metrics of Instructions per second (IPS), RAM, Disk and Bandwidth consumption which form a feature vector of size F . Then, we express task utilization metrics $\{\mathcal{U}(a_j^{t-1}) | \forall j, a_j^{t-1} \in A_{t-1}\}$ as an $M \times F$ matrix denoted as $\phi(A_{t-1})$, where the first $|A_{t-1}|$ rows are the feature vectors of tasks in A_{t-1} in the order of increasing creation intervals (breaking ties uniformly at random), and the rest of the rows are 0. We also form feature vectors of hosts with IPS, RAM, Disk and Bandwidth utilization with capacities and communication latency of each host, each of size F' . Thus,

¹Both AEC and ART are unit-less metrics and lie between 0 and 1

at each interval I_t for $|H|$ hosts, we form a $|H| \times F'$ matrix $\phi(H_{t-1})$ using host utilization metrics of interval I_{t-1} .

Finally, as we have N_t, W_{t-1}, A_{t-1} and L_t at the start of I_t , the decision matrix $\phi(\mathcal{D})$ is an $M \times |H|$ matrix with the first $|N_t \cup W_{t-1} \cup A_{t-1} \cup L_t|$ rows being one-hot vector of allocation of tasks to one of the H hosts. The remaining rows being 0.

3.2.3 Model Training

We now describe how a neural model can be trained to approximate $\mathcal{O}(P_t)$ using the input parameters $[\phi(A_{t-1}), \phi(H_{t-1}), \phi(\mathcal{D})]$. Consider a continuous function $f(x; \theta)$ as a neural approximator of $\mathcal{O}(P_t)$ with the θ vector denoting the underpinning neural network parameters and x as a continuous or discrete variable with a bounded domain. Here, x is the collection of utilization metrics of tasks and hosts with a scheduling decision. The parameters θ are learnt using the dataset $\Lambda = \{[\phi(A_{t-1}), \phi(H_{t-1}), \phi(\mathcal{D})], \mathcal{O}(P_t)\}_t$ such that a given loss function $\mathcal{L}$ is minimized for this dataset. We form this dataset by running a random scheduler and saving the traces generated (more details about the setup in Section 3.5). The loss $\mathcal{L}$ quantifies the dissimilarity between the predicted output and the ground truth. We use Mean Square Error (MSE) as the loss function as done in prior work [Tuli et al., 2020]. Hence,

$$\mathcal{L}(f(x; \theta), y) = \frac{1}{T} \sum_{t=0}^T (y - f(x; \theta))^2, \text{ where } (x, y) \in \Lambda.$$

Thus, for datapoints $(x, y) \in \Lambda$, where y is the value of $\mathcal{O}(P_t)$ for the corresponding x , we have $\theta = \arg \min_{\hat{\theta}} \sum_{(x, y) \in \Lambda} [\mathcal{L}(f(x; \hat{\theta}), y)]$. To do this, we calculate the gradient of the loss function with respect to θ as $\nabla_{\theta} \mathcal{L}$ and use back-propagation to learn the network parameters. By the universal approximation theorem [Scarselli and Tsoi, 1998], we can assume that we already have parameters θ after network training, such that this neural network approximates a generic function to an arbitrary degree of precision (considering a sufficiently large parameter set θ). A key part of this ability of neural networks rests on randomly initializing a parameterized function and changing the large number of parameters based on the loss function.

Now, with this, we need to find the appropriate decision matrix $\phi(\mathcal{D})$, such that $f(x; \theta)$ is minimized. We formulate the resulting optimization problem as

$$\begin{aligned} & \underset{\phi(\mathcal{D})}{\text{minimize}} && f(x; \theta) \\ & \text{subject to} && \text{each element of } \phi(\mathcal{D}) \text{ is bounded} \\ & && \forall t, \text{ Eqs. (3.1) - (3.6)}. \end{aligned} \tag{3.11}$$

This is a reformulation of the program (3.7) using the neural approximator of $\mathcal{O}(P_t)$ as the objective function. Note that, because there is a bounded space of inputs, there must be an optimal solution, *i.e.*, $\exists \hat{\phi}(\mathcal{D})$ such that

$$\begin{aligned} f([\phi(A_{t-1}), \phi(H_{t-1}), \hat{\phi}(\mathcal{D})]; \theta) &\leq \\ f([\phi(A_{t-1}), \phi(H_{t-1}), \phi(\mathcal{D})]; \theta), &\forall \text{ feasible } \phi(\mathcal{D}). \end{aligned}$$

We solve this optimization problem using gradient-based methods, for which we need $\nabla_x f(x; \theta)$. This is uncommon for neural network training since prior work normally modifies the weights using input and ground truth values. Thus the gradients can be calculated without having to update the weights. Now, for a typical feed-forward artificial neural network, $f(x; \theta)$ is a composition of linear layers and non-linear activation functions. Just like the back-propagation approach, we can find gradients with respect to input x . For simplicity, we consider non-affine functions like $\tanh()$ and $\text{softplus}()$ as activations in our models, with no effect to the generality of the neural approximator. For instance, gradients for a single linear layer with $\tanh()$ non-linearity is given in Proposition 1².

Proposition 1 (Finding gradients with respect to input). *For a linear layer with $\tanh()$ non-linearity that is defined as follows:*

$$f(x; W, b) = \tanh(W \cdot x + b),$$

the derivative of this layer with respect to x is given as:

$$\nabla_x f(x; W, b) = W^T \times (1 - \tanh^2(W \cdot x + b)).$$

Proof. It is well known that $\nabla_x \tanh(x) = 1 - \tanh^2(x)$, therefore,

$$\begin{aligned} \nabla_x f(x; W, b) &= \nabla_x \tanh(W \cdot x + b) \\ &= \nabla_{W \cdot x + b} \tanh(W \cdot x + b) \times \nabla_x (W \cdot x + b) \\ &= (1 - \tanh^2(W \cdot x + b)) \times \nabla_x (W \cdot x + b) \\ &= W^T \times (1 - \tanh^2(W \cdot x + b)) \end{aligned} \quad \square$$

Using the chain-rule, we can find the derivative of an arbitrary neural approximator f as $f(x; \theta)$ is a composition of multiple such layers. Once we have the gradients, we can initialize an arbitrary input x and use gradient descent to minimize $f(x; \theta)$ as

$$x_{n+1} \leftarrow x_n - \gamma \cdot \nabla_x f(x_n; \theta), \quad (3.12)$$

where γ is the learning rate and n is the iteration count. We apply Eq. (3.12) until convergence. The calculation of $\nabla_x f(x_n; \theta)$ in the above equation is carried out in a similar fashion as back-propagation for model parameters. We can use momentum based methods like Adam [Kingma and Ba, 2014] or annealed schedulers like cosine annealing with warm restarts [Loshchilov and Hutter, 2016] to prevent the optimization loop getting stuck in local optima. As we show in our experiments, the combination of gradient calculation with respect to inputs and advances like momentum and annealing allow *faster* optimization *avoiding* local minima, giving a more robust and efficient approach.

3.2.4 Scheduling

After training the model f , at start of each interval I_t , we optimize $\phi(\mathcal{D})$ by the following rule (γ is the learning rate) until the absolute value of the gradient is more than the convergence threshold ϵ , *i.e.*,

²Derivations for other activation functions with a single linear layer are given in Supplementary Section 2.

Algorithm 3.1 The GOBI scheduler**Require:**

```

Pre-trained function approximator  $f(x; \theta)$ 
Dataset used for training  $\Lambda$ ; Convergence threshold  $\epsilon$ 
Iteration limit  $\sigma$ ; Learning rate  $\gamma$ 
Initial random decision  $\mathcal{D}$ 
1: procedure MINIMIZE( $\mathcal{D}, f, z$ )
2:   Initialize decision matrix  $\phi(\mathcal{D})$ ;  $i = 0$ 
3:   do
4:      $x \leftarrow [z, \phi(\mathcal{D})]$  ▷ Concatenation
5:      $\delta \leftarrow \nabla_{\phi(\mathcal{D})} f(x; \theta)$  ▷ Partial gradient
6:      $\phi(\mathcal{D}) \leftarrow \phi(\mathcal{D}) - \gamma \cdot \delta$  ▷ Decision update
7:      $i \leftarrow i + 1$ 
8:   while  $|\delta| > \epsilon$  and  $i \leq \sigma$ 
9:   Convert matrix  $\phi(\mathcal{D})$  to scheduling decision  $\mathcal{D}^*$ 
10:  return  $\mathcal{D}^*$ 
11: end procedure
12: procedure GOBI(scheduling interval  $I_t$ )
13:  if ( $t == 0$ )
14:    Initialize random decision  $\mathcal{D}$ 
15:  else
16:     $\mathcal{D} \leftarrow \mathcal{D}^*$  ▷ Output for the previous interval
17:  Get  $\phi(A_{t-1}), \phi(H_{t-1})$ 
18:   $\mathcal{D}^* \leftarrow \text{MINIMIZE}(\mathcal{D}, f, [\phi(A_{t-1}), \phi(H_{t-1})])$ 
19:  Fine-tune  $f$  with loss =
20:     $MSE(\mathcal{O}(P_{t-1}), f([\phi(A_{t-2}), \phi(H_{t-2}), \phi(\mathcal{D}^{t-1})]; \theta))$ 
21:  return  $\mathcal{D}^*$ 
22: end procedure

```

$$\phi(\mathcal{D})_{n+1} \leftarrow \phi(\mathcal{D})_n - \gamma \cdot \nabla_{\phi(\mathcal{D})} f(x_n; \theta). \quad (3.13)$$

In dynamic scenarios, as the approximated function for the objective $\mathcal{O}(P_t)$ changes with time we continuously fine-tune the neural approximator as per the loss function $MSE(\mathcal{O}(P_{t-1}), f([\phi(A_{t-2}), \phi(H_{t-2}), \phi(\mathcal{D}^{t-1})]; \theta))$ (line 20 in Algorithm 3.1). Equation 3.13 then leads to a decision matrix with a lower objective value (line 6 in Algorithm 3.1). Hence, *gradient based iteration with continuous fine-tuning of the neural approximator allows GOBI to adapt quickly in dynamic scenarios*. When the above equation converges to $\phi(\mathcal{D}^*)$, the rows represent the likelihood of allocation to each host. We take the arg max over each row to determine the host to which each task should be allocated as $\mathcal{D}^*$. This then becomes the final scheduling decision of the GOBI approach. The returned scheduling decision is then run on a simulated or physical platform (as in Eq. (3.6)) and tasks sets are updated as per Eqs. (3.3)-(3.5). To find the executed decision $\hat{\mathcal{D}}$, we sort all tasks in $\mathcal{D}^*$ in descending order of their wait times (breaking ties uniformly at random) and then try to allocate them to the intended host in $\mathcal{D}^*$. If the allocation is not possible due to utilization constraints, we ignore such migrations and do not add them to $\hat{\mathcal{D}}$. At run-time, we fine tune the trained neural model f , by newly generated data for the model to adapt to new settings.

3.3 The COSCO Framework

Motivated from prior coupled-simulators [Bosmans et al., 2019], we now present the COSCO framework. A basic architecture is shown in Figure 3.3. Implementation details of the COSCO framework are given in

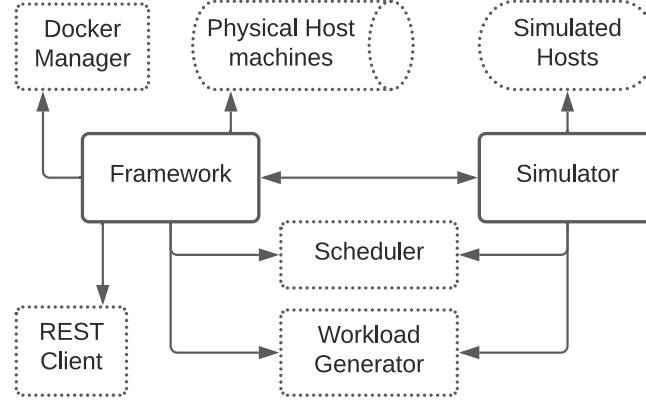


Figure 3.3: COSCO architecture

Supplementary Section 1. The hosts might correspond to simulated hosts or physical compute nodes with Instructions per second (IPS), RAM, Disk and Bandwidth capacities with power consumption models with CPU utilization. The workloads are either time-series models of IPS, RAM, Disk, Bandwidth requirements in the case of simulation or are actual application programs in the case of physical experiments. At the beginning of each scheduling interval, N_t workloads are created. The framework also maintains a waiting queue for workloads that cannot be allocated in the previous interval (W_{t-1}).

3.3.1 Simulator

The *Simulator* uses a *Scheduler* (GOBI for instance), which uses the utilization metrics and QoS parameters to take an allocation decision for new workloads or migration decisions for active workloads. At the start of a scheduling interval I_t , the simulator destroys workloads completed in I_{t-1} , denoted as L_t . It also gets workloads $N_t \cup W_{t-1}$. The *Scheduler* decides which host to allocate these tasks to and whether to migrate active tasks from $A_{t-1} \setminus L_t$. The *Simulator* object considers all $N_t \cup W_{t-1}$ tasks in the order of the interval at which they were created. If the target host is unable to accommodate a task, it is added to the wait queue. This is done for each utilization metric of active task a_j^t in $\mathcal{U}(a_j^t)$. As for new tasks, the utilization metrics are unknown, $u = 0 \forall u \in \mathcal{U}(n_j^t) \forall n_j^t \in N_t$. To prevent overflow of host resources, we compare for each new task n_j^t the sum of the maximum possible utilization metrics of the corresponding application and target host against the host's maximum capacity (more details in Supplementary Section 1). Hence, at each interval, Eq. (3.5) holds. Instead of running the tasks on physical machines, we use trace driven discrete-event simulation.

3.3.2 Framework

The *Framework* instantiates the tasks as Docker containers. Docker is a container management platform as a service used to build, ship and run containers on physical or virtual environments. Moreover, the hosts are physical computational nodes in the same Virtual Local Area Network (VLAN) as the server machine on which the *Scheduler* program runs. The server communicates with each host via HTTP REST APIs. At every host machine, a *DockerClient* service runs with a *Flask* HTTP web-service [Grinberg, 2018]. *Flask* is a web application REST client which allows interfacing between hosts and the server. To migrate a container, we use the Checkpoint/Restore In Userspace (CRIU) [Venkatesh et al., 2019] tool. For each migration decision $(a_j^t, h_i) \in \hat{\mathcal{D}}(A_t)$, a checkpoint of a_j^t is created which is equivalent to a snapshot of the container. Then,

Algorithm 3.2 The GOBI* scheduler**Require:**

```

Pre-trained function approximator  $f^*(x; \theta^*)$ 
Pre-trained LSTM model  $LSTM(\{\mathcal{U}(a_j^t) \forall t' < t\})$ 
Dataset used for training  $\Lambda^*$ ; Convergence threshold  $\epsilon$ 
Learning rate  $\gamma$ ; Initial random decision  $\mathcal{D}$ 
1: procedure GOBI*(scheduling interval  $I_t$ )
2:   if ( $t == 0$ )
3:     Initialize random decision  $\mathcal{D}$ 
4:   else
5:      $\mathcal{D} \leftarrow \mathcal{D}^*$  ▷ Output for the previous interval
6:   Get  $\phi(A_{t-1}), \phi(H_{t-1})$ 
7:    $\bar{\mathcal{U}}(a_j^t) \forall a_j^t \in A_t \leftarrow LSTM(\{\mathcal{U}(a_j^{t'}) \forall t' < t\})$ 
8:    $\bar{\mathcal{D}} \leftarrow GOBI(I_t)$ 
9:    $\{\mathcal{U}(h_i^t) | \forall h_i \in H\}, A\bar{E}C_t, A\bar{R}T_t \leftarrow$ 
10:      $\mathcal{S}(\bar{\mathcal{D}}(A_t), \{\bar{\mathcal{U}}(a_j^t) \forall a_j^t \in A_t\})$ 
11:    $\mathcal{O}(\bar{P}_t) \leftarrow \alpha \cdot A\bar{E}C_t + \beta \cdot A\bar{R}T_t$ 
12:    $\mathcal{D}^* \leftarrow \text{MINIMIZE}(\mathcal{D}, f, [\phi(A_{t-1}), \phi(H_{t-1}), \mathcal{O}(\bar{P}_t)])$ 
13:   Fine-tune  $f^*$  with loss =
14:      $MSE(\mathcal{O}(P_{t-1}), f([\phi(A_{t-2}), \phi(H_{t-2}), \phi(\mathcal{D}^{t-1})]; \theta)) + \mathbb{1}(\mathcal{O}(\bar{P}_{t-1}) < \mathcal{O}(P_{t-1})) \times MSE(\bar{\mathcal{D}}^{t-1}, \mathcal{D}^{t-1})$ 
15:   return  $\mathcal{D}^*$ 
16: end procedure

```

this snapshot is transferred (migrated) to the target host h_i , where the container is resumed (restored). The allocation and migrations are done in a non-blocking fashion, *i.e.* the execution of active containers is not halted when some tasks are being migrated.

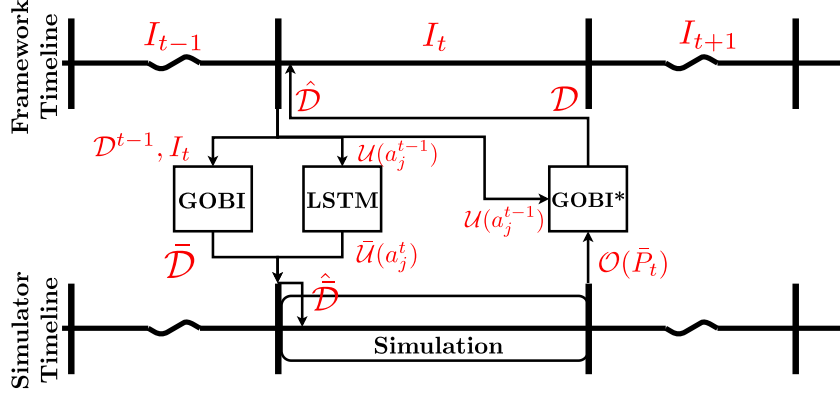
3.3.3 Model Interface

We now discuss the novelty of the COSCO framework. As all scheduling algorithms and simulations are run on a central fog broker [Mahmud et al., 2018], it can be easily run in fog environments which follow a master-slave topology. As discussed in Section 3.1, a simulator can execute tasks on simulated host machines to return QoS parameters P . To execute an interval I_t , the simulator needs utilization metrics $\mathcal{U}(a_j^t), \forall a_j^t \in A_t$, and a scheduling decision $\bar{\mathcal{D}}^t$ to be executed on the simulator. Thus, at the beginning of the interval I_t , we have L_t, N_t, W_{t-1} and A_{t-1} . After checking for the possibility of allocation/migration, we get $\hat{\bar{\mathcal{D}}}(A_t), \hat{N}_t$ and $\hat{W}_{t-1}$. Now, using Eq. (3.3), we find A_t . Using given utilization metrics $\mathcal{U}(a_j^t)$ and $\hat{\bar{\mathcal{D}}}(A_t)$, we execute A_t tasks on the simulator to get $\mathcal{S}(\hat{\bar{\mathcal{D}}}(A_t), \{\mathcal{U}(a_j^t) | \forall a_j^t \in A_t\}) = \{\mathcal{U}(h_i^t) | \forall h_i \in H\}, \bar{P}_t$. This means that at the beginning of interval I_t , the COSCO framework allows simulation of the next scheduling interval (with an action of interest $\bar{\mathcal{D}}$ and predicted utilization metrics) to predict the values of the QoS parameters in the next interval $\bar{P}_t$. This single step look-ahead simulation allows us to take better scheduling decisions in the GOBI* algorithm, as shown in the next section.

3.4 Extending GOBI to GOBI*

3.4.1 Leveraging Co-Simulations

Now that we have a scheduling approach based on back-propagation of gradients, we extend it to incorporate simulated results. To do this, we use the following components.

Figure 3.4: Iteration of GOBI* approach at the start of interval I_t

1. *GOBI Scheduler*: We assume that we have the GOBI scheduler which can give us a preferred action of interest $\bar{\mathcal{D}}$. Thus, at the start of a scheduling interval I_t , we get $\bar{\mathcal{D}} = \text{GOBI}(I_t)$ (line 8 in Algorithm 3.2). $\mathcal{D}$ now denotes the decision of GOBI*.
2. *Utilization prediction models*: We train a utilization metric prediction model, such that using previous utilization metrics of the tasks, we get a predicted utilization metric set for the next interval. As is common in prior work [Gupta and Dinesh, 2017], we use a Long-Short-Term-Memory (LSTM) neural network for this and train it using the same Λ dataset that we used for training the GOBI neural approximator. Thus, at the start of interval I_t , using $\bar{\mathcal{D}}$ from GOBI and checking allocation possibility, we get $\hat{N}_t$ and $\hat{W}_{t-1}$. Using (3.3), we get A_t . Then, we predict $\bar{U}(a_j^t), \forall a_j^t \in A_t$, using the LSTM model (line 7 in Algorithm 3.2). Hence, we get the $\phi(A_{t-1}), \phi(H_{t-1})$ matrices.
3. *Simulator*: Now that we have an action $\bar{\mathcal{D}}$ and predicted utilization metrics $\bar{U}(a_j^t), \forall a_j^t \in A_t$, we can use the simulator to predict the QoS parameters at the end of I_t as described in Section 3.3.3 (lines 9 and 10 in Algorithm 3.2).

Since at the start of the interval I_t , we do not have utilization models of all containers and subsequently hosts for I_t , our GOBI model is forced to predict QoS metrics at the end of I_t using only utilization metrics of the previous interval, *i.e.*, I_{t-1} . This can make GOBI's predictions inaccurate in some cases. However, if we can predict with reasonable accuracy the utilization metrics in I_t using the utilization values of the previous intervals, we can simulate and get an improved estimate of the QoS parameters during the next period I_t . Adding these as inputs to another neural approximator for $\mathcal{O}(P_t)$ denoted as $f^*(x, \theta^*)$, we now have $x = [\phi(A_{t-1}), \phi(H_{t-1}), \mathcal{O}(\bar{P}_t), \phi(\mathcal{D})]$.

Again, using a random scheduler and pre-trained GOBI model, we generate a dataset Λ^* to train this new neural approximator f^* . At execution time, we freeze the neural model f of GOBI and fine tune f^* with the loss function. This loss function is the MSE of objective values. We add the MSE between the predicted action $\mathcal{D}$ by GOBI* and $\bar{\mathcal{D}}$ by GOBI in case the estimated objective value of GOBI is lower than that of GOBI*. Thus,

$$\mathcal{L} = \text{MSE}(\mathcal{O}(P_{t-1}), f^*([\phi(A_{t-2}), \phi(H_{t-2}), \phi(\mathcal{D}^{t-1})]; \theta^*)) + \mathbb{1}(\mathcal{O}(\bar{P}_{t-1}) < \mathcal{O}(P_{t-1})) \times \text{MSE}(\bar{\mathcal{D}}^{t-1}, \mathcal{D}^{t-1}). \quad (3.14)$$

Using this trained model, we can now provide schedules as shown in Algorithm 3.2 and Figure 3.4. At interval I_t , the previous output of GOBI*, *i.e.*, $\mathcal{D}^{t-1}$ is given to the GOBI model as the initial decision. The GOBI model uses utilization metrics to output the action $\hat{\mathcal{D}}$. Using a single-step simulation, we obtain an estimate of

the objective function for this decision. Then, GOBI* uses this estimate to predict the next action D . Based on whether GOBI*'s decision is better than GOBI or not, GOBI* is driven to the decision which has lower objective value. *This interactive dynamic should allow GOBI* to make a more informed prediction of QoS metrics for I_t and hence perform better than GOBI.*

3.4.2 Convergence of GOBI and GOBI*

We outline two scenarios of an ideal and a more realistic case to describe the interactive dynamic between GOBI and GOBI* and how the latter optimizes the objective scores.

Ideal Case: Consider the case when the neural approximator of GOBI, f can *perfectly* predict the objective value of the next interval I_t as $\mathcal{O}(P_t)$. In this case, the predicted action $\bar{\mathcal{D}}$ is optimal. In such a setting, we expect GOBI* to converge such that it predicts the same action as $\bar{\mathcal{D}}$. Assuming an ideal simulator which *exactly* mimics the real-world, $\mathcal{O}(\bar{P}_t) = \mathcal{O}(P_t)$. Thus, any action other than $\bar{\mathcal{D}}$ is sub-optimal. Hence, to minimize the loss metric, GOBI* would directly forward $\bar{\mathcal{D}}$ and ignore all other information and converge to GOBI (which is optimal).

Real Case: While the limited explainability of neural networks prevent us from giving formal convergence properties, we supply of number of technical observations that justify the convergence we have seen in practice when applying the methods to real systems. Thus, considering the real-case, time-bound training only gives a sub-optimal approximator f for GOBI. Hence, in general the predicted decision $\mathcal{D}$ is not always optimal. Assuming GOBI* predicts a decision other than GOBI, GOBI* will sometimes get a sub-optimal action $\mathcal{D}$ such that $\mathcal{O}(\bar{P}_t) < \mathcal{O}(P_t)$ and other times $\mathcal{O}(\bar{P}_t) > \mathcal{O}(P_t)$. The former is the case when the action predicted by GOBI, $\bar{\mathcal{D}}$, is better and the latter is the case when the GOBI* action, $\mathcal{D}$, has a lower objective score. (1) In the case of $\mathcal{D}$ being better than $\bar{\mathcal{D}}$ (having lower objective score), in an attempt to minimize the first part of the loss,

$$\mathcal{L}_1 = \text{MSE}(\mathcal{O}(P_{t-1}), f^*([\phi(A_{t-2}), \phi(H_{t-2}), \phi(\mathcal{D}^{t-1})]; \theta^*)), \quad (3.15)$$

GOBI* would converge to avoid $\bar{\mathcal{D}}$ (considering the ideal simulator that returns exact answers). Furthermore, the predicted action $\mathcal{D}$ has a lower objective score, hence the optimization loop would tend to converge to this or better scheduling decisions. (2) In the case of $\bar{\mathcal{D}}$ being better than $\mathcal{D}$, which means that $\mathcal{O}(\bar{P}_{t-1}) < \mathcal{O}(P_{t-1})$, GOBI* is given an incentive to predict $\bar{\mathcal{D}}$. This is because the latter part of the loss makes the model converge such that it minimizes

$$\mathcal{L}_2 = \text{MSE}(\bar{\mathcal{D}}^{t-1}, \mathcal{D}^{t-1}). \quad (3.16)$$

This, with the former part of the loss, creates an interactive dynamic such that the GOBI* model can find scheduling decisions even better than the one predicted by the GOBI model. However, in cases when GOBI*'s prediction is worse, the GOBI's prediction aids the model to quickly converge to a better decision (that of GOBI's). Thus, in both cases, each model update based on the loss in Eq. (3.14), yields a scheduling decision with objective score same or lower than that given by GOBI. Assuming a sufficiently long training time, GOBI* would eventually converge to predict $\mathcal{D}$ such that the objective value when the decision is $\mathcal{D}$ is never greater than when it is $\bar{\mathcal{D}}$. We observe in our experiments that in the real-case GOBI* always performs better than GOBI in terms of objective scores. The interactive training with simulations allows GOBI* to converge quickly and adapt to volatile environments.

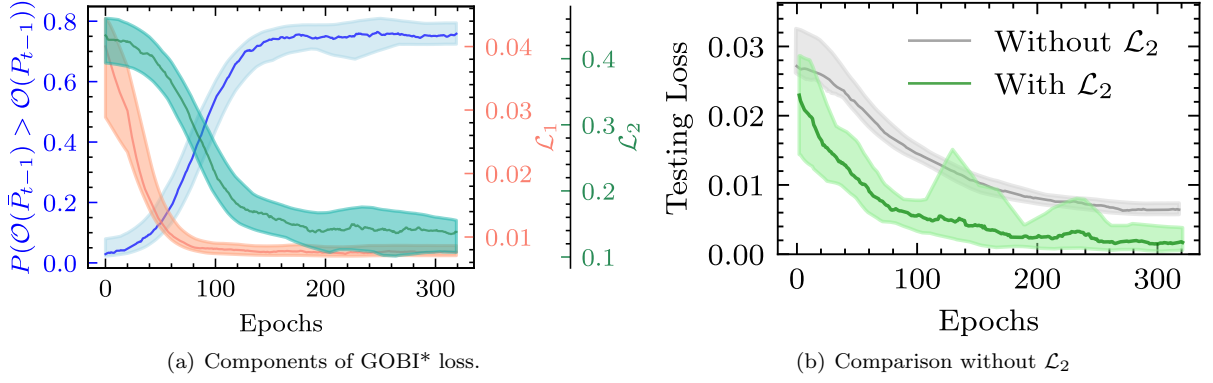


Figure 3.5: Convergence plots for GOBI*.

The convergence plots for GOBI* are shown in Figure 3.5. It demonstrates that as the model gets trained, the probability of the GOBI*'s actions being better than GOBI increases. This means that initially $\mathcal{L}_2$ has a high contribution to the loss for model training which aids GOBI* to quickly converge to GOBI's performance. This can be seen by the sudden drop in $\mathcal{L}_1$ in Figure 3.5(a). This is also shown by Figure 3.5(b) that GOBI*'s loss converges faster with the additional $\mathcal{L}_2$ component.

3.5 Performance Evaluation

3.5.1 Experimental Setup

To test the efficacy of the proposed approaches and compare against the baseline methods, we perform experiments on both simulated and physical platforms. As COSCO has the same underlying models of task migration, workload generation and utilization metric values for both simulation and physical test-bench, we can test all models on both environments with the same underlying assumptions. As in prior work [Tuli et al., 2020], we use $\alpha = \beta = 0.5$ in (3.8) for our experiments. Moreover, we consider all tasks, to be allocated and migrated, in batches with each batch having feature vectors corresponding to up to $M = |H|^2$ tasks. If number of tasks is less than M , we pad the input matrix with zero vectors. This style of inference strategy is in line with prior work [Gill et al., 2020]. This is due to the limitation of non-recurrent neural networks to take a fixed size input for a single forward pass, hence requiring action inference for tasks being executed in batches.

Physical Environment: We use the Microsoft Azure cloud provisioning platform to create a test-bed of 10 VMs located in two geographically distant locations. The gateway devices are considered to be in the same LAN of the server which was hosted in London, United Kingdom. Of the 10 VMs, 6 were hosted in London and 4 in Virginia, United States. The host capacities $\mathcal{C}(h_i) \forall h_i \in H$ are shown in Table 3.1 and Supplementary Table 1. All machines use Intel Haswell 2.4 GHz E5-2673 v3 processor cores. The cost per hour (in US Dollar) is calculated based on the costs of similar configuration machines offered by Microsoft Azure at the datacenter in the South UK region³. The power consumption models are taken from the SPEC benchmarks repository⁴. We run all experiments for 100 scheduling intervals, with each interval being 300 seconds long, giving a total experiment time of 8 hours 20 minutes. We average over 5 runs and use diverse workload types to ensure statistical significance.

³Microsoft Azure pricing calculator for South UK <https://azure.microsoft.com/en-gb/pricing/calculator/>

⁴More details in Supplementary Section 4

Table 3.1: Host characteristics of Azure fog environment.

Name	Quantity	Core count	MIPS	RAM	RAM Bandwidth	Ping time	Network Bandwidth	Disk Bandwidth	Cost Model	Location
Edge Layer										
Azure B2s server	4	2	4029	4295 MB	372 MB/s	3 ms	1000 MB/s	13.4 MB/s	0.0472 \$/hr	London, UK
Azure B4ms server	2	4	8102	17180 MB	360 MB/s	3 ms	1000 MB/s	10.3 MB/s	0.1890 \$/hr	London, UK
Cloud Layer										
Azure B4ms server	2	4	8102	17180 MB	360 MB/s	76 ms	1000 MB/s	10.3 MB/s	0.166 \$/hr	Virginia, USA
Azure B8ms server	2	8	2000	34360 MB	376 MB/s	76 ms	2500 MB/s	11.64 MB/s	0.333 \$/hr	Virginia, USA

Simulation Environment: We consider 50 hosts machines as a scaled up version of the 10 machines from the last subsection. Here, each category has 5 times the instance count to give a total of 50 machines in a comparatively larger-scale fog environment as considered in prior art [Basu et al., 2019, Ahmed and Pierre, 2018]. As we cannot place simulated nodes in geographically distant locations, we model the latency and networking characteristics of these nodes in our simulator as per Table 3.1.

The two tests with 10 VMs in the physical environment and 50 VMs in the simulated environment are arbitrarily chosen. The former is for more precise experiments and the latter is to test whether the trends match in larger scale setups.

3.5.2 Model Training and Assumptions

For the GOBI and GOBI* algorithms we use standard feed-forward neural models with the following characteristics. We use non-affine activation functions for our neural approximators to be differentiable for all input values.

- Input layer of size $M \times F + |H| \times F' + M \times N$ for GOBI and $M \times F + |H| \times F' + M \times N + 1$ for GOBI*. The non-linearity used here is `softplus`⁵ as in [Tuli et al., 2020]. Note, $|H|$ may vary from 10 in tests in the physical environment to 50 in the simulator.
- Fully connected layer of size 128 with `softplus` activation.
- Fully connected layer of size 64 with `tanhshrink` activation.
- Fully connected layer of size 1 with `sigmoid` activation.

To implement the proposed approach, we use PyTorch Autograd package [Paszke et al., 2017] to calculate the gradients of the network output with respect to input keeping model parameters constant. We generate training data for the GOBI model f by running a random scheduler for 2000 intervals on the simulator. We then run the random scheduler on the framework for 300 intervals and fine-tune the GOBI model on this new dataset. For GOBI*, we use a random scheduler and predictions of the GOBI model f to generate data of the form described in Section 3.2 of size 2000 on simulator and fine-tune it on data corresponding to 300 intervals on the framework. The dataset was used to create the matrices $[\phi(A_{t-1}), \phi(H_{t-1}), \phi(\mathcal{D})]$, where each column of $\phi(A_{t-1}), \phi(H_{t-1})$ is normalized by the maximum and minimum utilization values, and $\phi(\mathcal{D})$'s rows are one-hot decision vectors. To train a model, we use the AdamW optimizer [Saleh and Mashaly, 2019] with learning rate

⁵The definitions of these activation functions can be seen at the PyTorch web-page: <https://pytorch.org/docs/stable/nn.html>

10^{-5} and randomly sample 80% of data to get the training set and the rest as the cross-validation set⁶. The convergence criterium used for model training is the loss of consecutive epochs summed over the last 10 epochs is less than 10^{-2} . For GOBI*, we also train predictors for utilization metrics. Specifically, we use a LSTM neural network [Hochreiter and Schmidhuber, 1997] with a single LSTM cell to achieve this and train it using the same dataset used to train the GOBI model.

3.5.3 Workloads

To generate workloads for training the GOBI and GOBI* models and testing them against baseline methods, we use two workload characteristics *Bitbrain traces* and *DeFog applications*. These were chosen because of their non-stationarity, their highly volatile workloads and their similarity with many real-world applications.

1. *BitBrain traces*. The dynamic workload is generated for cloudlets based on the real-world, open-source Bitbrain dataset [Shen et al., 2015]⁷. This dataset consists of real traces of resource utilization metrics from 1750 VMs running on BitBrain distributed datacenter. The workloads running on these servers are from a variety of industry applications including computational analytical programs used by major banks, credit operators and insurers [Shen et al., 2015] and are commonly used for benchmarking fog-cloud models [Khamse-Ashari et al., 2018, Li et al., 2017]. The dataset consists of workload information for each time-stamp (separated by 5 minutes) including the number of requested CPU cores, CPU usage in terms of Million Instructions per Second (MIPS), RAM requested with Network (receive/transmit) and Disk (read/write) bandwidth characteristics. These different categories of workload data constitute the feature values of $\phi(A_{t-1})$ and $\phi(H_{t-1})$. As these workload traces correspond to applications running on real infrastructure, we use these to run our simulations and generate training data. At the start of each interval I_t , the size of the new tasks N_t set follows a discrete Poisson distribution⁸ $Poisson(\lambda)$, as per prior works [He et al., 2021]. Here, $\lambda = 1.2$ jobs for $|H| = 10$ and $\lambda = 5$ jobs for $|H| = 50$.
2. *DeFog applications*. DeFog [McChesney et al., 2019] is a fog computing benchmark which consists of six real-time heterogeneous workloads such as Yolo, Pocketsphinx, Aeneas, FogLamp and iPokeMon. We use three specific heterogeneous applications of DeFog: Yolo (Memory, Bandwidth and Compute Intensive benchmark), PocketSphinx (Compute Intensive) and Aeneas (Bandwidth Intensive). Yolo runs with 1500 user requests and each request uploads a unique image with different density and can run upto three intervals. PochetSphinx runs with 320 user requests and Aeneas with 1500. Here too, at the start of each interval I_t , the size of the new tasks N_t set follows a Poisson distribution $Poisson(\lambda)$ with $\lambda = 1.2$ jobs. Out of the N_t tasks, the distribution of selection of Yolo/Pocketsphinx/Aeneas follows the probabilities of (p_y, p_p, p_a) for a run. For our final comparison experiments we run using $p_y = p_p = p_a = 0.33$. However, to test diverse workload characteristics, we also run using any one as 0.80 and other two as 0.10. We call these workload dominant runs. We perform all our experiments to compare with baselines using the DeFog benchmarks.

⁶All model training and experiments were performed on a system with configuration: Intel i7-10700K CPU, 64GB RAM, Nvidia GTX 1060 and Windows 10 Pro OS.

⁷The BitBrain dataset can be downloaded from: <http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains>

⁸The Poisson distribution models the number of independent arrivals that occur in a period, so it is apt to model for a batch size if the batch is formed by arrivals in a time window (scheduling interval in our case).

3.5.4 Baseline Models

We evaluate the performance of the proposed algorithms and compare them against the state-of-the-art scheduling approaches. The reasons for comparing against these baselines are described in Chapter 2. We consider two light-weight heuristic based schedulers *LR-MMT* and *MAD-MC* that have low scheduling times, but higher energy consumption, response times and SLO violations. We also consider a deep-learning based gradient-free optimization method *GA*, and two reinforcement-learning based schedulers *DQLCM* and *A3C*. Finally, we compare against a one max-weight based allocation method *POND*.

3.5.5 Evaluation Metrics

We use the following evaluation metrics to test the GOBI and GOBI* models as motivated from prior works [Basu et al., 2019, Gill et al., 2019, Beloglazov and Buyya, 2012]. We also use *AEC* and *ART* as in Section 3.2.

1. *SLO Violations* which is given as

$$\frac{\sum_t \sum_{l_j^t \in L_t} \mathbb{1}(\text{Response Time}(l_j^t) \leq \psi(l_j^t))}{\sum_t |L_t|},$$

where $\psi(l_j^t)$ is the 95th percentile response time for this application type (Yolo/PocketSphinx/Aeneas for DeFog and random/sequential for Bitbrain) on the state of the art baseline *A3C*. This definition of percentile-based SLO is defined for the response time metrics of completed tasks and is inspired from [Bolor et al., 2010].

2. *Fairness* which is given as the Jain's Fairness Index

$$\frac{(\sum_t \sum_{l_j^t \in L_t} \text{Response Time}(l_j^t))^2}{(\sum_t |L_t|) \times (\sum_t \sum_{l_j^t \in L_t} \text{Response Time}(l_j^t)^2)}.$$

3. *Average Migration Time* which is the average time for the migration of tasks over all intervals in the run

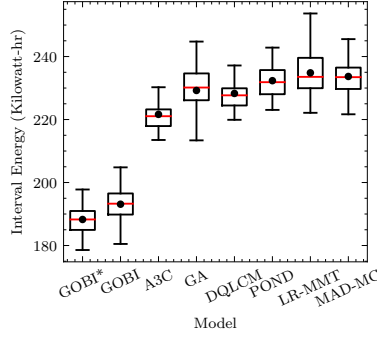
$$\frac{\sum_t \sum_{(a_j^t, h_i) \in \hat{\mathcal{D}}(A_t)} \text{Migration Time}(a_j^t, h_i)}{\sum_t |\hat{\mathcal{D}}(A_t)|}.$$

4. *Scheduling Time* which is the average time to reach the scheduling decision over all intervals in the run.
5. *Average Wait Time* which is the average time for a container in the wait queue before it starts execution.

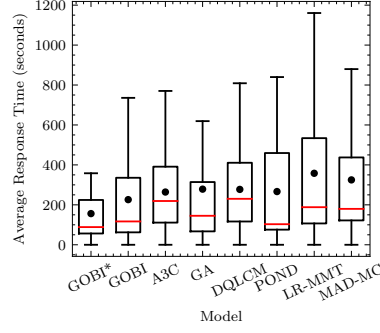
3.5.6 Results

In this section we provide comparative results, showing how GOBI and GOBI* perform against other baselines as described in Section 3.5.4. We compare the proposed approaches with respect to the evaluation metrics described in Section 3.5.5. Additional results are given in Supplementary Section 5. The graphs in Figure 3.6 show the results for runs with length of 100 scheduling intervals viz 8 hours 20 minutes using the DeFog workloads on 10 physical Azure machines. Figure 3.7 shows similar trends in the result for 50 hosts in a simulated environment.

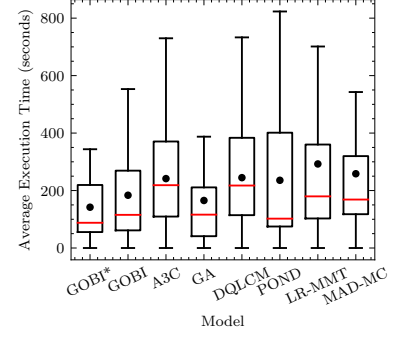
Figure 3.6(a) shows the energy consumption of Azure host machines for each scheduling policy. Among the baseline approaches, *A3C* consumes the least average interval energy of 221.63 KW-hr. Running the *GOBI*



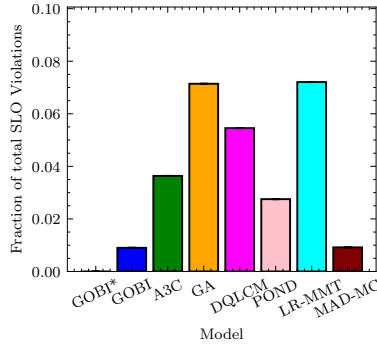
(a) Average Energy Consumption



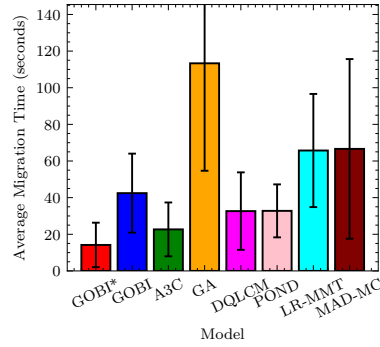
(b) Average Response Time



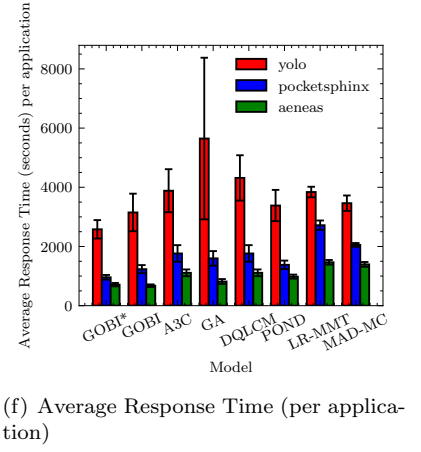
(c) Average Execution Time



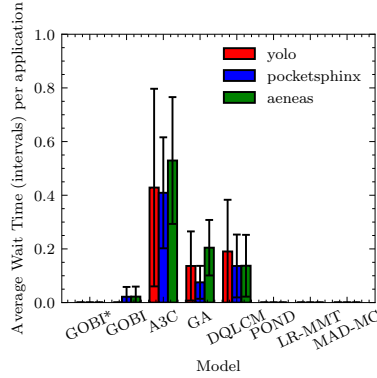
(d) Fraction of SLO Violations



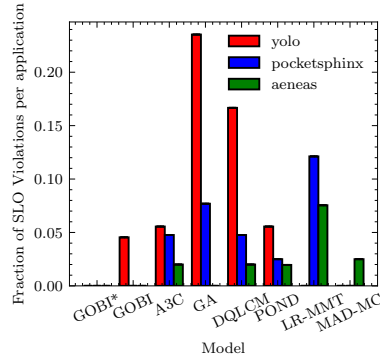
(e) Average Migration Time



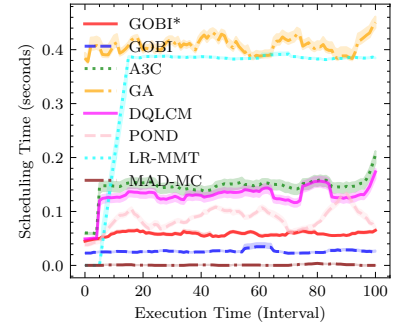
(f) Average Response Time (per application)



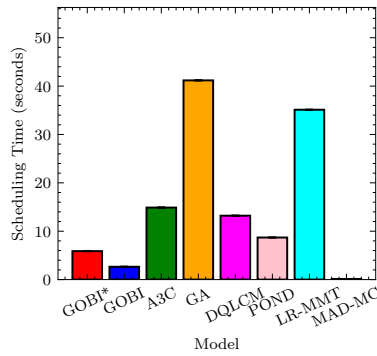
(g) Average Wait Time (per application)



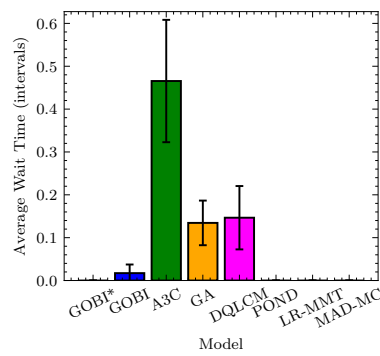
(h) Average SLO Violations (per application)



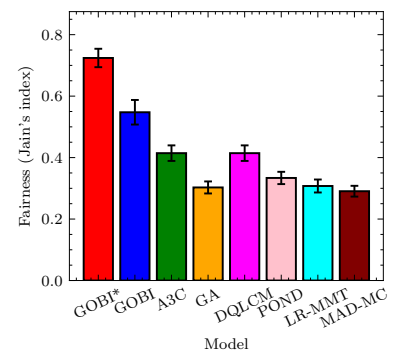
(i) Scheduling Time versus execution time



(j) Average Scheduling Time



(k) Average Wait Time



(l) Fairness

Figure 3.6: Comparison of GOBI and GOBI* against baselines on framework with 10 hosts

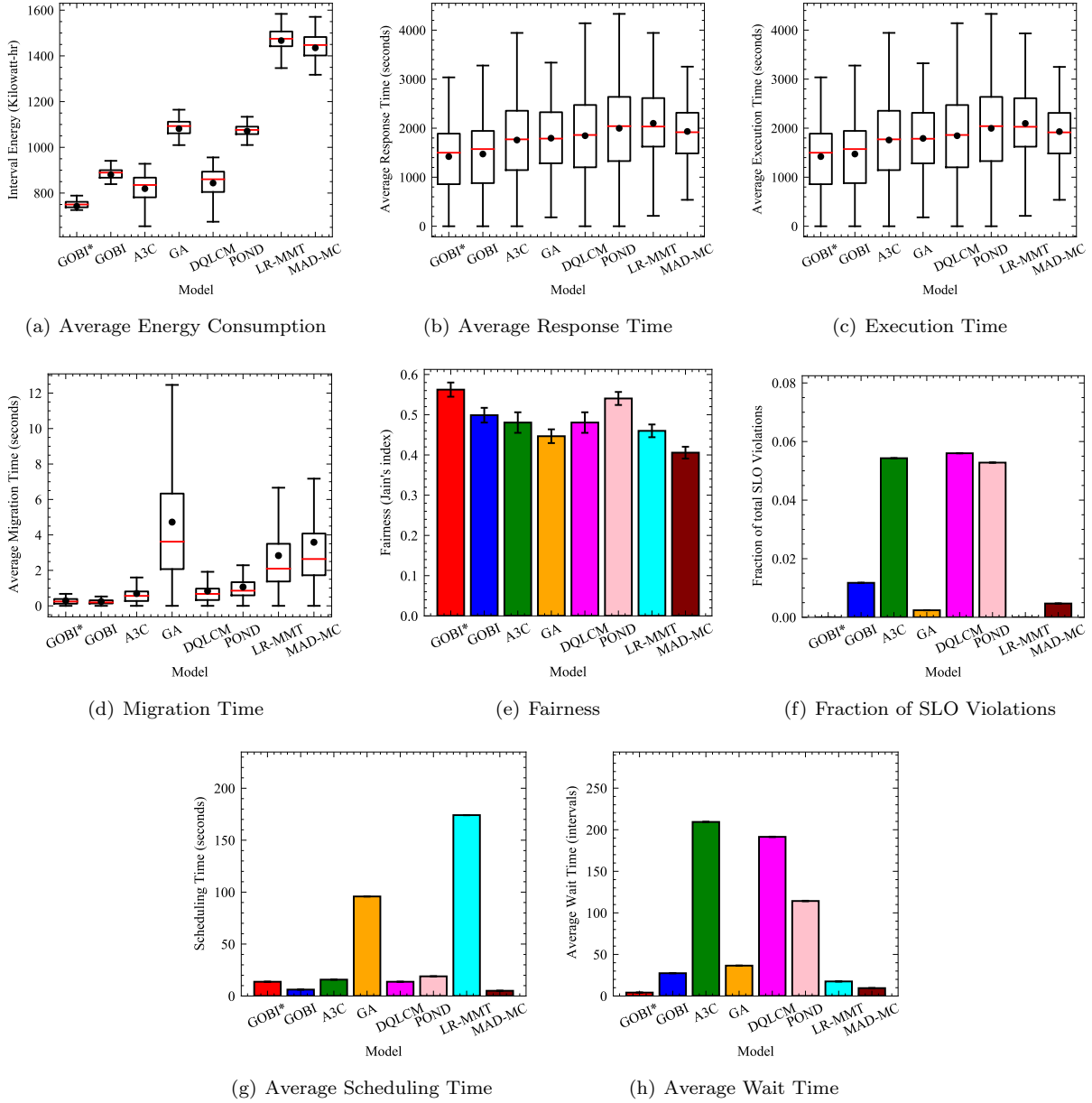


Figure 3.7: Comparison of GOBI and GOBI* against baselines on simulator with 50 hosts

approach on the Azure platform consumes 193.11 KW-hr, 12.86% lower than *A3C*. Further, *GOBI** consumes the least energy 188.26 KW-hr (15.05% lower than that of *A3C*). The major reason for this is the low response time of each application, which leads to more tasks being completed within the same duration of 100 intervals. Thus, even with similar energy consumption across all policies, *GOBI* and *GOBI** have a much higher task completion rate of 112-115 tasks in 100 intervals compared with only 107 tasks. Similarly, when we consider 50 host machines in a simulated platform (Figure 3.7(a)), *GOBI** still consumes 9.39% lower than *A3C*. The energy consumption with *GOBI* is very close to *A3C*.

Figure 3.6(b) shows the average response time for each policy. Here, response time is the time between the creation of a task from an IoT sensor and up to the gateway receiving the response. Among the baselines models, *POND* has the lowest average response time of 266.53 seconds. Here, *GOBI* and *GOBI** have 226.04 and 156.09 seconds (15.19% – 41.43% better than *POND*). For the three applications, the average response time for each policy is shown in Figure 3.6(f). Clearly, among the DeFog benchmark applications, Yolo takes the

maximum time. The highest response time for Yolo among all policies is for *GA* as it prefers scheduling shorter jobs first *i.e.* Pocketsphinx and Aeneas to reduce response time. This is verified by the results which show that Aeneas has the lowest response time when the *GA* policy is used. Another disadvantage of this preference is that it leads to high wait times when running the *GA* policy. However, in larger scale experiments on 50 hosts in simulation (Figure 3.7(b)), *GA* has much lower wait times leading to the best response times among the baseline algorithms. Here, compared to *GA*, *GOBI* and *GOBI** have 17.98% – 20.87% lower average response times, respectively.

Figure 3.6(k) shows the average waiting time (in intervals) for tasks running on the Azure framework with each policy. As prior works have established that RL approaches are slow to adapt in highly volatile environments [Findling et al., 2019], the *A3C* and *DQLCM* approaches are unable to adapt when a host is running at capacity. This means that for a large number of scheduling intervals (47.72%), they predict an overloaded host. Hence, the task cannot be assigned in the same interval and has to wait until either it is assigned to another host, or the resources of this host are released. This is also reflected in the wait time per application (Figure 3.6(g)). As established in prior work, reinforcement-learning approaches scale poorly and hence this waiting time gets more pronounced when running with 50 hosts (Figure 3.7(h)). However, as *GOBI* and *GOBI** are able to adapt quickly to the environment, because of the neural model update in each iteration, they do not face such problems even on larger scale experiments.

As seen in Figure 3.6(d), due to the high response times of the baseline approaches, they have a high SLO violation rate. For the experiments of physical hosts, as *POND* has the lowest response time, the SLO violation rate is also the lowest among the baselines (2.75%). *GOBI* has only 0.9% SLO violations and *GOBI** has none. This is due to the back-propagation approach which minimizes the response time, as well as avoiding local optima using the adaptive moment estimation approach (Adam optimizer). For *GOBI*, the SLO violations are only for Yolo tasks (4.5%). In the simulation environment, due to the high wait and response times of the *A3C*, *DQLCM* and *POND* schedulers, their SLO violation rates are highest among all methods. Here, *GOBI* has only 1.1% SLO violations. Due to the high response times of *A3C* and *DQLCM* for 50 hosts, the SLO itself is much higher giving the *GA* approach with low response times only 0.2% SLO violations. *GOBI** has no SLO violations in this case as well.

The main reason for low response times in *GOBI* and *GOBI** is their low scheduling time. Even with single-step simulation, the back-propagation based optimization strategy with the AdamW optimizer *converges faster than the gradient-free optimization* methods like *A3C*, *DQLCM* *GA* and *POND*. This is apparent from the scheduling times seen in Figures 3.6(j) and 3.6(i) (shaded region shows the 95% confidence interval). Here, *MAD-MC* has the lowest scheduling time, with *GOBI* next at 2.65 seconds and *GOBI** at 5.88 seconds. Even with 50 hosts (Figure 3.7(g)), *GOBI* and *GOBI** have one of the lowest scheduling times across all policies (except *MAD-MC*).

Figure 3.6(e) shows the average migration times for all policies. Here, *GA* has the highest migration time due to the largest number of migrations. This is as expected due to the *non-local jumps* in the *GA* approach which lead to a high number of migrations. Approaches like the back-propagation method (*GOBI* and *GOBI**) and RL based approaches (*A3C* and *DQLCM*) have comparatively low migration times. A similar trend is also apparent with 50 hosts as seen in Figure 3.7(d). Due to convergence to better optima, the execution times itself for iterative optimization approaches including *GOBI*, *GOBI** and *GA* are the lowest among all approaches

Table 3.2: Comparison between GOBI and GOBI*

Algorithm	Scheduling Time	Objective Value	MSE
GOBI	2.65±0.019 s	0.5673	$2.31 \pm 0.15 \times 10^{-3}$
GOBI*	5.88±0.015 s	0.4744	$1.98 \pm 0.22 \times 10^{-3}$

(Figures 3.6(c) and 3.7(c)). Finally, Figures 3.6(l) and 3.7(e) show that *GOBI* and *GOBI** are the most fair schedulers among all that have been tested.

3.5.7 Comparing GOBI and GOBI*

Now that *GOBI* and *GOBI** have been shown to out-perform prior works, we directly compare the two approaches. Table 3.2 shows *GOBI** has a lower objective value (16.4%) and prediction error of objective value (14.2%) for the next interval. This clearly shows that *GOBI** can optimize the objective value better than *GOBI*. This is because *GOBI* predicts the objective value for the interval I_t using only the utilization metrics of interval I_{t-1} . However, *GOBI** has more information as the LSTM models predict utilization metrics for I_t and the *GOBI* model gives a tentative action, which is then used to simulate I_t and get a reasonable estimate of the objective value for I_t . This estimate with the utilization metrics of I_{t-1} aid *GOBI** to have much closer predictions and hence better scheduling decisions. However, the scheduling time of *GOBI** is more than twice that of *GOBI*. This makes the *GOBI* approach more appropriate for environments with resource constrained fog brokers.

3.6 Summary

We have presented a coupled-simulation approach to leverage simulators to predict the QoS parameters and make better decisions in a heterogeneous fog setup. The presented COSCO framework allows deployment of a holistic platform that provides an easy to use interface for scheduling policies to access simulation capabilities required to achieve the three objectives highlighted in Chapter 1. Moreover, we have presented two scheduling policies based on back-propagation of gradients with respect to input, namely *GOBI* and *GOBI**. These approaches use neural approximators to model objective scores for the scheduling decisions. *GOBI** uses *GOBI*, predictors of utilization characteristics and an underlying simulator to better predict the objective values leading to better decisions. Comparing *GOBI* and *GOBI** against state-of-the-art schedulers using real-world fog applications, we see that our methods are able to reduce energy consumption, response time, SLO violations and scheduling time. Between *GOBI* and *GOBI**, *GOBI** gives better QoS values; however, *GOBI* is more suitable for resource constrained servers. We shall utilize the ideas developed in this chapter to solve the objectives described in Chapter 1 and address the limitations mentioned in Chapter 2.

Chapter 4

AI Augmented Resource Provisioning

This chapter describes the novel resource provisioning approach, namely CILP, to achieve objective 1(a) as described in Chapter 1. We leverage the coupled simulation ideas developed in Chapter 3 to extend imitation learning models for optimal provisioning, specifically using a co-simulated oracle. Optimal provisioning enables us to optimize operational costs and QoS metrics in Fog Computing environments.

Challenges. Virtualization allows the deployment of independent virtual machines (VMs) on physical machines (PMs) [Armbrust et al., 2010]. However, this underpinning technology gives rise to the challenge of efficiently managing resources. This becomes crucial for public cloud providers that serve a large number of customers and even companies with private cloud infrastructures to curtail the operational costs of the cloud machines. Resource management is also paramount for users that execute workloads on cloud infrastructures with limited cost budgets [Gu et al., 2015]. A key aspect of resource management is VM provisioning, which instantiates and deallocates VMs based on dynamic workload demands. Most prior work aims at the automation of VM provisioning to optimize various performance measures such as energy consumption, operational cost and task response time [Levy et al., 2020, Al-Asaly et al., 2022]. As users shift to workloads that rely on the integration with Internet of Things (IoT) sensors and actuators, contemporary applications have become latency-critical [Schulz et al., 2017]. In order to make sure that resources are available as soon as possible, proactive VM provisioning is required to avoid degradation of system performance. This entails predicting the resource demands of workloads in a future state and allocating new or deallocating existing VMs for QoS optimization. This broad-level formulation has been widely used in prior work and is commonly referred to as *predictive VM provisioning (PreVMP)* [Luo et al., 2020, 2021]. However, this problem is challenging due to the non-stationary utilization characteristics of most workloads [Ebadifard and Babamir, 2021], requiring methods to dynamically adapt their provisioning policies to the changes in the environment.

Existing solutions. Several proactive VM provisioning methods have been proposed in the past. Since the optimization objectives, such as cost and response time, depend on *a-priori* unknown future utilization characteristics of workloads, the problem is challenging to solve using conventional optimization strategies. Thus, most state-of-the-art methods decompose the provisioning problem into first predicting the demands of running workloads and then optimizing the VM provisioning plan [Stuckey et al., 2020, Luo et al., 2020, Aliyu et al., 2020]. However, existing solutions tend to ignore crucial aspects such as provisioning overheads and heterogeneous VM costs in a cloud system. As VM provisioning entails the time-consuming creation of

VMs, this can affect the response time of the new workloads that are placed on new VMs by the underlying scheduler. Moreover, deallocating VMs, in an effort to possibly reduce operation costs, requires the workloads being executed in that VM to be preemptively migrated to other active VMs in the system, further increasing the response times. Higher response times could increase the fraction of the user-defined Service Level Agreement (SLA) violations. Further, cloud providers typically offer a diverse range of VM types with disparate costs and computational capacities. Choosing the optimal set of VMs is crucial to provide the best QoS in the cloud [Sohani and Jain, 2021, Zhang et al., 2014].

Research contributions. We propose a novel imitation learning framework that leverages a co-simulated digital-twin, *i.e.*, an approximate system model of the cloud infrastructure, to obviate the lack of overhead and cost metrics in decision optimization [Kaur et al., 2020]. We term this framework as Co-simulation based Imitation Learner (CILP). Our imitation learner is a composite neural network that predicts future workload demands and provisioning decisions by imitating a co-simulator based decision making oracle. This allows *CILP* to take optimal provisioning decisions, ameliorating the need for running costly simulations at test time and enabling *CILP* to efficiently scale with the size of the cloud infrastructure. Experimental evaluation with three public benchmarks demonstrates that *CILP* gives up to 22% higher resource utilization, 14% higher QoS scores, and 44% lower execution costs than the state-of-the-art techniques.

4.1 System Model and Problem Formulation

4.1.1 System Model

We assume a distributed cloud computing environment with multiple VMs that process a set of independent workloads. Unlike Chapter 3, we do not assume a fixed set of host nodes. We consider a discrete-time control problem, *i.e.*, we divide the timeline into fixed size execution intervals (of Δ time) and denote the t -th interval by I_t . We consider a bounded execution with T intervals; thus, $t \in \{1, \dots, T\}$. At I_t , the set of VM hosts is denoted by $\mathcal{H}_t$ and the set of workloads by $\mathcal{W}_t$. Each workload $w_t^j \in \mathcal{W}_t$ is characterized by its CPU utilization in terms of the number of instructions per second (IPS), denoted by c_t^j ; RAM utilization in GBs, denoted by r_t^j ; and disk storage utilization in GBs, denoted by s_t^j . Here, $j \in \{1, \dots, |\mathcal{W}_t|\}$. The feature vector for workload w_t^j is denoted by $W_t^j = [c_t^j, r_t^j, s_t^j]$. The collection of feature vectors of all workloads in I_t is denoted by W_t . In I_t , we consider a workload allocation, also referred to as a schedule, for each host. The set of workloads allocated to host $h_t^i \in \mathcal{H}_t$ is denoted by $\mathcal{A}_t^i \subseteq \mathcal{W}_t$, where $i \in \{1, \dots, |\mathcal{H}_t|\}$. Similar to workloads, for each host $h_t^i \in \mathcal{H}_t$, the feature vector includes cumulative utilization and maximum capacity of resources (CPU, RAM and disk) and is denoted by $H_t^i = [\sum_{w_t^j \in \mathcal{A}_t^i} W_t^j, \bar{c}^i, \bar{r}^i, \bar{s}^i]$, where $\sum$ denotes vector sum and $\bar{c}^i, \bar{r}^i, \bar{s}^i$ denote IPS, RAM and disk storage capacities of the host. Each host has a VM type that corresponds to a distinct set of utilization capacities and execution costs in a public cloud deployment. We consider a static set of VM types $\mathcal{V}$, where the type of host h_t^i is denoted by $V_t^i = (\bar{c}^i, \bar{r}^i, \bar{s}^i, \mu^i, \xi^i)$. Here, μ^i is the cost per unit time and ξ^i is the distribution of the provisioning time of a VM instance for a VM type V_t^i . We assume μ^i and ξ^i to be stationary with time $\forall i$. As elements of V_t^i are independent of time, these symbols are not sub-scripted with t . V_t denotes VM types for all hosts in I_t .

VM provisioning. VM provisioning is performed at the start of each interval. We denote the set of VM provisioning actions at the start of I_t by P_t , which is a collection of VM types to be provisioned with the

number of instances ($n_t \subseteq \mathcal{V} \times \mathbb{Z}$) and hosts to be deallocated from the system ($d_t \subseteq \mathcal{H}_{t-1}$). The new set $\mathcal{H}_t$ is the set of hosts in the previous interval $\mathcal{H}_{t-1}$ union the provisioned hosts in n_t minus the deallocated hosts d_t . Deallocation of hosts in the system entails migrating workloads executing in those hosts to other preexisting active or newly provisioned hosts in the system. VM provisioning also entails prediction of the workload utilization characteristics for I_t , *i.e.*, W_t . We denote this prediction by $\hat{W}_t$, for which the provisioner may use historic data $\{W_k | 1 \leq k < t\}$. We denote the provisioner by f_{θ}^{prov} that uses a scheduling decision D_t and workload utilization metrics of the previous interval W_{t-1} to predict $\hat{W}_t$ and P_t . As we use a neural network in our model, we use θ to denote the weights of such a network. Thus, $\hat{W}_t, P_t = f_{\theta}^{prov}(D_t, W_{t-1})$. We now formulate the underlying scheduler that generates D_t .

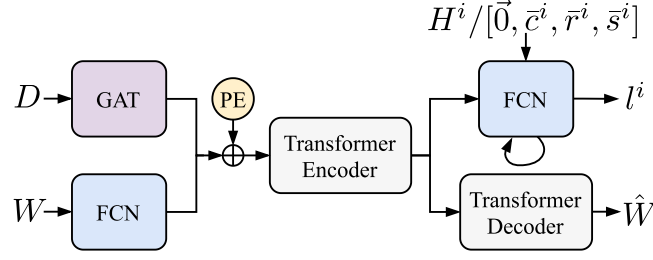
Underlying scheduler. We consider the presence of a scheduler f^{sched} that predicts a schedule D_t using the feature vectors of workloads and hosts and a provisioning decision P_t . However, the set of workloads $\mathcal{W}_{t-1}$ and hosts $\mathcal{H}_{t-1}$ in I_{t-1} might change in the next interval I_t . Thus, the scheduler utilizes $\hat{W}_t$ for existing workloads, $\vec{0}$ for new workloads, $[\vec{0}, \vec{c}^i, \vec{r}^i, \vec{s}^i]$ for new hosts and drops the feature vectors of workloads that complete execution or hosts that are deallocated at the end of I_{t-1} by the provisioner. If P_t is empty, *i.e.*, it has no provisions or deallocations, then we can use D_t directly to schedule tasks. However, for a non-empty decision P_t , the scheduler also needs to decide where to migrate tasks running in hosts that need to be deallocated. The schedule that also includes these migration decisions is denoted by $\hat{D}_t$. Thus, with the described modification, the schedule for I_t is evaluated as $\hat{D}_t = f^{sched}(H_{t-1}, \hat{W}_t, P_t)$. The schedule D_t is a bipartite graph with edges from $\mathcal{W}_t$ to $\mathcal{H}_t$ corresponding to the placement of workloads on hosts as in prior work [Tuli et al., 2021]. The set $\mathcal{A}_t^i$, described previously, is inferred from D_t . The graph nodes are initialized with embeddings corresponding to the feature vectors of workloads and hosts in the system. This graphical modeling of the schedule enables us to scale our neural models with the number of workloads and hosts in the cloud environment.

4.1.2 Co-Simulator Based Predictions

We also utilize a co-simulator as in Chapter 3, which stores the time-series utilization characteristics W_t and H_t . It executes a simulation of the cloud model for a given workload features W_{t-1} , VM types V_t , provisioning decision P_t and scheduling decision D_t to generate QoS metrics for interval I_t . These metrics include energy consumption, the response time of completed workloads, SLA violation rates, execution cost and utilization ratio. It also provisions new VMs and deallocates existing ones as per P_t . We denote the co-simulator by f^{sim} and the set of QoS metrics by $\mathcal{Q}_t = f^{sim}(W_{t-1}, V_t, P_t, \hat{D}_t)$.

4.1.3 Problem Formulation

At the start of the interval I_t , given a set of workloads $\mathcal{W}_t$, VM types $\mathcal{V}$, active hosts $\mathcal{H}_{t-1}$ we define our problem as to find a feasible provisioning decision P_t that maximizes the CPU core utilization of hosts to avoid system under-utilization or resource wastage in private clouds. However, this may not capture the heterogeneous pricing policy adopted by cloud providers; thus, we also need to minimize the execution cost for the end user for public cloud deployments. A combination of these two may be required for hybrid cloud environments. We denote these QoS metrics for I_t in our formulation by r_t (utilization ratio) and ϕ_t (cost). We define these two

Figure 4.1: *CILP* neural model.

metrics as,

$$r_t = \frac{\sum_{w_t^j \in \mathcal{W}_t} c_t^j}{\sum_{h_t^i \in \mathcal{H}_t} \bar{c}_t^i} \text{ and } \phi_t = \sum_{h_t^i \in \mathcal{H}_t} \mu^i \cdot \Delta. \quad (4.1)$$

The utilization ratio is an important metric that translates to the effective usage efficiency of a cloud system and is a standard metric in prior work [Luo et al., 2020, 2021]. A higher utilization ratio typically corresponds to lower energy consumption amortized over the number of completed tasks, and thus directly reflects the QoS of the system. Similarly, cost is a crucial metric for both cloud providers and users to reduce the financial footprint of workload execution. These metrics also appear in prior work [Luo et al., 2021, Gu et al., 2015] and are easy to compute in our formulation, assuming our VM provisioner predicts c_t^j s in equation (4.1). The formal optimization program is described as follows.

$$\begin{aligned} & \underset{\theta}{\text{maximize}} && \sum_{t=1}^T r_t - \gamma \cdot \phi_t \\ & \text{subject to} && \forall t, \forall w_t^j \in \mathcal{W}_t, \sum_{w_t^j \in \mathcal{A}_t^i} W_t^j \leq [\bar{c}^i, \bar{r}^i, \bar{s}^i] \\ & && \forall t, \hat{W}_t, P_t = f_{\theta}^{prov}(D_t, W_{t-1}) \\ & && \forall t, D_t = f^{sched}(H_{t-1}, \hat{W}_t, P_t) \\ & && \forall t, r_t, \phi_t \in f^{sim}(W_{t-1}, V_t, P_t, \hat{D}_t) \end{aligned} \quad (4.2)$$

for a given co-simulator f^{sim} and scheduler f^{sched} and $t = \{1, \dots, T\}$. Here, γ is a weighting parameter that can be set by a user based on the deployment scenario.

Even with known workload characteristics W_t at the start of I_t , the problem is known to be NP-hard [Zhao et al., 2018].

4.2 The CILP Provisioner

CILP learns to predict *a-priori* unknown workload characteristics of the next interval and provisioning decisions and is realized as a neural model f_{θ}^{prov} .

4.2.1 Neural Model

For interval I_t , the inputs of the neural model are schedules D_t and workload utilizations of the previous interval W_{t-1} . To predict $\hat{W}_t$ and P_t , we use a composite neural model by inferring the decision using a graph attention network (GAT), the utilization characteristics using a feed-forward network (also referred to as a fully-connected

network or FCN) and a Transformer to capture the temporal trends in the data (see Figure 4.1 for an overview). Representing the scheduling decision as a graph allows the model to consider the other tasks on a particular host while allocating a particular task on the same host. To infer through such a representation, we use attention based graph-convolution operations provided by a GAT network. The utilization characteristics may or may not have inter-task dependencies, hence we use them as tensor inputs to an FCN.

For notational convenience, while discussing the neural model, we drop the subscript t without loss in generality. The schedules are encoded as bipartite graph with nodes $\mathcal{N} = \mathcal{W} \cup \mathcal{H}$ and relations $(w^j, h^i) \in D$. Each workload node w^j and host node h^i gets the embeddings W^j and H^i respectively. For a generic node $n \in \mathcal{N}$, we denote its embedding by e_n and its set of neighbors by $\mathcal{S}_n$. Following [Veličković et al., 2018], we perform graph attention convolution as

$$\begin{aligned}\alpha_n &= \text{softmax}_{k \in \mathcal{S}_n}(\text{LeakyReLU}(W_g^0 e_k + b_g^0)), \\ e_n &= \sigma\left(\sum_{k \in \mathcal{S}_n} \alpha_n \cdot (W_g^1 e_k + b_g^1)\right),\end{aligned}\tag{4.3}$$

where α_n are the attention weights for node n . GAT enables the method to efficiently scale with the number of workloads and hosts in the system. The stacked representation for all nodes (e_n) is represented as E^G . We also pass the normalized input W of the previous interval through a feed-forward network with sigmoid activation

$$E^W = \sigma(\text{FeedForward}(W)).\tag{4.4}$$

We then pass the concatenated vector $E = [E^G, E^W]$ through a Transformer encoder after adding positional encodings (PE) [Vaswani et al., 2017] using multi-head self-attention, giving an encoded representation,

$$E^0 = \text{TransformerEncoder}(E).\tag{4.5}$$

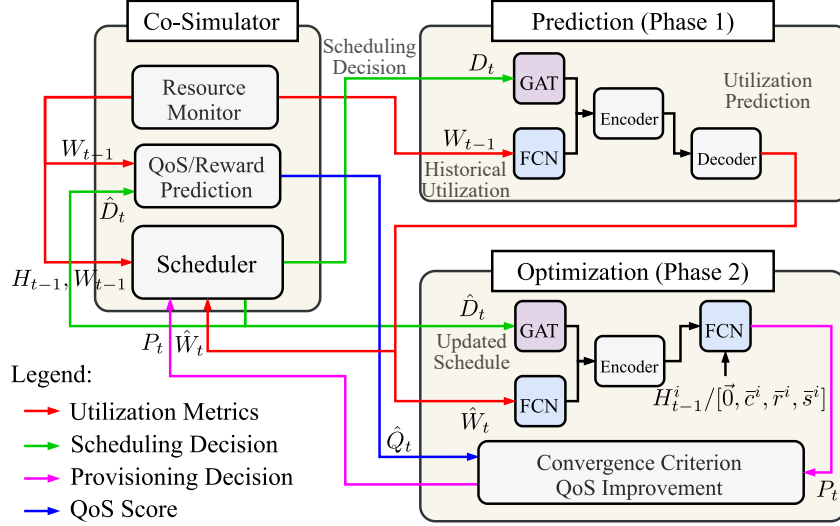
Early fusion of E^W and E^G enables the downstream predictors to exploit them together. Using E^0 , we then predict an estimate of utilization characteristics of the next interval, *i.e.* $\hat{W}$, we use a Transformer decoder as

$$\hat{W} = \text{TransformerDecoder}(E^0).\tag{4.6}$$

To generate a provisioning action, we predict the likelihood score for each action independently using a feed-forward network. To allow the model to scale with $|\mathcal{H}|$, for each active host in $\mathcal{H}$, we use the feature vectors from the previous interval H^i and for each new VM type we use the vector $[\vec{0}, \vec{c}^i, \vec{r}^i, \vec{s}^i]$ to infer on which host should be deallocated or provisioned. The likelihood score for each such provisioning action, denoted by p^i with feature vector F^i such that $i \in \{1, \dots, |\mathcal{H}| + |\mathcal{V}|\}$, is represented as l^i and calculated as

$$l^i = \sigma(\text{FeedForward}(E^0, F)).\tag{4.7}$$

This factored-style prediction of the likelihood score for each provisioning action enables our model to be agnostic to the number of hosts in the setup. The final provisioning action, denoted as p , becomes the action corresponding to the highest likelihood score. These likelihood scores give us a single provisioning action.

Figure 4.2: Top level design of the interleaved co-simulation and prediction in *CILP*.

However, a provisioning decision is a collection of multiple such actions. Moreover, we want to generate likelihood scores using the predicted utilization metrics $\hat{W}$. To do this, we run inference in two phases.

4.2.2 Two-phase inference

An overview of the novel two-phase interleaved co-simulation and prediction in *CILP* is presented in Fig. 4.2. We initialize an empty provisioning decision $P_t = \emptyset$. In the *first phase*, for interval I_t , using schedule D_t and workload utilization characteristics W_{t-1} , we predict $\hat{W}_t$ that estimates the workload demands in I_t ; thus $\hat{W}_t \leftarrow f_{\theta}^{prov}(D_t, W_{t-1})$. We get $\hat{W}_t$ as the output of the first phase. These prediction estimates of the workload utilization characteristics using the historical values enable the model to make informed decisions for an estimated future system state. Initially, we start without any provisioning; hence, the scheduling decision is obtained as $D_t = f^{sched}(H_{t-1}, W_{t-1}, \emptyset)$. In the *second phase*, we utilize the predicted $\hat{W}_t$ to leverage estimated workload demands in the next interval. Again, we initially keep P_t as an empty set. This gives $\hat{D}_t = f^{sched}(H_{t-1}, \hat{W}_t, \emptyset)$ as the initial schedule. Using $\hat{D}_t$ and $\hat{W}_t$, we evaluate the likelihood scores for each provisioning action $l_t^i = f^{prov}(\hat{D}_t, \hat{W}_t)$. The decided action p is the one with the highest likelihood score. We add this action to P_t and iteratively run the following, updating P_t at each step and evaluating a QoS estimate score, denoted by $\hat{Q}_t$. Thus,

$$\begin{aligned}
 \hat{D}_t &\leftarrow f^{sched}(H_{t-1}, \hat{W}_t, P_t), \\
 l_t^i &\leftarrow f^{prov}(\hat{D}_t, \hat{W}_t), \\
 p &\leftarrow p^{\arg \max_i l_t^i}, \\
 P_t &\leftarrow P_t \cup \{p\}, \\
 \hat{Q}_t &\leftarrow f^{sim}(W_{t-1}, V_t, P_t, \hat{D}_t).
 \end{aligned} \tag{4.8}$$

Note that the scheduler also decides the preemptive migrations in $\hat{D}_t$ for every non-empty P_t that has host deallocation actions. We continue the above until $\hat{Q}_t$ is non-decreasing. This auto-regressive style of action prediction enables the model to remain parsimonious in terms of the provisioning decisions and avoid excessive

Algorithm 4.1 CILP Provisioner**Require:**

```

  Pretrained model  $f_{\theta}^{prov}$ 
  scheduler  $f^{sched}$ 
  co-simulator  $f^{sim}$ 
  Initialize  $W_{-1} \leftarrow \vec{0}$ 
1: for  $t \in \{1, \dots, T\}$ 
2:    $\hat{W}_t \leftarrow f_{\theta}^{prov}(D_t, W_{t-1})$  ▷ Predict
3:    $P_t \leftarrow \emptyset$  ▷ Initialize Decision
4:    $\hat{D}_t \leftarrow f^{sched}(H_{t-1}, W_{t-1}, \emptyset)$ 
5:    $l_t^i \leftarrow f^{prov}(\hat{D}_t, \hat{W}_t)$ 
6:    $p \leftarrow p^{\arg \max_i l_i}$ 
7:   while  $\hat{Q}_t$  non-decreasing
8:     Update  $P_t, \hat{Q}_t$  using equation (4.8) ▷ Optimize
9:   Execute  $P_t$  and  $\hat{D}_t$ 

```

overheads. Only those actions are performed that lead to an increase in the expected QoS of the system. Further, the interleaving of action prediction and co-simulation enables us to train an imitation learner, ameliorating the need for costly simulations at test time. Parameter sharing between the demand and likelihood prediction reduces the size of the parameter set, enables *CILP* to jointly learn temporal trends and gain training stability. The converged decision P_t is used for VM provisioning at the start of the interval I_t .

4.2.3 Model Training

We train the *CILP* model using an imitation learning setup where the teacher is the co-simulator, acting as an oracle that generates ground truth actions [Sun et al., 2017]. In the second phase, for each input pair $(\hat{D}_t, \hat{W}_t)$, the model generates likelihood score l_t^i for action p^i . We also co-simulate the provisioning action p^i , generate scheduling decision $\hat{D}_t$ and a reward parameter $\hat{R}_t$ as

$$\begin{aligned}
 \hat{D}_t &\leftarrow f^{sched}(H_{t-1}, \hat{W}_t, \{p^i\}), \\
 r_t^0, \phi_t^0 &\leftarrow f^{sim}(W_{t-1}, V_t, \emptyset, D_t), \\
 r_t^1, \phi_t^1 &\leftarrow f^{sim}(W_{t-1}, V_t, \{p^i\}, \hat{D}_t), \\
 \hat{R}_t^k &\leftarrow r_t^k - \gamma \cdot \phi_t^k, k \in \{0, 1\},
 \end{aligned} \tag{4.9}$$

where r_t and ϕ_t are defined as per (4.1) and superscripts correspond to whether p^i is executed or not (1 if it is). The reward parameter is the objective function in the program (4.2). The ground truth label then becomes $g_t^i = \arg \max_k R_t^k$, which signifies whether p^i improves the reward parameter. Now, for each p^i we use the binary cross-entropy loss

$$L_{BCE}(g_t^i, l_t^i) = -\frac{1}{2}(g^i \cdot \log(p^i) + (1 - g^i) \cdot \log(1 - p^i)). \tag{4.10}$$

We also utilize the mean-square-error between the predicted demands and those from the dataset in the next interval

$$L_{MSE}(\hat{W}_t, W_t) = \frac{1}{|W_t|} \|\hat{W}_t - W_t\|^2. \tag{4.11}$$

Table 4.1: Comparing utilization ratio r_t , cost ϕ_t and QoS score $\hat{Q}_t$ for each interval across all models.

Model	Azure2017			Azure2019			Bitbrain		
	r	cost	QoS	r	cost	QoS	r	cost	QoS
<i>ARIMA+ACO</i>	0.385 ± 0.103	0.481 ± 0.060	0.789 ± 0.008	0.581 ± 0.105	0.491 ± 0.056	0.740 ± 0.014	0.374 ± 0.103	0.505 ± 0.062	0.749 ± 0.024
<i>LSTM+ACO</i>	0.411 ± 0.114	0.508 ± 0.065	0.786 ± 0.008	0.389 ± 0.114	0.519 ± 0.059	0.756 ± 0.020	0.401 ± 0.080	0.538 ± 0.054	0.759 ± 0.018
<i>Decision-NN</i>	0.595 ± 0.113	0.701 ± 0.022	0.738 ± 0.006	0.732 ± 0.068	0.681 ± 0.024	0.736 ± 0.007	0.697 ± 0.081	0.647 ± 0.054	0.714 ± 0.021
<i>Semi-Direct</i>	0.655 ± 0.065	0.679 ± 0.017	0.736 ± 0.005	0.868 ± 0.059	0.746 ± 0.057	0.701 ± 0.006	0.733 ± 0.069	0.736 ± 0.063	0.711 ± 0.018
<i>UAHS</i>	0.753 ± 0.086	0.738 ± 0.045	0.715 ± 0.004	0.809 ± 0.060	0.701 ± 0.028	0.746 ± 0.008	0.668 ± 0.098	0.775 ± 0.038	0.696 ± 0.009
<i>Narya</i>	0.773 ± 0.068	0.607 ± 0.051	0.771 ± 0.003	0.740 ± 0.073	0.624 ± 0.032	0.727 ± 0.006	0.696 ± 0.079	0.579 ± 0.051	0.727 ± 0.018
<i>CAHS</i>	0.800 ± 0.073	0.668 ± 0.023	0.738 ± 0.007	0.753 ± 0.065	0.779 ± 0.043	0.698 ± 0.013	0.660 ± 0.058	0.655 ± 0.040	0.721 ± 0.015
<i>CILP_IL</i>	0.608 ± 0.083	0.495 ± 0.025	0.780 ± 0.004	0.736 ± 0.060	0.506 ± 0.019	0.763 ± 0.007	0.617 ± 0.106	0.492 ± 0.044	0.736 ± 0.018
<i>CILP_Trans</i>	0.843 ± 0.051	0.477 ± 0.016	0.796 ± 0.005	0.817 ± 0.046	0.549 ± 0.020	0.781 ± 0.010	0.712 ± 0.056	0.441 ± 0.028	0.746 ± 0.025
<i>CILP</i>	0.857 ± 0.049	0.429 ± 0.020	0.839 ± 0.004	0.893 ± 0.051	0.438 ± 0.036	0.796 ± 0.011	0.806 ± 0.051	0.420 ± 0.029	0.803 ± 0.020

Thus, the model training loss for a given utilization trace dataset at each interval t is evaluated as

$$L = L_{MSE}(\hat{W}_t, W_t) + \sum_{h_t^i \in \mathcal{H}_t} L_{BCE}(g_t^i, l_t^i). \quad (4.12)$$

Unlike prior work, while calculating the utilization ratio r_t , our co-simulator considers the migration delay and uses an average resource utilization value of workloads over the execution interval I_t . This makes our optimization objective resemble more closely to real systems.

4.2.4 Provisioning in Practice

Using a pre-trained neural model, the *CILP* provisioner is illustrated in Algorithm 4.1. In each interval, we first predict the workload demands in the next interval (line 2), initialize decision (line 3) and optimize the preemptive migration decision iteratively as described earlier in equation (4.8) (line 8). The final provisioning decision P_t and schedule $\hat{D}_t$ are then executed in the cloud environment. As shown in the algorithm, each iteration in the *CILP* training process includes a loop as in equation (4.8) that updates the provisioning decision each time. As a decision could include provisioning a new host, that is one of the VM types $\mathcal{V}$. A decision could also deallocate a VM from the existing set of active hosts in the system, *i.e.*, $\mathcal{H}$. Thus, in the worst case, we run $|\mathcal{H}| + |\mathcal{V}|$ iterations of the inner loop.

4.3 Performance Evaluation

4.3.1 Setup

Datasets. In order to evaluate the performance of *CILP*, we utilize three public datasets: **Azure2017**, **Azure2019** and **Bitbrain**. The first two are collected from Microsoft Azure public cloud platform and are representative workload traces across thirty consecutive days [Cortez et al., 2017]. The **Azure2017** was generated using 110 cloud VMs, whereas the **Azure2019** using 150 VMs, both across 30 consecutive days. The final dataset, **Bitbrain** consists of traces of resource utilization metrics from 1750 VMs running on BitBrain distributed datacenter [Shen et al., 2015]. The workloads running on these servers are from various industry applications, showing that these datasets capture real-world workloads. We utilize these traces as utilization characteristics of workloads and generate workloads using the same distribution as done in the traces. As all datasets use an interval duration of five minutes, we set the same value as Δ in our experiments *i.e.* the interval duration in our formulation (see Section 4.1).

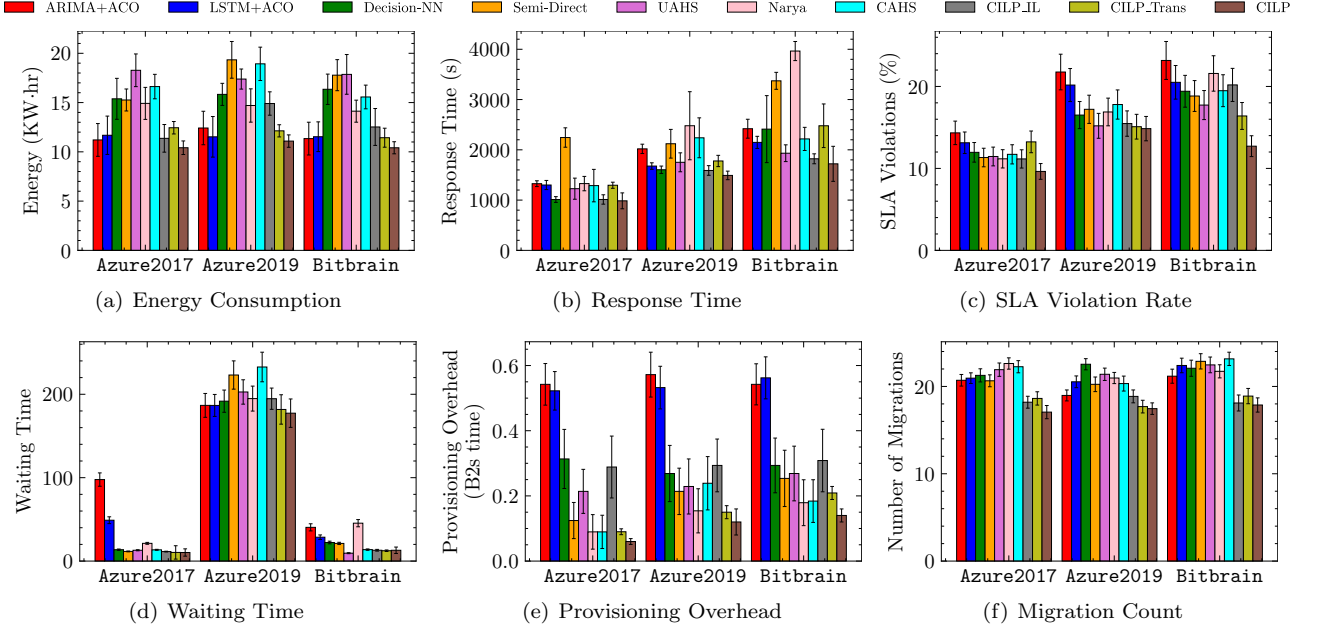


Figure 4.3: Comparison of QoS parameters (averaged over intervals) of *CILP* against baselines and ablated models.

Baselines. We compare *CILP* against 7 baselines. We integrate the *ACO* algorithm with two demand forecasting methods: AutoARIMA and LSTM, and call these *AutoARIMA+ACO* and *LSTM+ACO*. We also include classical predict+optimize methods *Decision-NN* and *Semi-Direct*. Finally, the state-of-the-art baselines are *UAHS*, *Narya* and *CAHS* (see Chapter 2).

4.3.2 Testbed

Cloud Environment. We perform our experiments on a Microsoft Azure platform using the Pre-Provisioning Service (PPS) to run our methods, with workloads as *Docker* containers having utilization characteristics like those of our dataset traces. We use diverse VM types in our cloud infrastructure, *i.e.*, *B2s* with a dual-core CPU and 4GB RAM, *B4ms* with a quad-core CPU and 16GB RAM and *B8ms* with an octa-core CPU and 32 GB RAM. All methods may provision up to 200 VMs in our testbed. The ξ^i variables of these VM types, described in Section 4.1, are set using Gaussian regression based on 100 datapoints corresponding to the provisioning time for each type. The costs μ^i are taken from Azure pricing calculator¹ for the East-US Azure datacenter [Copeland et al., 2015]. The power consumption values of increments of 10% CPU utilization of Azure VM types are taken from Standard Performance Evaluation Corporation benchmark repository².

Metrics and Implementation. Our QoS score ($\hat{Q}_t$) is a convex combination of three QoS metrics obtained from the co-simulator, *i.e.*, normalized energy consumption (q_t^e), average response time (q_t^r) of completed workloads and SLA violation fraction (q_t^{sla}). Thus,

$$\hat{Q}_t \leftarrow 1 - (\alpha \cdot q_t^e + \beta \cdot q_t^r + \delta \cdot q_t^{sla}), \quad (4.13)$$

where α, β, δ are convex-combination weights set as per user requirements. For our experiments, we set them

¹<https://azure.microsoft.com/en-us/pricing/calculator/>.

²https://www.spec.org/cloud_iaas2018/results/

to $1/3$ as per prior work [Tuli et al., 2021]. The co-simulator f^{sim} obtains energy estimates using simulated power models; it adds migration time and waiting time for provisioned hosts (using ξ^i) to the response times of affected workloads. The final response time values are used to decide the violations of SLA deadlines. To implement *CILP*, we build upon the co-simulation primitives provided by the COSCO framework (see Chapter 3) by modifying and integrating with custom resource provisioning methods. We use the Gradient Optimization using Backpropagation to Input (GOBI) scheduler as our underlying scheduling model and predefined SLA deadlines in COSCO. Code submitted as supplementary material.

Experimental Details. The *CILP* and baseline methods run on a separate server with this configuration: Intel i7-10700K CPU, 64GB RAM, Nvidia RTX 3080 and Windows 11 OS. We run for $T = 200$ intervals. We set the user-defined parameter γ using grid-search over the average QoS score $\frac{1}{T} \sum_{t=1}^T \hat{Q}_t$ generated from the co-simulator (see sensitivity analysis in Section 4.3.3).

Training Details Model training used a learning rate of 5×10^{-4} . The Adam optimizer [Kingma and Ba, 2014] with a weight decay parameter of 10^{-5} and a batch size of 64 was used. To train the neural model, we randomly divide the training time series into 80% training data and 20% validation data. An early stopping criterion was applied for convergence using the loss of the validation data.

4.3.3 Results and Ablation Analysis

Comparison with baselines. Table 4.1 presents the utilization ratio, cost in USD and QoS score ($\hat{Q}_t$) averaged over the execution intervals. Figure 4.3 shows the individual QoS metrics for each method. For all datasets, *CILP* outperforms the baselines in terms of average utilization ratio, execution cost and QoS scores. Compared to state-of-the-art baselines, *i.e.*, *UAHS*, *Narya* and *CAHS*, *CILP* gives up to 13.81%-22.12% higher values of average utilization ratio, 41.87%-45.80% lower average execution cost and 14.04%-17.34% higher average QoS score. When comparing individual QoS metrics (see Figure 4.3), *CILP* gives the least average energy consumption of 10.41 KW-hr, 7.78% lower than *ARIMA+ACO* with lowest energy consumption across all baselines. This is due to the cost minimization in *CILP* that ensures the least number of hosts are active in the system, leading to a lower energy footprint. *CILP* also gives the lowest average response time, 7.41% lower than the best baseline *Decision-NN* that leads to the lowest SLA violation rates that are up to 28.32% lower than the best baseline *UAHS*. This is due to the QoS augmented decision strategy in the *CILP* methodology. Additionally, the improvements in response times come directly from proactive provisioning in *CILP*, which minimizes the time tasks spent waiting to be scheduled, as shown in Figure 4.3(d). Finally, *CILP* also gives the least overheads in terms of provisioning delays and causes the minimum number of migrations, thanks to its co-simulation driven stopping criterion in Algorithm 4.1.

Ablation analysis. To test the efficacy of the co-simulated imitation learning and Transformer based neural models in *CILP*, we modify the approach as follows. First, we consider a model without co-simulated reward scores $\hat{R}_t$, but instead evaluate equation (4.1) without migration and provisioning overheads to train the neural model. We call this the *CILP_IL* model. Second, we replace the Transformer with a feed-forward network to test the importance of temporal trends that the Transformer captures. We call this the *CILP_Trans* model. The results in Table 4.1 and Figure 4.3 show a drop in all performance metrics for these models when compared to *CILP*, demonstrating the effectiveness of the Transformer based neural architecture and model training using co-simulated imitation learning.

γ	Azure2017			Azure2019			Bitbrain		
	r	cost	QoS	r	cost	QoS	r	cost	QoS
0.0	0.862	0.463	0.819	0.901	0.488	0.766	0.814	0.451	0.795
0.1	0.861	0.462	0.821	0.900	0.479	0.771	0.814	0.444	0.797
0.2	0.860	0.458	0.827	0.899	0.468	0.779	0.811	0.437	0.798
0.3	0.858	0.447	0.830	0.896	0.452	0.785	0.809	0.429	0.801
0.4	0.857	0.429	0.839	0.894	0.447	0.790	0.806	0.420	0.803
0.5	0.854	0.420	0.838	0.893	0.438	0.796	0.802	0.411	0.802
0.6	0.853	0.418	0.835	0.892	0.431	0.784	0.801	0.408	0.796
0.7	0.853	0.411	0.831	0.891	0.424	0.780	0.800	0.406	0.794
0.8	0.851	0.409	0.829	0.889	0.419	0.780	0.799	0.405	0.793
0.9	0.850	0.408	0.829	0.889	0.411	0.776	0.797	0.401	0.791
1.0	0.845	0.404	0.818	0.883	0.408	0.775	0.795	0.399	0.788

Table 4.2: Sensitivity analysis with the weight of the execution cost in the objective function (γ). All standard deviations are <0.05 .

Sensitivity with γ . Table 4.2 shows the performance of the *CILP* method for different values of user-defined hyper-parameter γ . Low γ corresponds to private-cloud deployments that do not have any monetary running costs, but require maximization of utilization ratio to reduce resource wastage. High γ corresponds to workload execution on public clouds with limited cost budgets, where the user aims at optimizing both system utilization and running costs. For comparison with baselines, we use $\gamma = 0.4, 0.5, 0.4$ for the three datasets, as per grid-search, while maximizing QoS scores. The *CILP* provisioner gives higher QoS scores than each baseline for every value of γ . These results provide clear evidence of the robustness of the *CILP* to different deployment scenarios and hyper-parameter settings.

The results demonstrate that *CILP* outperforms baselines not only in terms of QoS parameters, but also gives lower overheads. Akin to a *CILP* style imitation learner, our adapted version of *Narya* is the only baseline that learns to predict provisioning actions directly (using MAB) instead of running online optimization of the provisioning decision at run-time (such as other baselines *LSTM+ACO*, *Decision-NN*, *SemiDirect*, etc.). Unlike online-optimization based baselines, the adapted *Narya* model gives lower provisioning overheads, as seen in Figure 4.3(e). This corroborates that *CILP* outperforms even offline learners such as *Narya* and as well as online optimization based provisioning methods.

4.4 Summary

This chapter proposes *CILP*, a co-simulated digital-twin based imitation learner that dynamically predicts VM provisioning decisions to optimize the QoS of a cloud computing environment. It uses the ideas of co-simulation developed in Chapter 3. Unlike the GOBI model introduced in Chapter 3, we demonstrate that co-simulators can also be utilized to train models that directly predict resource management decisions in lieu of QoS parameters. It uses a Transformer based composite neural network to first predict the workload demands in the near future and auto-regressively find the optimal set of provisioning actions. Imitation learning enables us to train the action predictor in order to imitate the QoS effects of each action using a co-simulator. Such simulations also facilitate accurate performance prediction for each action and avoid excessive provisioning overheads. Extensive experiments on three public datasets show that *CILP* outperforms the state-of-the-art in QoS metrics, including energy consumption, resource utilization, execution cost and SLA violation rates. This gives us a state-of-the-art resource provisioner, achieving objective 1(a) as mentioned in Chapter 1. As part of

future work, we shall explore how *CILP* can be extended to not only make optimal provisioning decisions, but also workload scheduling decisions using the co-simulator.

Chapter 5

AI Augmented Application Deployment

This chapter describes the novel resource provisioning approach, namely SplitPlace, for optimal application deployment (objective 1(b)) as described in Chapter 1. In Chapter 4, we discussed how to optimally provision resources in a fog environment. Considering we have a provisioned set of resources, we now shift our focus on optimal deployment of AI based workloads. As described in Chapter 2, high accuracy neural models impose a high memory overhead and to accommodate these models in resource limited devices, we leverage the coupled simulation and gradient-based optimization ideas developed in Chapter 3 to extend data-driven bandit models for optimal application deployment. Optimal application deployment enables us to optimize response times and average accuracy of AI workloads in Fog Computing environments.

Challenges. In Fog systems, it is possible to handle the processing limitations of massive DNN models by effective preemption and prolonged job execution. However, memory bottlenecks are much harder to solve [Shao and Zhang, 2020]. In a distributed edge environment, storage spaces are typically mapped to network-attached-storage (NAS) media. Thus, prior work that runs inference on a pre-trained DNN without memory-aware optimizations leads to high network bandwidth overheads due to frequent overflow of memory and the use of virtual memory (swap space) on NAS media, making high fidelity inference using DNNs hard [Liu and Leon, 2020, Laskaridis et al., 2020]. To deploy an upgraded AI model, tech giants like Amazon, Netflix and Google usually consider completely revamping their infrastructures and upgrading their devices, raising many sustainability concerns [Gill et al., 2019]. This has made the integration of massive neural network models with such devices a challenging and expensive ordeal.

Motivation. To solve the above challenge, we use the idea of neural network splitting introduced in Chapter 2. Layer splitting is easier to deploy as pre-trained models can be just divided into multiple layer groups and distributed to different mobile edge nodes. However, layer splits require a semi-processed input to be forwarded to the subsequent edge node with the final processed output to be sent to the user, thus increasing the overall execution time. Moreover, layer-wise splitting gives higher accuracy compared to semantic splitting. Comparison of accuracies and average response times for the two strategies is shown in Figure 5.1. The figure shows results for 10 edge worker nodes using popular image classification datasets: MNIST, FashionMNIST and CIFAR100 [LeCun et al., 1998, Xiao et al., 2017, Krizhevsky et al., 2009] averaged over ResNet50-V2,

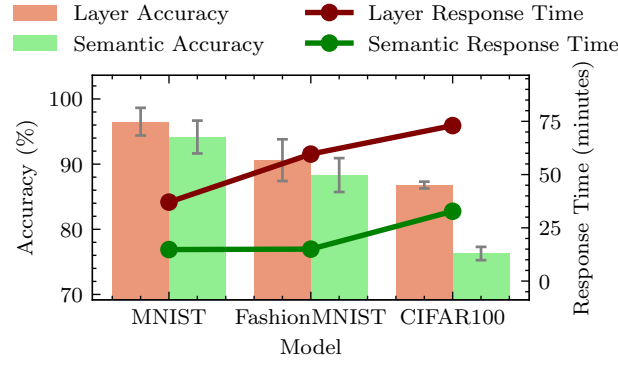


Figure 5.1: Comparison of Layer and Semantic Splits.

MobileNetV2 and InceptionV3 neural models [Gunasekaran et al., 2020]. As is apparent from the figure, layer splits provide higher accuracy and response time, whereas semantic splits provide lower values for both. We leverage this contrast in traits in Chapter 5 to trade-off between inference accuracy and response time based on SLA requirements of the input tasks. Despite the considerable drop in inference accuracy when using semantic splitting scheme, it is still used in the proposed SplitPlace approach (see Chapter 5) as it is better than model compression or early-exit strategies for quick inference. This is acceptable in many industrial applications [Goli et al., 2020, Huang et al., 2020] where latency and service level agreements are more important performance metrics than high-fidelity result delivery. In this chapter, we consider a system with both SLA violation rates and inference accuracy as optimization objectives. This makes the combination of layer and semantic splitting a promising choice for such use cases.

New Insights. A promising solution for this problem is the development of strategies that can accommodate large-scale DNNs within legacy infrastructures. However, many prior efforts in this regard have not yet tackled the challenge of providing a holistic strategy for not only distributed learning, but also inference in such memory-constrained environments [Kim et al., 2017, Shi et al., 2020, Lim et al., 2020]. Recently, research ideas have been proposed like Cloud-AI, Edge-AI and Federated learning that aim to solve the problem of running enormous deep learning models on constrained edge devices by splitting them into modular fragments [Shi et al., 2020, Lim et al., 2020]. However, in Cloud-AI where AI systems are deployed on cloud machines, the high communication latency leads to high average response times, making it unsuitable for latency-critical applications like healthcare, gaming and augmented reality [Kraemer et al., 2017, Siriwardhana et al., 2021]. Instead, Edge-AI provides low-latency service delivery, thanks to edge devices being in the same Local Area Network (LAN), where the input data from multiple edge nodes are combined to a single fixed broker node for processing. Edge-AI based methods aim at scheduling deep neural networks for providing predictable inference [Xiao et al., 2018, Gujarati et al., 2020]. However, due to the centralized collection of data, these solutions typically suffer from high bandwidth overheads and poor service quality [Shi et al., 2020]. Federated learning depends on data distribution over multiple nodes where the model training and inference are performed in a decentralized fashion. However, this paradigm assumes that neural models with data batches can be accommodated in the system memory. This is seldom the case for common edge devices like Arduinos or Raspberry Pis [Chen and Ran, 2019].

Existing Solutions. Recent works offer lower precision models that can fit within the limited memory of such devices by using methods like Model Compression or Model Pruning [Capotondi et al., 2020, Gunasekaran et al., 2020, Huang et al., 2020]. However, compressed and low-precision models lose inference accuracy, making them

unsuitable for accuracy-sensitive applications like security and intrusion detection [Le et al., 2020]. Recently, split neural network models have been proposed. They show that using semantic or layer-wise splitting, a large deep neural network can be fragmented into multiple smaller networks for dividing network parameters onto multiple nodes [Matsubara et al., 2019, Ushakov et al., 2018, Kim et al., 2017, Gordon and Crouson, 2008]. The former partitions a neural network into parallel disjoint models that produce a part of the result. The latter partitions a neural network into sequential models that generate intermediate results. We illustrate the accuracy and response time trade-offs through sample test cases in Section 2.2.3. Our experiments show that using layer and semantic splitting gives higher inference accuracies than previously proposed model compression techniques (see Section 5.3). However, no appropriate scheduling policies exist that can intelligently place such modular neural fragments on a distributed infrastructure to optimize both accuracy and SLA together. The placement of such split models is non-trivial considering the diverse and complex dynamism of task distribution, model usage frequencies and geographical placement of mobile edge devices [Ahmed and Rehmani, 2017].

Research Contributions. This chapter proposes a novel neural splitting and placement policy, *SplitPlace*, for enhanced distributed neural network inference at the edge. SplitPlace leverages a mobile edge computing platform to achieve low latency services. It allows modular neural models to be integrated for best result accuracies that could only be provided by cloud deployments. SplitPlace is the *first* splitting policy that dynamically decides between semantic and layer-wise splits to optimize both inference accuracy and the SLA violation rate. This decision is taken for each incoming task and remains unmodified until the execution of all split fragments of that task are complete. The idea behind the proposed splitting policy is to decide for each incoming task whether to use the semantic or layer-wise splitting strategy based on its SLA demands. Due to their quick adaptability, SplitPlace uses Multi-Armed-Bandits to model the decision strategy for each application type by checking if the SLA deadline is higher or lower than an estimate of the response time for a layer split decision [Bouneffouf et al., 2020]. Further, SplitPlace optimizes the placement decision of the modular neural network fragments using a split decision aware surrogate model. We present techniques to dynamically adapt to non-stationary workloads and mobile environments. We present a gradient-based optimization approach for task placement decision conditioned on split decisions. Experiments on real-world application workloads on a physical edge testbed show that the SplitPlace approach outperforms the baseline approaches by reducing the SLA violation rate and improving the average inference accuracy.

5.1 System Model and Problem Formulation

In this chapter, we assume a scenario with a fixed number of multiple heterogeneous edge nodes in a broker-worker fashion, which is a typical case in mobile-edge environments [Basu et al., 2019, Shao and Zhang, 2020, Matsubara et al., 2019]. Here, the broker node takes all resource management related decisions, such as neural network splitting and task placement. The processing of such tasks is carried out by the worker nodes. Examples of broker nodes include personal laptops, small-scale servers and low-end workstations. Example of common worker nodes in edge environments include Raspberry Pis, Arduino and similar System-on-Chip (SoC) computers [Gill et al., 2019]. All tasks are received from an IoT layer that includes sensors and actuators to collect data from the users and send it to the edge broker via the gateway devices. Akin to typical edge configurations, the edge broker then decides which splitting strategy to use and schedules these fragments to

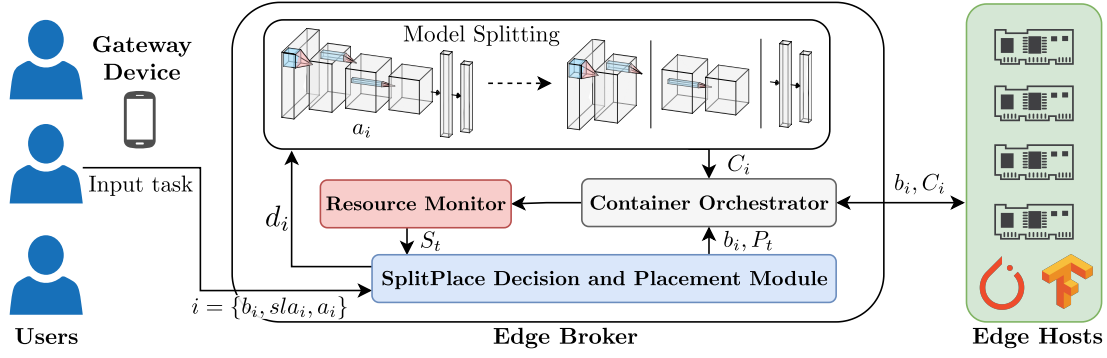


Figure 5.2: SplitPlace System Model

various edge nodes based on deployment constraints like sequential execution in a layer-decision. The data to be processed comes from the IoT sensors/actuators, which with the decision of which split fragment to use is forwarded by the broker to each worker node. Some worker nodes are assumed to be mobile, whereas others are considered to be fixed in terms of their geographical location. In our formulation, we consider mobility only in terms of the variations in terms of the network channels and do not consider the worker nodes or users crossing different networks. We assume that the CPU, RAM, Bandwidth and Disk capacities of all nodes are known in advance, and similarly the broker can sample the resource consumption for each task in the environment at any time (see *Resource Monitor* in Fig. 5.2). As we describe later, the broker periodically measure utilizations of CPU, RAM, Bandwidth and Disk for each task in the system. The broker is trusted with this information such that it can make informed resource management decisions to optimize QoS. Moreover, we consider that tasks include a batch of inputs that need to be processed by a DNN model. Further, for each task, a service level deadline is defined at the time the task is sent to the edge environment. We give an overview of the SplitPlace system model in Figure 5.2. We decompose the problem into deciding an optimal splitting strategy and a fragment placement for each application (motivation in Section 5.3.12 and more details in Section 5.1).

5.1.1 Workload Model

We consider a bounded discrete time control problem as discussed in Chapter 3. For deployment, we consider that $t \in \{0, \dots, \Gamma\}$, where $\Gamma + 1$ is the number of intervals in an execution. We assume a fixed number of worker machines in the edge layer and denote them as H . We also consider that new tasks created at the interval I_t are denoted as N_t , with all active tasks being denoted as A_t (and $N_t \subseteq A_t$). Each task $i \in A_t$ consists of a batch input b_i , SLA deadline sla_i and a DNN application a_i . The set of all possible DNN applications is denoted by $\mathcal{A}$. For each new task $i \in N_t$, the edge broker takes a decision d^i , such that $d^i \in \{L, S\}$, with L denoting layer-wise splitting and S denoting semantic split strategy. The collection of all split decisions for active tasks in interval I_t is denoted as $\mathcal{D}_t = \{d^i\}_{i \in N_t}$. Based on the decision d^i for task i , this task is realized as an execution workflow in the form of containers C^i . Similar to a VM, a container is a package of virtualized software that contains all of the necessary elements to run in any environment. The set of all containers active in the interval I_t is denoted as $C_t = \cup_{i \in A_t} C^i$. The set of all utilization metrics of CPU, RAM, Network Bandwidth and Disk for all containers and workers at the start of the interval I_t defines the state of the system, denoted as S_t .

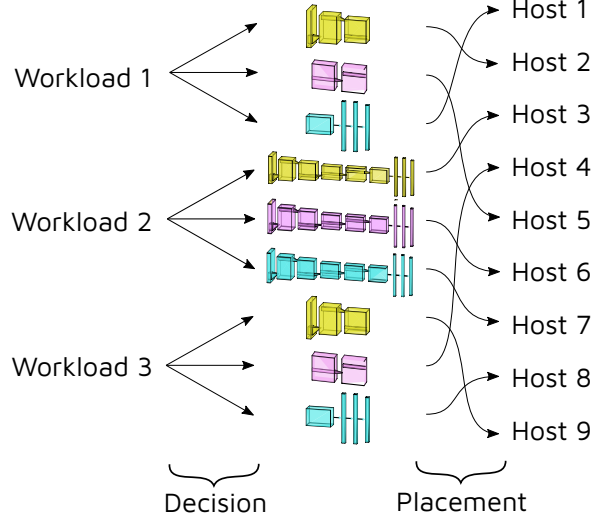


Figure 5.3: SplitPlace decision and placement problems.

5.1.2 Split Nets Placement

We partition the problem into two sub-problems of deciding the optimal splitting strategy for input tasks and that of placement of active containers in edge workers (see Figure 5.3). Considering the previously described system model, at the start of each interval I_t , the SplitPlace model takes the split decision d^i for all $i \in N_t$. Moreover, it also takes a placement decision for all active containers C_t , denoted as an adjacency matrix $P_t : C_t \times H$. This is realized as a container allocation for new tasks and migration for active tasks in the system.

The main idea behind the layer-wise split design is first to divide neural networks into multiple independent splits, classify these splits in preliminary, intermediate and final neural network layers and distribute them across different nodes based on the node capabilities and network hierarchy. This exploits the fact that communication across edge nodes in the LAN with few hop distances is very fast and has low latency and jitter [Park et al., 2016]. Moreover, techniques like knowledge distillation can be further utilized to enhance the accuracy of the results obtained by passing the input through these different classifiers. However, knowledge distillation needs to be applied at the training stage, before generating the neural network splits. As there are many inputs in our assumed large-scale deployment, the execution can be performed in a pipelined fashion to further improve throughput over and above the low response time of the nodes at the edge of the network. For the semantic split, we divide the network weights into a set or a hierarchy of multiple groups that use disjoint sets of features. This is done by making assignment decisions of network parameters to edge devices at deployment time. This produces a tree-structured network that involves no connection between branched sub-trees of semantically disparate class groups. Each sub-group is then allocated to an edge node. The input is either broadcasted from the broker or forwarded in a ring-topology to all nodes with the network split corresponding to the input task. We use standard layer [Yu et al., 2021] and semantic splitting [Kim et al., 2017] methods as discussed in Section 2.2.3.

We now outline the working of the proposed distributed deep learning architecture for edge computing environments. Figure 5.2 shows a schematic view of its working. As shown, there is a shared repository of neural network parameters which is distributed by the broker to multiple edge nodes. The layer and semantic splits are realized as Docker container images that are shared by the broker to the worker nodes at the start of each execution trace. The placement of tasks is realized as spinning up a Docker container using the corresponding

image on the worker. As the process of sharing container images is a one-time event, transferring all semantic and layer split fragments does not impose a high overhead on network bandwidth at runtime. This sharing of containers for each splitting strategy and dataset type are transferred to the worker nodes is performed at the start of the run. At run-time, only the decision of which split fragment to be used is communicated to the worker nodes, which executes a container from the corresponding image. The placement of task on each worker is based on the resource availability, computation required to be performed in each section and the capabilities of the nodes (obtained by the *Resource Monitor*). For intensive computations with large storage requirements (Gated Recurrent Units or LSTMs) or splits with high dimension size of input/output (typically the final layers), the splits are sent to high-resource edge workers. The management of allocation and migration of neural network splits is done by the *Container Orchestrator*. Other attention based sub-layer extensions can be deployed in either edge or cloud node based on application requirements, node constraints and user demands. Based on the described model assumptions, we now formulate the problem of taking splitting and placement decisions to optimize the QoS parameters. Implementation specific details on how the results of layer-splits are forwarded and outputs of semantic splits combined across edge nodes are given in Section 5.2.4.

5.1.3 Problem Formulation

The aim of the model is to optimize an objective score O_t (to be maximized), which quantifies the QoS parameters of the interval I_t , such as accuracy, SLA violation rate, energy consumption and average response time. This is typically in the form of a convex combination of energy consumption, response time, SLO violation rates, etc. [da Silva and da Fonseca, 2018]. The constraints in this formulation include the following. Firstly, the container decomposition for a new task $i \in N_t$ should be based on d^i . Secondly, containers corresponding to the layer-split decisions $\{C^i | d^i = L\}$ should be scheduled as per the linear chain of precedence constraints. This means that a container later in the neural inference pipeline should be scheduled only after the complete execution of the previous containers in the pipeline. This is because the output of an initial layer in an inference pipeline of a neural network is required before we can schedule a latter layer in the pipeline. Thirdly, the placement matrix $P_t : C_t \times H$ should adhere to the allocation constraints, *i.e.*, it should not allocate/migrate a container to a worker where the worker does not have sufficient resources available to accommodate the container. Thus, the problem can be formulated as

$$\begin{aligned}
& \underset{P_t, \mathcal{D}_t}{\text{maximize}} && \sum_t^T O_t \\
& \text{subject to} && \forall t, \forall i \in N_t, C^i \text{ containers created} \\
& && \text{based on splitting decision } d^i, \\
& && \forall t, P_t \text{ is feasible,} \\
& && \forall d^i = L, C^i \text{ follow precedence chain.}
\end{aligned} \tag{5.1}$$

5.2 SplitPlace Policy

We now describe the SplitPlace decision and placement policy. For the first sub-problem of deciding optimal splitting strategy, we employ a Multi-Armed Bandit model to dynamically enforce the decision using external

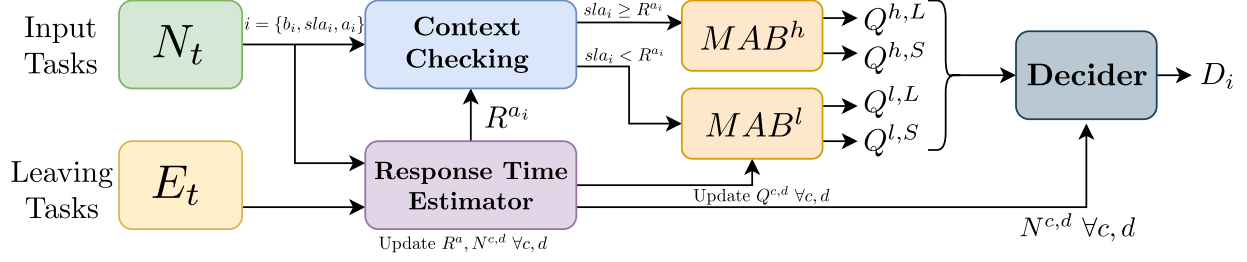


Figure 5.4: MAB decision workflow.

reward signals¹. Our solution for the second sub-problem of split placement uses a reinforcement-learning based approach that specifically utilizes a surrogate model to optimize the placement decision (agnostic to the specific implementation)². This two-stage approach is suboptimal since the response time of the splitting decision depends on the placement decision. In case of large variation in terms of the computational resources, it is worth exploring joint optimization of both decisions. However, in our large-scale edge settings, this segregation helps us to make the problem tractable as we describe next.

The motivation behind this segregation is two-fold. First, having a single reinforcement-learning (RL) model that takes both splitting and placement decisions makes the state-space explode exponentially, causing memory bottlenecks in resource-constrained edge devices [Chen et al., 2018]. Having a simple RL model does not allow it to scale well with several devices in modern IoT settings (see Section 5.3 with 50 edge devices the and Gillis RL baseline). One of the solutions that we explore in this chapter is to simplify this complex problem by decomposing it into split decision making and task placement. Second, the response time of an application depends primarily on the splitting choice, layer or semantic, making it a crucial factor for SLA deadline based decision making. To minimize the SLA violation rates we only use the response time based context for our Multi-Armed bandit model. Other parameters like CPU or RAM utilization have high variability in a volatile setting and are not ideal choices for deciding which splitting strategy to opt. Instead, the inference accuracy is another key factor in taking this decision. Thus, SLA violation and inference accuracy are apt objectives for the first sub-problem. Further, the energy consumption and average response time largely depend on the task placement, making them an ideal objective for optimization in the task placement sub-problem.

5.2.1 Multi-Armed Bandit Decision Module

Multi-Armed Bandit, in short MAB, is a policy formulation where a state-less agent is expected to take one of many decisions with each decision leading to a different reward. The objective of such an agent is to maximize the expected long-term reward [Bouneffouf et al., 2020]. However, in our case, the most important factor to consider when making a decision of whether to use layer or semantic splits for a task is its SLA deadline.

¹Compared to other methods like A/B testing and Hill Climbing search [Kohavi and Longbotham, 2017], Multi-Armed Bandits allow quick convergence in scenarios when different cases need to be modelled separately, which is the case in our setup. Thus, we use Multi-Armed Bandits for deciding the optimal splitting strategy for an input task.

²In contrast to Monte Carlo or Evolutionary methods, Reinforcement learning allows placement to be goal-directed, *i.e.*, aims at optimizing QoS using it as a signal, and allows the model to adapt to changing environments [Wiering and Van Otterlo, 2012]. Hence, we use a RL model, specifically using a surrogate model due for its scalability, to decide the optimal task placement of network splits.

5.2.1.1 Estimating Response Time of Layer-Splits

The idea behind the proposed SplitPlace approach is to maintain MABs for two different contexts: 1) when SLA is greater than the estimate of the response time for a layer decision, 2) when SLA is less than this estimate. The motivation behind these two contexts is that in case of the SLA deadline being lower than the execution time of layer split, a “layer” decision would be more likely to violate the SLA as result delivery would be after the deadline. However, the exact time it takes to completely execute all containers corresponding to the layer split decision is apriori unknown. Thus, for every application type, we maintain estimates of the response time, *i.e.*, the total time it takes to execute all containers corresponding to this decision.

Let us denote the tasks leaving the system at the end of I_t as E_t . Now, for each task $i \in E_t$, we denote response time and inference performance using r_i and p_i . We denote the layer response time estimate for application $a \in \mathcal{A}$ as R^a . To quickly adapt to non-stationary scenarios, for instance due to the mobility of edge nodes in the system, we update our estimates using new data-points as exponential moving averages using the multiplier $\phi \in [0, 1]$ for the most recent response time observation. Moving averages presents a low computational cost and consequently low latency compared to more sophisticated smoothing functions.

$$R^a \leftarrow \phi \cdot r_i + (1 - \phi) \cdot R^a, \forall i \in E_t \wedge d^i = L, \forall a \in A. \quad (5.2)$$

Compared to simple moving average, the above equation gives higher weights to the latest response times, allowing the model to quickly respond to recent changes in environment and workload characteristics.

5.2.1.2 Context based MAB Model

Now, for any input task $i \in N_t$, we divide it into two cases: $sla_i \geq R^{a_i}$ and $sla_i < R^{a_i}$. Considering that the response time of a semantic-split decision would likely be lower than the layer-split decision, in the first case both decisions would most likely not lead to an SLA violation (*high SLA* setting). However, in the second case, a layer-split decision would likely lead to an SLA violation but not the semantic-split decision (*low SLA* setting). To tackle the problem for these different contexts, we maintain two independent MAB models denoted as MAB^h and MAB^l . The former represents a MAB model for the high-SLA setting and the latter for the low-SLA setting.

For each context and decision $d \in \{L, S\}$, we define reward metrics as

$$O^{h,d} = \frac{\sum_{i \in E_t} (\mathbb{1}(r_i \leq sla_i) + p_i) \cdot \mathbb{1}(sla_i \geq R^{a_i} \wedge d^i = d)}{2 \cdot \sum_{i \in E_t} \mathbb{1}(sla_i \geq R^{a_i} \wedge d^i = d)}, \quad (5.3)$$

$$O^{l,d} = \frac{\sum_{i \in E_t} (\mathbb{1}(r_i \leq sla_i) + p_i) \cdot \mathbb{1}(sla_i < R^{a_i} \wedge d^i = d)}{2 \cdot \sum_{i \in E_t} \mathbb{1}(sla_i < R^{a_i} \wedge d^i = d)}. \quad (5.4)$$

The first term of the numerator, *i.e.*, $\mathbb{1}(r_i \leq sla_i)$ quantifies SLA violation reward (one if not violated and zero otherwise). The second term, *i.e.*, p_i corresponds to the inference accuracy of the task. These two objectives have been motivated at the start of Section 5.2. Thus, each MAB model gets the reward function for its decisions allowing independent training of the two. The weights of the two metrics, *i.e.*, accuracy and SLA violation can be set by the user to modify the relative importance between the metrics as per application requirements. In our experiments, the weight parameters of both metrics are set to be equal based on grid-search, maximizing

Algorithm 5.1 SplitPlace Decision and Placement Module**Require:**

Pre-trained MAB models MAB^h, MAB^l
 Discounting factor $\gamma \in (0, 1)$

- 1: **procedure** SPLITPLACE(scheduling interval I_t)
- 2: Get new tasks N_t
- 3: Get leaving tasks E_t
- 4: Calculate $O^{c,d} \forall c \in \{h, l\}, d \in \{L, S\}$ using (5.3) and (5.4)
- 5: Update gain estimates $Q^{c,d} \forall c, d$ using (5.5) ▷ Q update
- 6: Update decision counts $N^{c,d} \forall c, d$ ▷ Count update
- 7: **for** $i \in N_t$ **do**
- 8: $i = \{b_i, sla_i, a_i\}$
- 9: $d^i \leftarrow \begin{cases} \arg \max_{d \in \{L, S\}} Q^{h,d} + c\sqrt{\frac{\log t}{N^{h,d}}}, & sla_i \geq R^{a_i} \\ \arg \max_{d \in \{L, S\}} Q^{l,d} + c\sqrt{\frac{\log t}{N^{l,d}}}, & sla_i < R^{a_i} \end{cases}$ ▷ UCB based decision
- 10: **end for**
- 11: $\mathcal{D}_t \leftarrow \{d^i\}_{i \in N_t}$
- 12: Get S_t from resource-monitor ▷ Current State
- 13: $P_t \leftarrow \text{DASO}([S_t, P_{t-1}, \mathcal{D}_t])$ ▷ Placement Decision
- 14: $O^{MAB} \leftarrow \frac{1}{4} \sum_{c \in \{h, l\}} \sum_{d \in \{L, D\}} O^{c,d}$
- 15: Fine-tune DASO using O^P calculated using (5.10)
- 16: **return** P_t
- 17: **end procedure**

the average reward.

Now, for each decision context $c \in \{h, l\}$ and $d \in \{L, S\}$, we maintain a decision count $N^{c,d}$ and a reward estimate $Q^{c,d}$ which is updated using the reward functions $O^{h,d}$ or $O^{l,d}$ as follows

$$Q^{c,d} \leftarrow Q^{c,d} + \gamma(O^{c,d} - Q^{c,d}), \forall d \in \{L, S\}, \forall c \in \{h, l\}. \quad (5.5)$$

where γ is the decay parameter. Thus, each reward-estimate is updated by the corresponding reward metric.

For both these MAB models, we use a parameter-free feedback-based ϵ -greedy learning approach that is known to be versatile in adapting to diverse workload characteristics [Maroti, 2019]. Unlike other strategies, this is known to scale asymptotically as the long-term Q estimates become exact under mild conditions [Sutton and Barto, 2018, § 2.2]. To train the model, we take the contextual decision

$$d^i = \begin{cases} \begin{cases} \text{random decision,} & \text{with prob. } \epsilon \\ \arg \max_{d \in \{L, S\}} Q^{h,d}, & \text{otherwise} \end{cases}, & sla_i \geq R^{a_i} \\ \begin{cases} \text{random decision,} & \text{with prob. } \epsilon \\ \arg \max_{d \in \{L, S\}} Q^{l,d}, & \text{otherwise} \end{cases}, & sla_i < R^{a_i} \end{cases}. \quad (5.6)$$

Here, the probability ϵ decays using the reward feedback, starting from 1. We maintain a reward threshold ρ that is initialized as a small positive constant $k < 1$, and use average reward $O^{MAB} = \frac{1}{4} \sum_{c \in \{h, l\}} \sum_{d \in \{L, D\}} O^{c,d}$

to update ϵ and ρ using the rules

$$\epsilon \leftarrow \begin{cases} \text{decay}(\epsilon), & O^{MAB} > \rho \\ \epsilon, & \text{otherwise} \end{cases}, \quad (5.7)$$

$$\rho \leftarrow \begin{cases} \text{increment}(\rho), & O^{MAB} > \rho \\ \rho, & \text{otherwise} \end{cases}. \quad (5.8)$$

Here $\text{decay}(\epsilon) = (1 - k) \cdot \epsilon$ and $\text{increment}(\rho) = (1 + k) \cdot \rho$. Note that $O^{MAB} > \rho$ refers to the current value of ρ prior to the update. The k value controls the rate of convergence of the model. The ϵ value controls the exploration of the model at training time allowing the model to visit more states and obtain precise estimates of layer-split response times.

However, at test time we already have precise estimates of the response times; thus exploration is only required to adapt in volatile scenarios. For this, ϵ -greedy is not a suitable approach as decreasing ϵ with time would prevent exploration as time progresses. Instead, we use an Upper-Confidence-Bound (UCB) exploration strategy that is more suitable as it takes decision counts also into account [Zhang et al., 2019, Gupta et al., 2020]. Thus, at test time, we take a deterministic decision using the rule

$$d^i = \begin{cases} \arg \max_{d \in \{L, S\}} Q^{h,d} + c \sqrt{\frac{\log t}{N^{h,d}}}, & sla_i \geq R^{a_i} \\ \arg \max_{d \in \{L, S\}} Q^{l,d} + c \sqrt{\frac{\log t}{N^{l,d}}}, & sla_i < R^{a_i} \end{cases}, \quad (5.9)$$

where t is the scheduling interval count and c is the exploration factor. An overview of the complete split-decision making workflow is shown in Figure 5.4. We now discuss the RL based placement module. It is worth noting that both components are independent of each other and can be improved separately in future; however, our experiments show that the MAB decision module accounts for most of the performance gains (see Section 5.3).

5.2.2 Reinforcement Learning based Placement Module

Once we have the splitting decision for each input task $i \in N_t$, we now can create containers using pre-trained layer and semantic split neural networks corresponding to the application a_i . This can be done offline on a resource rich system, where the split models can be trained using existing datasets. Once we have trained models, we can generate container images corresponding to each split fragment and distribute to all worker nodes [Ahmed and Pierre, 2018]. Then we need to place containers, translating to the worker node initializing a container for the corresponding image, of all active tasks C_t to workers H . To do this, we use a learning model which predicts the placement matrix P_t using the state of the system S_t , decisions $\mathcal{D}_t = d^i \forall i \in A_t$ and a reward signal O^P . To define O^P , we define the following metrics:

1. *Average Energy Consumption* (AEC) is defined for any interval I_t as the mean energy consumption of all edge workers in the system.
2. *Average Response Time* (ART) is defined for any interval I_t as mean response time (in scheduling intervals) of all leaving tasks E_t .

The choice of these two objectives for the placement sub-problem has been motivated at the start of Section 5.2. Using these metrics, for any interval I_t , O^P is defined as

$$O^P = O^{MAB} - \alpha \cdot AEC_t - \beta \cdot ART_t. \quad (5.10)$$

Here, α and β (such that $\alpha + \beta = 1$) are hyper-parameters that can be set by users as per the application requirements. Higher α aims to optimize energy consumption at the cost of higher response times, whereas low α aims to reduce average response time. Thus, a RL model f , parameterized by θ takes a decision P_t , where the model uses the reward estimate as the output of the function $f([S_t, P_t, \mathcal{D}_t]; \theta)$, where the parameters θ are updated based on the reward signal O^P . We call this learning approach “decision-aware” as part of the input is the split-decision taken by the MAB model.

DASO placement module. In the proposed framework, we use decision-aware surrogate based optimization method (termed as DASO) to place containers in a distributed mobile edge environment. This is motivated from the GOBI method mentioned in Chapter 3. Here, we consider a Fully-Connected-Network (FCN) model $f(x; \theta)$ that takes an x as a tuple of input state S_t , split-decision $\mathcal{D}_t$ and placement decision P_t , and outputs an estimate of the QoS objective score O_t . This is because FCNs are agnostic to the structure of the input and hence a suitable choice for modeling dependencies between QoS metrics and model inputs like resource utilization and placement decision [Basu et al., 2019]. Exploration of other styles of neural models, such as graph neural networks that can take the network topology graph as an input are part of future work. Now, using existing execution trace dataset, $\Lambda = \{[S_t, P_t, \mathcal{D}_t], O_t\}_b$, the FCN model is trained to optimize its network parameters θ such that the Mean-Square-Error (MSE) loss

$$\mathcal{L}(f(x; \theta), y) = \frac{1}{b} \sum_{t=0}^b (y - f(x; \theta))^2, \text{ where } (x, y) \in \Lambda. \quad (5.11)$$

is minimized as in GOBI (see Chapter 3). To do this, we use AdamW optimizer [Loshchilov and Hutter, 2018] and update θ up till convergence. This allows the surrogate model f to predict an QoS objective score for a given system state S_t , split-decisions $\mathcal{D}_t$ and task placement P_t . Once the surrogate model is trained, starting from the placement decision from the previous interval $P_t = P_{t-1}$, we leverage it to optimize the placement decision using the following rule

$$P_t \leftarrow P_t - \eta \cdot \nabla_{P_t} f([S_t, P_t, \mathcal{D}_t]; \theta), \quad (5.12)$$

for a given state and decision pair $S_t, \mathcal{D}_t$. Here, η is the learning rate of the model. The above equation is iterated till convergence, *i.e.*, the $L2$ norm between the placement matrices of two consecutive iterations is lower than a threshold value. Thus, at the start of each interval I_t , using the output of the MAB decision module, the DASO model gives us a placement decision P_t .

5.2.3 SplitPlace Algorithm

An overview of the SplitPlace approach is given in Algorithm 5.1. Using pre-trained MAB models, *i.e.*, Q-estimates $Q^{c,d}$ and decision counts $N^{c,d}$, the model decides the optimal splitting decision using the UCB metric (line 9). To adapt the model in non-stationary scenarios, we dynamically update the Q-estimates and decision counts (lines 5 and 6). Using the current state and the split-decisions of all active tasks, we use the DASO

Table 5.1: Edge Worker characteristics of Azure Edge Environment.

Name	Qty	Core count	MIPS	RAM	RAM Bandwidth	Ping time	Network Bandwidth	Disk Bandwidth	Cost Model
Worker Nodes									
B2ms	20	2	4029	4295 MB	372 MB/s	2 ms	1000 MB/s	13.4 MB/s	0.0944 \$/hr
E2asv4	10	2	4019	4172 MB	412 MB/s	2 ms	1000 MB/s	10.3 MB/s	0.148 \$/hr
B4ms	10	4	8102	7962 MB	360 MB/s	3 ms	2500 MB/s	10.6 MB/s	0.189 \$/hr
E4asv4	10	4	7962	7962 MB	476 MB/s	3 ms	2500 MB/s	11.64 MB/s	0.296 \$/hr
Broker Node									
L8sv2	1	8	16182	17012 MB	945 MB/s	1 ms	4000 MB/s	17.6 MB/s	0.724 \$/hr

approach to take a placement decision for the active containers (line 13). Again, we fine-tune the DASO’s surrogate model using the reward metric to adapt to changes in the environment, for instance the changes in the latency of mobile edge nodes and their consequent effect on the reward metrics (line 15). However, the placement decision must conform to the allocation constraints as described in Section 5.1.3. To relax the constraint of having only feasible placement decisions, in SplitPlace we allocate or migrate only those containers for which it is possible. Those containers that could not be allocated in a scheduling interval are placed to nodes corresponding to the highest output of the neural network f . If no worker placement is feasible the task is added to a wait queue, which are considered again for allocation in the next interval.

5.2.4 Implementation

To implement and evaluate the SplitPlace policy, we use the COSCO framework introduced in Chapter 3.

We use only the edge-layer deployment in the framework and use the Docker container engine to containerize and execute the split-neural networks in various edge workers [Ahmed and Pierre, 2018]. We use the Checkpoint/Restore In Userspace (CRIU) [Venkatesh et al., 2019] tool for container migration. Further, the DASO approach is implemented using the Autograd package in the PyTorch module [Paszke et al., 2017].

To implement SplitPlace in the COSCO framework, we extend the `Framework` class to allow constraints for sequential execution of layer-splits. The function `getPlacementPossible()` was modified to also check for containers of layer-split partitioning scheme to be scheduled sequentially. Moreover, we implemented data transferring pipeline for broadcasting inputs in semantic-split decision and forwarding the outputs in layer-split decision. Finally, the inference outputs were synchronized and brought to the broker to calculate the performance accuracy and measure the workflow response time. For synchronization of outputs and execution of network splits, we use the HTTP Notification API.

5.3 Performance Evaluation

To test the efficacy of the SplitPlace approach and compare it against the baseline methods, we perform experiments on a heterogeneous edge computing testbed. To do this we emulate a setting with mobile edge devices mounted on self-driving cars, that execute various image-recognition tasks.

5.3.1 Experiment Setup

As in prior work [Basu et al., 2019], we use $\alpha = \beta = 0.5$ in (5.10) for our experiments (we consider other value pairs in Section 5.3.8). Also, we use the exploration factor $c = 0.5$ for the UCB exploration and the exponential moving average parameter $\phi = 0.9$, chosen using grid-search using the cumulative reward as the metric to maximize. We create a testbed of 50 resource-constrained VMs located in the same geographical location of London, United Kingdom using Microsoft Azure. The worker resources are shown in Table 5.1. All machines use Intel i3 2.4 GHz processor cores with processing capacity of no more than a Raspberry Pi 4B device. To keep storage costs consistent, we keep Azure P15 Managed disk with 125 MB/s disk throughput and 256 GB size³. The worker nodes have 4-8 GB of RAM, whereas the broker has 16 GB RAM. To factor in the mobility of the edge nodes, we use the **NetLimiter** tool to tweak the communication latency with the broker node using the mobility model described in [Gilly et al., 2020]. Specifically, we use the latency and bandwidth parameters of workers from the traces generated using the Simulation of Urban Mobility (SUMO) tool [Krajzewicz et al., 2012] that emulates mobile vehicles in a city like environment. SUMO gives us the parameters like **ping** time and network bandwidth to simulate in our testbed using **NetLimiter**. The moving averages and periodic fine-tuning allow our approach to be robust towards any kind of dynamism in the edge environment, including the one arising from mobility of worker nodes.

Our Azure environment is such that all devices are in the same LAN with 10 MBps network interface cards to avoid network bottlenecks while transferring inputs, outputs and intermediate results across neural network splits. Even if the nodes are in the same LAN with high bandwidth connections, the SUMO model would emulate the affects of mobility as is common in prior work on mobile edge computing [Wang et al., 2022]. Further, we use the **cPickle**⁴ Python module to save the intermediate results using **bzip2** compression and **rsync**⁵ file-transfer utility to minimize the communication latency. For containers corresponding to a layer-split workload that are deployed in different nodes, the intermediate results are forwarded using the **scp** utility to the next container in the neural network pipeline. Similarly, for semantic splitting, the **cPickle** outputs are collected using **rsync** and concatenated using the **torch.cat** function.

We use the Microsoft Azure pricing calculator to obtain the cost of execution per hour (in US Dollars)⁶. The power consumption models are taken from the Standard Performance Evaluation Corporation (SPEC) benchmarks repository⁷. The Million-Instruction-per-Second (MIPS) of all VMs are computed using the **perf-stat**⁸ tool on the SPEC benchmarks. We run all experiments for 100 scheduling intervals, *i.e.*, $\Gamma = 100$, with each interval being 300 seconds long, giving a total experiment time of 8 hours 20 minutes. We average over five runs and use diverse workload types to ensure statistical significance in our experiments. We consider variations of the experimental setup in Section 5.3.9.

5.3.2 Workloads

Motivated from prior work [Yu et al., 2021], we use three families of popular DNNs as the benchmarking models: ResNet50-V2 [He et al., 2016], MobileNetV2 [Sandler et al., 2018] and InceptionV3 [Szegedy et al., 2017]. Each

³Azure Managed Disks <https://docs.microsoft.com/en-us/azure/virtual-machines/disks-types#premium-ssd>.

⁴**cPickle** module <https://docs.python.org/2/library/pickle.html#module-cPickle>.

⁵**rsync** tool <https://linux.die.net/man/1/rsync>.

⁶Microsoft Azure pricing calculator for South UK <https://azure.microsoft.com/en-gb/pricing/calculator/>.

⁷SPEC benchmark repository https://www.spec.org/cloud_iaas2018/results/.

⁸**perf-stat** tool <https://man7.org/linux/man-pages/man1/perf-stat.1.html>.

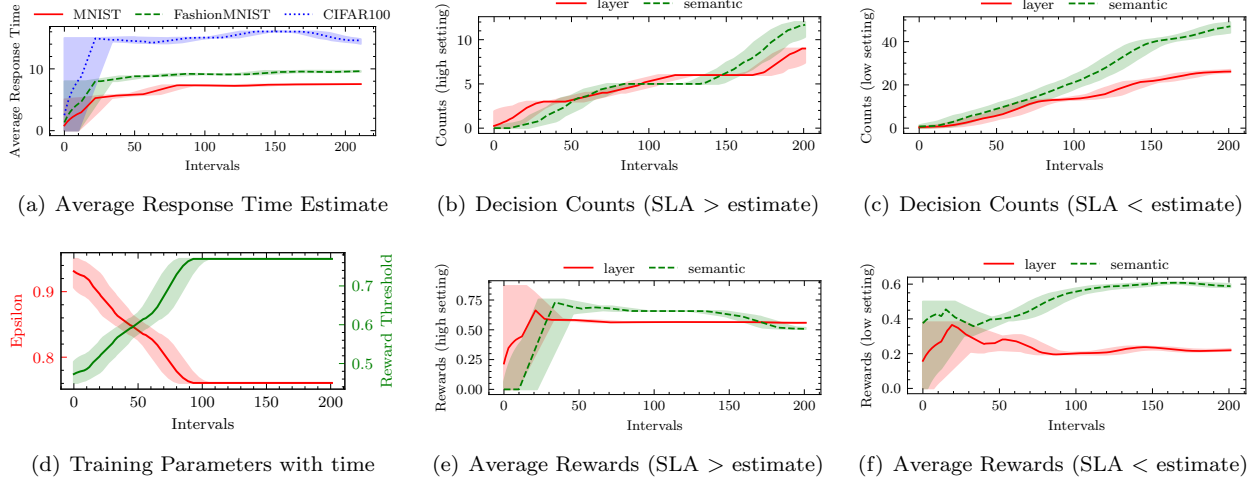


Figure 5.5: MAB training curves

family has many variants of the model, each having a different number of layers in the neural model. For instance, the ResNet model has 34 and 50 layers. We use three image-classification data sets: MNIST, FashionMNIST and CIFAR100 [LeCun et al., 1998, Xiao et al., 2017, Krizhevsky et al., 2009]. MNIST is a hand-written digit recognition dataset with 28×28 gray-scale images to 10-dimensional output. FashionMNIST has 28×28 RGB images with 10 dimensional output. CIFAR100 has 32×32 RGB images with 100-dimensional output. Thus the application set $\mathcal{A}$ becomes $\{\text{MNIST}, \text{FashionMNIST}, \text{CIFAR100}\}$. These models have been taken directly from the AIoTBench workloads [Luo et al., 2018]. This is a popular suite of AI benchmark applications for IoT and Edge computing solutions. The three specific datasets used in our experiments are motivated from the vertical use case of self-driving cars, which requires DNN-based applications to continuously recognize images with low latency requirements. Herein, an image recognition software is deployed that reads speed signs (digit recognition, MNIST), recognizes humans (through apparel and pose [Hoang, 2018], FashionMNIST), identifies other objects like cars and barriers (object detection, CIFAR100). We use the implementation of neural network splitting from prior work [Yu et al., 2021, Kim et al., 2017].

We use the inference deadline from the work [Yu et al., 2021] as our SLA. To create the input tasks, we use batch sizes sampled uniformly from 16,000 – 64,000. At the beginning of each scheduling interval, we create $Poisson(\lambda)$ tasks with $\lambda = 6$ tasks for our setup, sampled uniformly from one of the three applications. We consider other λ values and single workload type (from MNIST, FashionMNIST and CIFAR100) in Sections 5.3.7 and 5.3.10. The split fragments for MNIST, FashionMNIST and CIFAR100 lead to container images of sizes 8-14 MB, 34-56 MB and 47-76 MB, respectively. To calculate the inference accuracy to feed in the MAB models and perform UCB exploration, we also share the ground-truth labels of all datasets with all worker nodes at the time of sharing the neural models as Docker container images. We also compare edge and cloud setups in Section 5.3.11 to establish the need for edge devices for latency critical workloads.

5.3.3 MAB Training

To train our MAB models, we execute the workloads on the test setup for 200 intervals and use feedback-based ϵ -greedy exploration to update the layer-split decision response time estimates, Q -estimates and decision counts. Figure 5.5 shows the training curves for the two models.

Figure 5.5(a) shows how the response time estimates for the layer-split decision are learned starting from zero using moving averages. Figure 5.5(d) shows how the reward-threshold ρ and decay parameter ϵ change with time. We use the decay and increment multipliers as 0.9 and 1.1 ($k = 0.1$ in (5.7)) for ϵ and ρ respectively, as done in [Maroti, 2019]. Figures 5.5(b) and 5.5(c) show the decision counts for high and low SLA settings for both decisions. Figures 5.5(e) and 5.5(f) show the Q -estimates for high and low SLA settings. The dichotomy between the two settings is reflected here. When the sla_i of the input task i is less than the estimate R^{a_i} (low setting) there is a clear distinction between the rewards of the two decisions as layer-split is likely to lead to SLA violation and hence lower rewards. However, when sla_i is greater than the estimate R^{a_i} (high setting), both decisions give relatively high rewards with layer-split decision slightly surpassing the semantic-split due to higher average accuracy as discussed in Section 2.2.3.

The feedback-based ϵ -greedy training allows us to obtain close estimates of the average response times of the layer-split executions for each application type and average rewards for both decisions in high and low SLA settings. Thus, in our experiments, we initialize the expected reward (Q) and layer-split response time (R) estimates by the values we get from this training approach. At test time, we dynamically update these estimates using (5.2) and (5.5).

5.3.4 Performance Metrics

We use the following evaluation metrics in our experiments as motivated from prior works [Gill et al., 2019, Basu et al., 2019]. We also use *AEC* and *ART* as discussed in Section 5.2.

1. *Average Accuracy* is defined for an execution trace as the average accuracy of all tasks run in an experiment, *i.e.*,

$$Accuracy = \frac{\sum_t \sum_{i \in E_t} p_i}{\sum_t |E_t|}. \quad (5.13)$$

2. *Fraction of SLA violation* is defined for an execution trace as the fraction of all tasks run in an experiment for which the response time is higher than the SLA deadline, *i.e.*,

$$SLA \text{ Violations} = \frac{\sum_t \sum_{i \in E_t} \mathbb{1}(sla_i \geq r_i)}{\sum_t |E_t|}. \quad (5.14)$$

3. *Average Reward* is defined for an execution trace as follows

$$Reward = \frac{\sum_t \sum_{i \in E_t} \mathbb{1}(sla_i \geq r_i) + p_i}{2 \cdot \sum_t |E_t|}. \quad (5.15)$$

4. *Execution Cost* is defined for an execution trace as the total cost incurred during the experiment, *i.e.*,

$$Cost = \sum_{h \in H} \int_x C_h(x) dx. \quad (5.16)$$

where $C_h(x)$ is the cost function for worker h with time.

5. *Average Wait Time* is the average time a task had to wait in the wait queue till it could be allocated to a worker for execution.

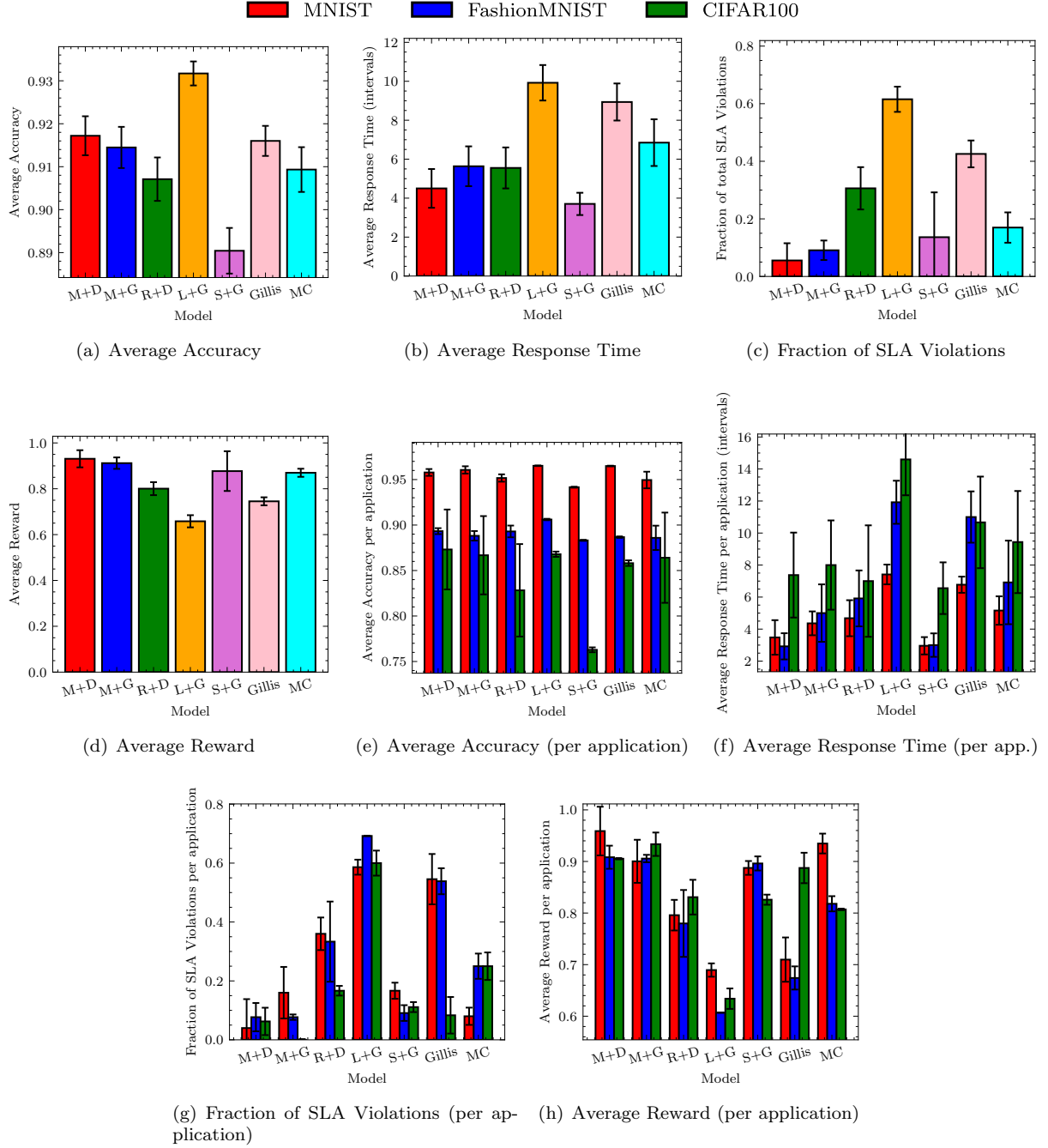


Figure 5.6: Comparison of SplitPlace against baselines and ablated models on physical setup with 50 edge workers

6. *Average Execution Time* is the response time minus the wait time, averaged for all tasks run in an experiment. This includes the compute time and the time lag due to container migrations.
7. *Fairness* is defined as the Jain's fairness index for execution on tasks over the edge workers (as in Chapter 3).

5.3.5 Baselines and Ablated Models

We compare the performance of the SplitPlace approach against the state-of-the-art baselines *Gillis* and BottleNet++ Model Compression (denoted as *MC* in our graphs) [Yu et al., 2021, Eshratifar et al., 2019, Shao and Zhang, 2020]. *Gillis* refers to the reinforcement learning method proposed in [Yu et al., 2021] that leverages

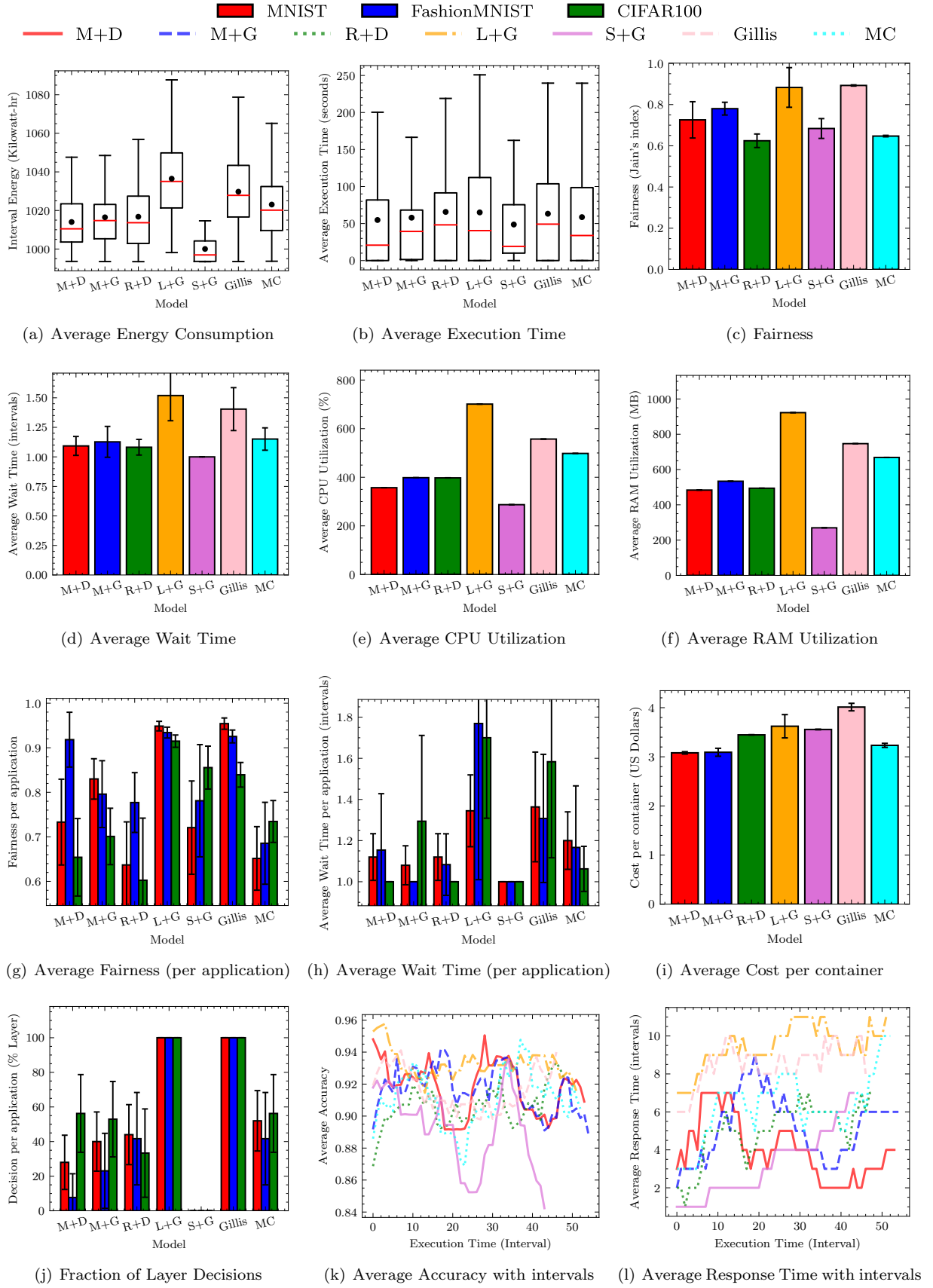


Figure 5.7: Additional results comparing SplitPlace with baselines and ablated models.

both layer-splitting and compression models to achieve optimal response time and inference accuracy. Note that contrary to the original Gillis' work, our implementation does not leverage serverless functions. MC is

Table 5.2: Comparison of SplitPlace with baseline and ablated models.

Model	Energy	Scheduling Time	Fairness	Wait Time	Response Time	SLA Violations	Accuracy	Average Reward
Baselines								
Model Compression	1.1368	8.84±0.02	0.65±0.01	1.15±0.09	6.85±1.20	0.26±0.02	89.93	83.98
Gillis	1.1442	8.22±0.01	0.89±0.03	1.40±0.18	8.39±0.95	0.22±0.03	91.90	84.17
Ablation								
Semantic+GOBI	1.1112	8.68±0.03	0.68±0.04	1.08±0.00	3.70±0.57	0.14±0.04	89.04	83.91
Layer+GOBI	1.1517	8.72±0.01	0.88±0.03	1.52±0.21	9.92±0.91	0.62±0.07	93.17	64.87
Random+DASO	1.1297	8.86±0.01	0.62±0.05	1.00±0.07	5.55±1.05	0.29±0.09	90.71	81.62
MAB+GOBI	1.1290	9.12±0.02	0.78±0.08	1.13±0.13	5.64±1.02	0.10±0.03	91.45	90.18
SplitPlace Model								
MAB+DASO	1.0867	9.32±0.02	0.73±0.01	1.09±0.08	4.50±1.00	0.08±0.02	92.72	94.18

a model-compression approach motivated from BottleNet++ that we implement using the PyTorch Prune library.⁹ Further details in Section 2.2.3. We do not include results for other methods discussed in Section 2.2.3 as MC and Gillis give better results empirically for all comparison metrics. We also compare SplitPlace with ablated models, where we replace one or both of the MAB or DASO components with simpler versions as described below.

- *Semantic+GOBI (S+G)*: Semantic-split decision only with vanilla GOBI placement module.
- *Layer+GOBI (L+G)*: Layer-split decision only with vanilla GOBI placement module.
- *Random+DASO (R+D)*: Random split decision with DASO placement module.
- *MAB+GOBI (M+G)*: MAB based split decider with vanilla GOBI placement module.

The final SplitPlace approach is represented as MAB+DASO or M+D in shorthand notation in the graphs. These ablated baselines help us determine the relative improvements in performance by the two components of MAB and DASO separately.

5.3.6 Results and Ablation Analysis

We now provide comparative results showing the performance of the proposed SplitPlace approach against the baseline models and argue the importance of the MAB and decision-aware placement using ablation analysis. We train the *GOBI* and *DASO* models using the execution trace dataset used to train the MAB models. The learning rate (η) was set to 10^{-3} .

Figure 5.6 shows the average reward and related performance metrics, *i.e.*, accuracy, response time and SLA violation rate. As expected, the L+G policy gives the highest accuracy of 93.17% as all decisions are layer-wise only with a higher inference performance than semantic-split execution. The S+G policy gives the least accuracy of 89.04%. However, due to layer-splits only the L+G policy also has the highest average response time, subsequently giving the highest SLA violation rate. On the other hand, S+G policy has the least average response time. However, due to the intelligent decision making in SplitPlace, it is able to get the highest total reward of 0.9418. Similar trends are also seen when comparing across models for each application. The

⁹PyTorch Prune. https://pytorch.org/docs/stable/generated/torch.nn.utils.prune.ln_structured.html. Accessed 10 October 2021.

accuracy is the highest for the MNIST dataset and lowest for CIFAR100. Average response time is highest for the CIFAR100 and lowest for MNIST in general. Among the baselines, the Gillis approach has the lowest SLA violation rate of 22% and SplitPlace improves upon this by giving 14% lower SLA violations (only 8%). Gillis has higher accuracy between the baselines of 91.9%, with SplitPlace giving an average improvement of 0.82%. Overall, the total reward of SplitPlace is higher than the baselines by at least 10.1%, giving the reward of 94.18%.

Figure 5.7 shows the performance of all models for other evaluation metrics like energy, execution time and fairness. Compared to the baselines, SplitPlace can reduce energy consumption by up to 4.41% – 5.03% giving an average energy consumption of 1.0867 MW-hr. However, the SplitPlace approach has higher scheduling time and lower fairness index (Table 5.2). The Gillis baseline has the highest fairness index of 0.89, however this index for SplitPlace is 0.73. SplitPlace has a higher overhead of 11.8% compared to the Gillis baseline in terms of scheduling time. Figure 5.7(i) compares the average execution cost (in USD) for all models. As SplitPlace is able to run the maximum number of containers in the 100 intervals, it has the least cost of 3.07 USD/container. The main advantage of SplitPlace is the intelligent splitting decisions facilitate overcoming the memory bottlenecks in edge environments, giving up to 32% lower RAM utilization compared to Gillis and Model Compression.

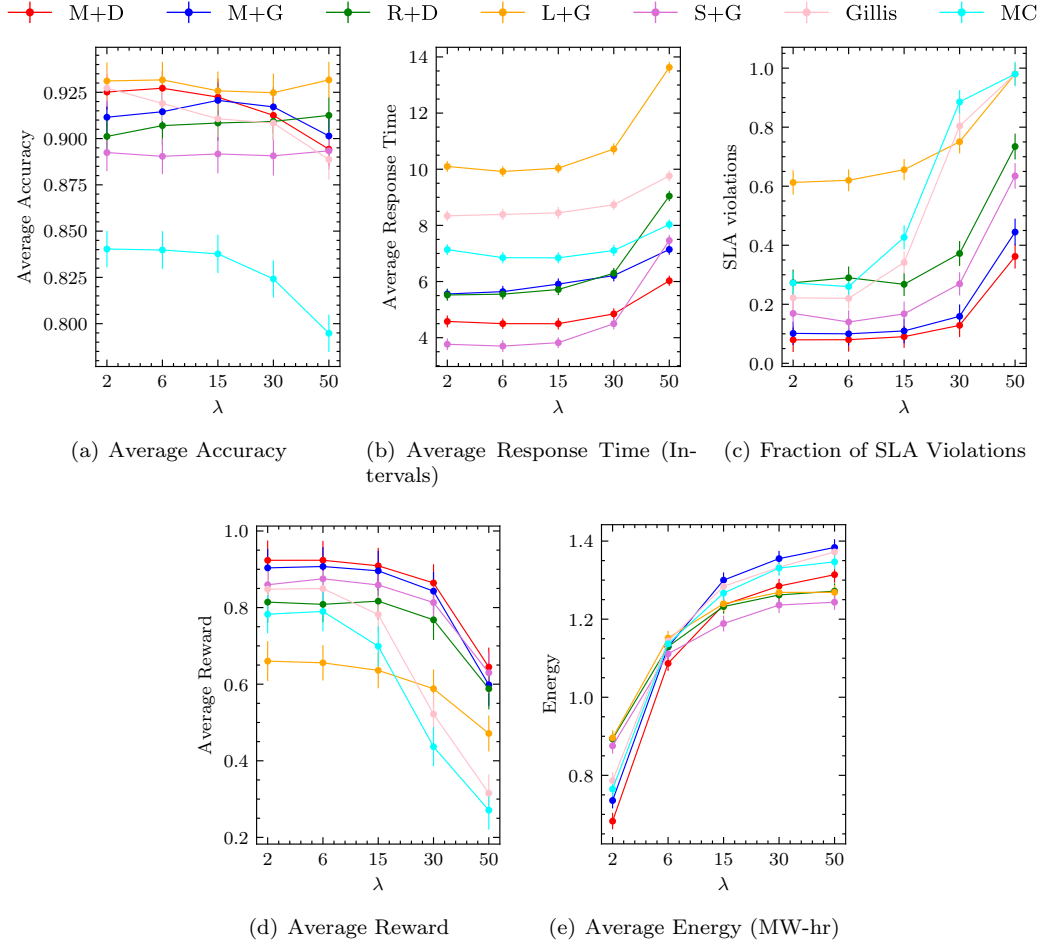
In terms of the initial communication time of the Docker container images, the SplitPlace method takes 30 seconds at the start of an execution. Gillis and MC have such communication times of 20 and 18 seconds, respectively. This demonstrates that SplitPlace has a low one-time overhead (up to 12 seconds) compared to the baselines when compared to the gains in response time (up to 46%) that linearly scales as the number of workloads increase.

A summary of comparisons with values of main performance metrics for all models is given in Table 5.2. The best values achieved for each metric are highlighted in bold.

5.3.7 Sensitivity to λ

We perform sensitivity analysis to the number of workloads being generated at the start of each scheduling interval to test the scalability of the SplitPlace model. As described in Section 5.3.2, we create $Poisson(\lambda)$ tasks at the beginning of each scheduling interval with $\lambda = 6$ for our setup. The tasks are sampled uniformly from the three applications: MNIST, FashionMNIST and CIFAR100 as done in our experiments in Section 5.3. We vary the λ parameter from 2 to 50 to test how the performance varies with this parameter. Beyond $\lambda = 50$, the baseline models diverge and the system breaks down due to computational overload for these models. Figure 5.8 demonstrates the average accuracy, response time, fraction of SLA violations, reward and energy consumption for all models with varying the λ parameter. Again, all experiments are performed for 100 scheduling intervals with the same setup as described in Section 5.3.1. Clearly, as λ increases, the number of workloads and hence the resource requirements increase.

The average accuracy observed with policies that take the splitting decision based on external reward signals (M+D, M+G, MC and Gillis) drops as the λ parameter increases. This is not seen for policies R+D, L+G or S+G, as their splitting decision (which affects the classification accuracy) are unaffected. The other policies see a gradual decline of accuracy by 2.61% – 4.83% as the ratio of semantic-split decisions increases to reduce resource consumption (see Figure 5.10). However, we observe that the response times increase with λ . This is

Figure 5.8: Sensitivity of λ on all models.

due to the increasing wait times and contention affects in congested edge environments. Still, the SplitPlace model (M+D) maintains a low average response time of 4.76 intervals, 33.61% – 45.31% lower than the baseline models (MC and Gillis). All models see a sharp increase in violation rate as λ increases to 30 tasks in terms of SLA violations. The L+G, MC and Gillis models show average violation rates of nearly 1 for $\lambda = 50$. However, the average slope is least for the SplitPlace model, increasing from 8% to 35.1% (increase of 27%), whereas for others this change is much higher 34.87% – 76.46%. Naturally, as SLA violation rates increase and accuracies decrease, the average reward would decrease (Figure 5.8(d)). The average rewards of the M+D model (88.44) are still higher than the baselines and ablated models (64.69 – 85.66). Figure 5.8(e) shows the change of energy consumption as λ increases. This metric sharply increases as λ changes from 2 tasks to 6 tasks. Upon further increase, the energy does not increase as significantly as edge environment reaches a saturation point and no more tasks can be scheduled. All models have similar trends in terms of the energy consumption metric.

5.3.8 Sensitivity with α, β

We also perform sensitivity analysis on the α, β parameters introduced in Section 5.2.2 for the SplitPlace and its ablated versions. α and β are the weights corresponding to the average energy consumption (AEC) and average response time (ART) metrics in Equation 5.10. Note that it is a convex combination of the two metrics, so $\alpha + \beta = 1$. These experiments help us test the robustness of the SplitPlace model in the presence of diverse QoS objectives.

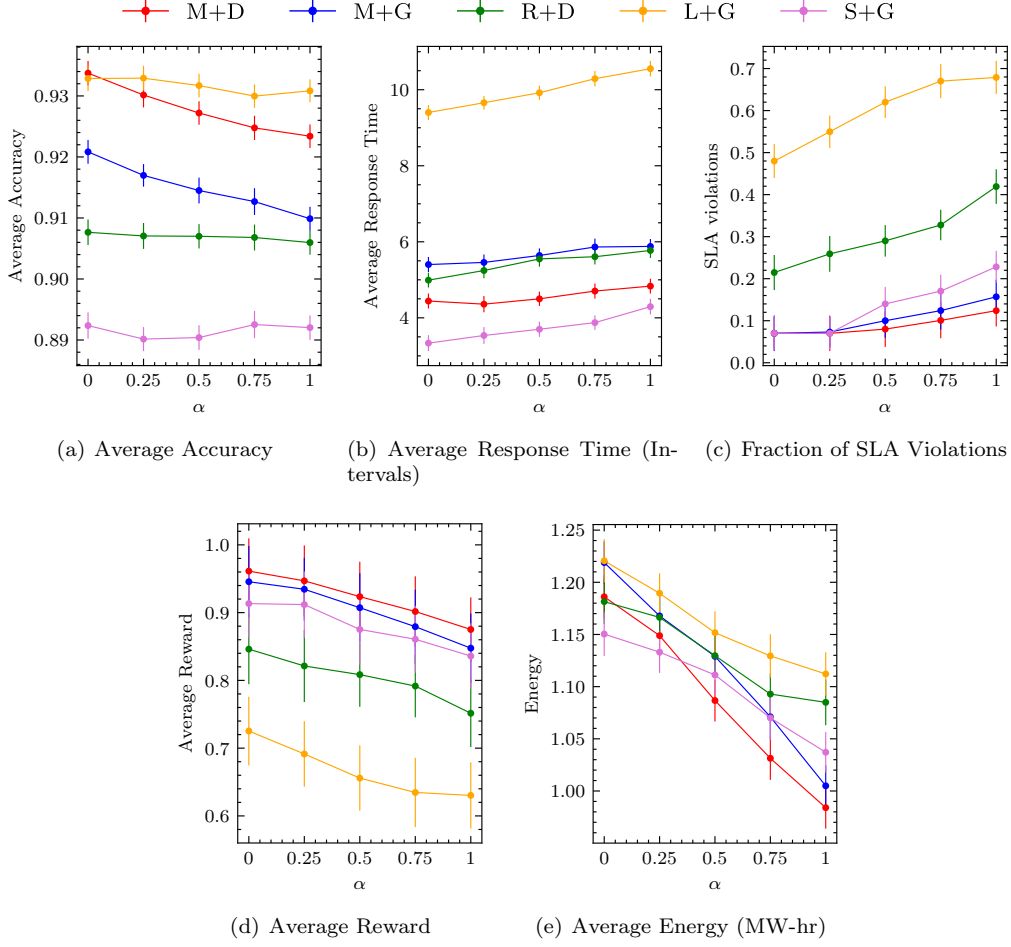
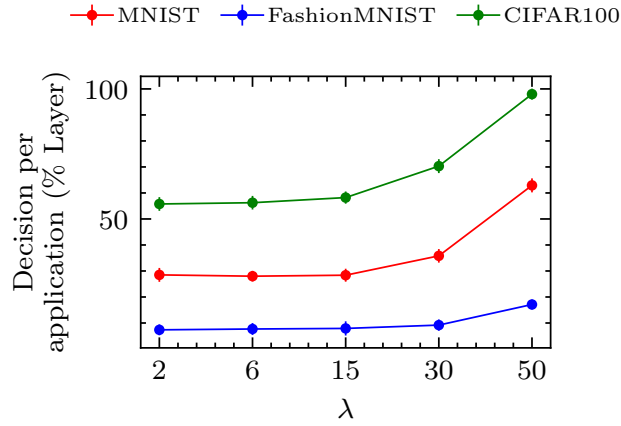
Figure 5.9: Sensitivity of α, β on ablated SplitPlace models.Figure 5.10: Fraction of layer-split decisions of the SplitPlace model with increasing λ .

Figure 5.9 shows the change in performance metrics when α varies from 0 to 1. Some observations overlap with Figure 5.8. In terms of energy consumption, all models show a decline in this metric as α increases. This is because of the energy conserving scheduling decisions by the GOBI model, allocating a higher fraction of tasks to the low-power edge nodes (B2ms and E2asv4). This leads to higher resource contention in these devices, leading to higher SLA violations, for which the MAB model tends to take a higher ratio of semantic-split decisions (due to their lower response times, see Figure 5.11). This leads to the decline in the accuracy of the M+D and M+G

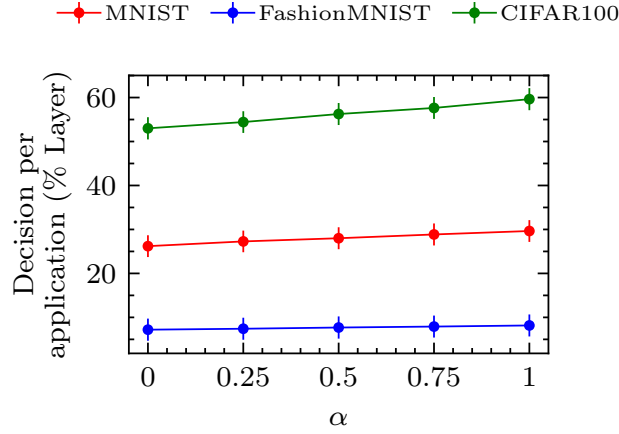


Figure 5.11: Fraction of layer-split decisions of the SplitPlace model with increasing α .

models as α increases, whereas the average accuracy for reward-free models (R+D, L+G, S+G) remains nearly unchanged. The average response times and SLA violation rates increase steadily for all models. However, the increase is not as significant for the MAB based models (M+D and M+G) due to the UCB based external signal. The average reward is still the highest for the SplitPlace model (91.93) compares to the ablated models (65.91 – 90.39).

5.3.9 Constraining Environments

We now perform the same experiments with various constraints in the original edge computing environment. These experiments demonstrate the ability of the MAB based models to adapt to varied deployment scenarios efficiently.

Changes in Setup: For each host in Table 5.1, we limit the core count (compute constrained), network bandwidth (network constrained) and RAM size (memory constrained) to half the original value¹⁰. These setups show us the performance of the SplitPlace model with respect to the baselines techniques in more diverse experimental scenarios.

Results: Figure 5.12 shows the performance metrics for the three constrained environments for all models. The primary y -axis gives the value of the metric in the new setting and the secondary y -axis gives us the ratio with respect to metric value in the original setup. Figure 5.13 shows the distribution of the response time to the average time taken for scheduling, waiting, data transfer, task execution and container migration. Figure 5.14 shows the distribution of the SLA violations for each application. Similar to what we observed earlier, for reward free models (R+D, L+G and S+G), the average accuracy is nearly unchanged, whereas for M+D, M+G, MC and Gillis, the accuracy drop is up to 40%. This drop is more pronounced in network and compute constrained environments as compared with the memory constrained one. This is because of the increasing average response times compared to the original setup due to lower computational resource availability, higher migration and data response times or low memory resources. This leads to higher resource contention in these devices, again for which the MAB model tends to take a higher ratio of semantic-split decisions to avoid too many SLA violations. This leads to the decline in accuracy of the M+D, M+G and

¹⁰To limit the CPU core usage in Linux kernel we modify the `grubconfig`. To limit the RAM usage, we use the `ulimit` tool. To limit the network bandwidth, we appropriately modify the parameters of the `Netlimiter` tool.

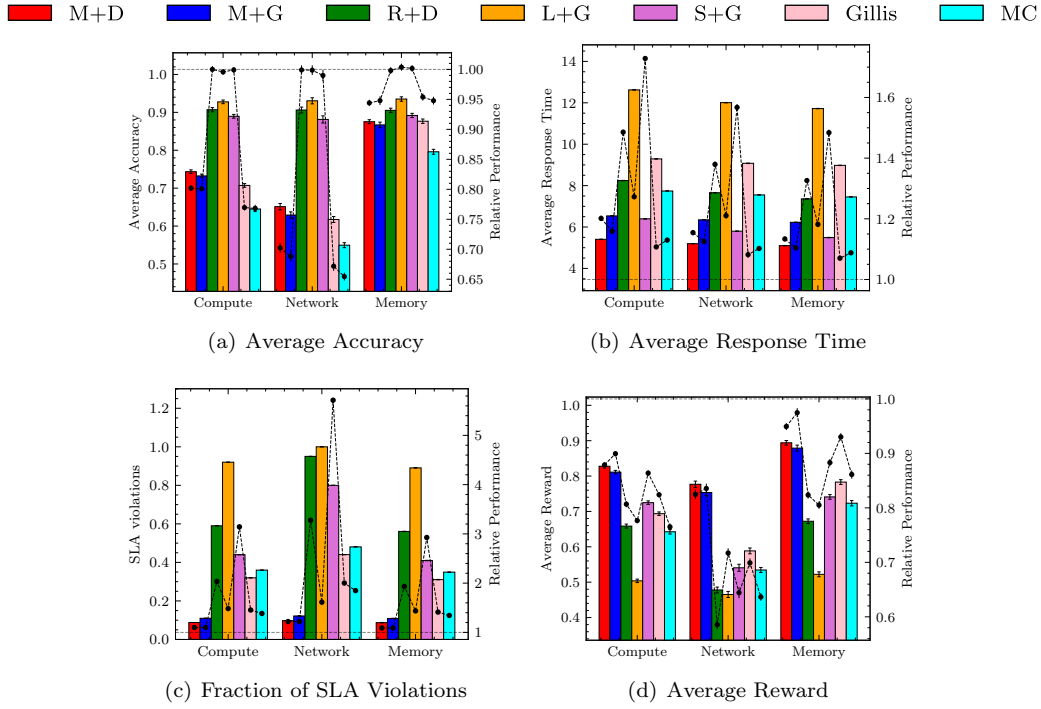


Figure 5.12: Comparison of SplitPlace against baselines and ablated models in Compute, Network and Memory constrained environments.

baseline models in the constrained environments, whereas the average accuracy for reward free models (R+D, L+G, S+G) remains nearly unchanged. However, the response times and fraction of SLA violations are much higher for the reward free models. The relative increase in the violation rates are much lower for the M+D and M+G policies, thanks to the intelligent adaptability of the MAB model to the changing response times in the new setups. The increase in the response times in the case of computational constraints is predominantly due to the increase in the execution times due to low compute resources available (Figure 5.13(b)). For network constrained setup, the increase in response time is majorly due to the increase in data transfer and container migration times (due to the lower bandwidth availability, see Figure 5.13(c)). In memory constrained setup, low memory availability causes the edge devices to use the disk-swap space increasing both execution and data transfer times (Figure 5.13(d)). These constraints have the highest impact on the resource hungry CIFAR100 workloads which leads to an increase in its SLA violation rate compared to other application types (Figure 5.14).

The average rewards for all models are lower than the original setup (Figure 5.12(d)). The relative rewards for MAB based models are the highest (0.84 – 0.95) compared to the baseline models (0.63 – 0.86). The reward drop is the least for the SplitPlace model 11.61%. The reward drops are higher for the reward-free models (14.05% – 22.91), due to their significantly higher SLA violation rate, even without any accuracy change. For all the three cases, the SplitPlace model has the highest average reward (0.77 – 0.90), whereas the Gillis and MC models have rewards in the range 0.58 – 0.78 and 0.53 – 0.71 respectively. The increase in the SLA violation rates are also the lowest for the SplitPlace model (9.24% – 22.01%) compared to the baselines with 34.73% – 103.11%. Similarly, the decrease in the average accuracy for the SplitPlace model is 9.31%, which for the baseline models is 16.98% – 32.13%. The average increase in the response time in the constrained setup for the SplitPlace model is 14.56%, whereas for the baselines it is 12.31%.

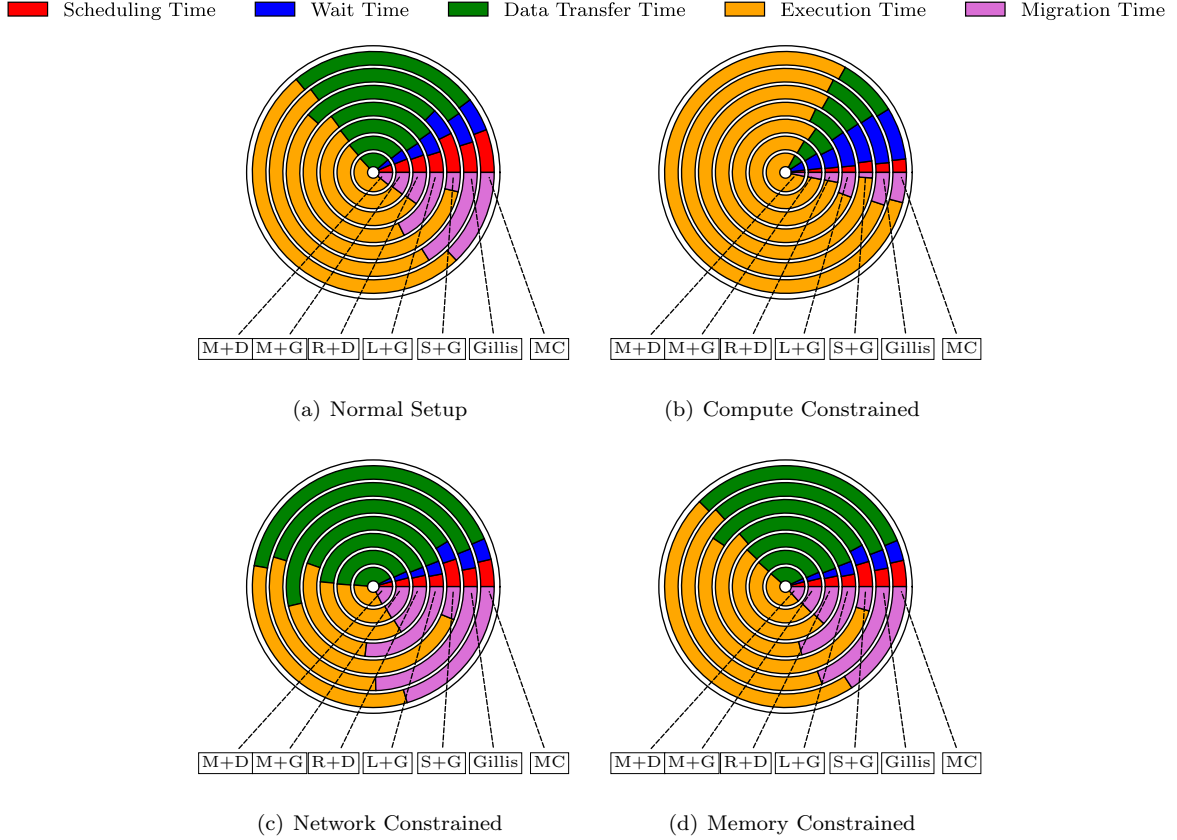


Figure 5.13: Distribution of response time in Normal and Compute, Network and Memory constrained environments.

5.3.10 Constraining Workloads

We now perform the same experiments, but now changing the workloads to MNIST only, FashionMNIST only and CIFAR100 only applications in lieu of sampling workloads uniformly at random from one of these. This helps us illustrate the robustness of the SplitPlace model against diverse application domains.

Results: Figure 5.15 shows the performance parameters for all models in the three workload settings. The graphs show contrasting workload specific trends. For instance, the accuracy of the MNIST application is higher and response time lower than the other two application types. Thus, for all models, the average accuracy is higher compared to the original workload and response time lower. However, due to lower resource requirements of the MNIST workload, the MC model seldom takes the decision to use compressed models, giving a high accuracy uplift compared to the original setup. Figure 5.16 shows the distribution of the response times of tasks for each workload setting. There are no drastic changes in the time distribution compared to the original workloads, the only difference being the different execution and data transfer times for various workloads. For smaller workloads like MNIST, the execution time and data transfer times are lower than larger workloads like CIFAR100. Considering the average response time and SLA violation rates, the SplitPlace model gives a lower average violations in all three cases. This is due to the limited workload complexity and simpler decision making for the MAB model.

Even with these constrained workloads, the SplitPlace model has the highest average accuracy with the lowest average response times and SLA violation rates. Overall, we see that the average reward of the SplitPlace model (89.83 – 95.36) is highest among all methods. The baseline models, Gillis and MC have rewards in the range

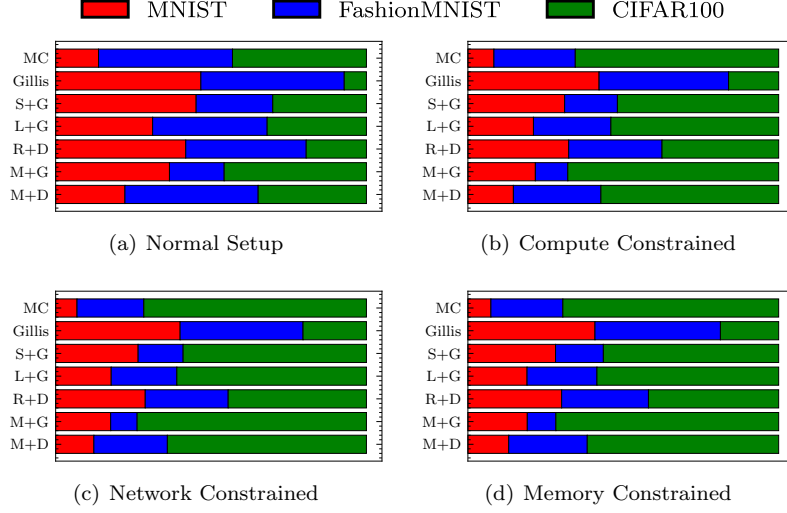


Figure 5.14: Distribution of SLA violations in Normal and Compute, Network and Memory constrained environments.

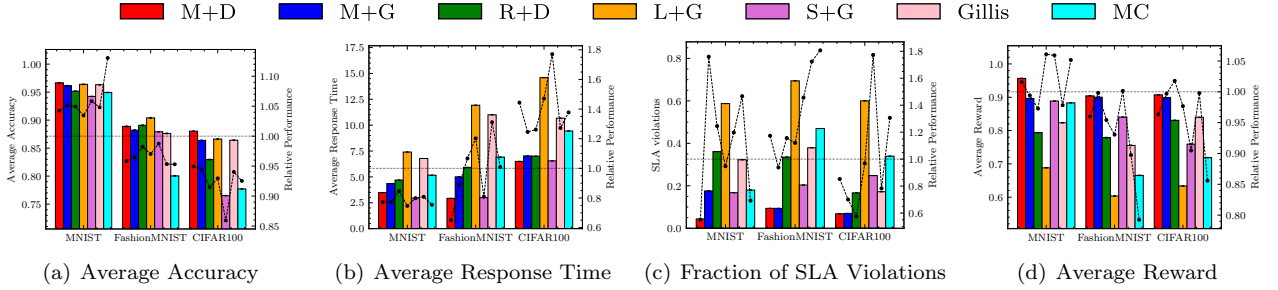


Figure 5.15: Comparison of SplitPlace against baselines and ablated models in settings with only MNIST, FashionMNIST and CIFAR100 workloads.

75.54 – 83.98 and 66.57 – 88.31 respectively.

5.3.11 Comparing Edge with Cloud

We also perform experiments to compare the performance of cloud versus edge platforms. For latency-critical applications, it is crucial to resort to edge devices that are in close proximity with the users. Relying on cloud nodes, which may be at multi-hop distance from the users lead to higher latency and subsequently the average response times. Thus, for latency critical applications, utilizing memory abundant cloud nodes, with typically 32-64 GB of memory available, is not feasible. This makes the problem challenging as now we need to ensure that we can run large-scale DNNs on edge, without any cloud backend whatsoever. Thus, in our formulation and experiments, we consider an environment with only memory-constrained edge nodes with 4-8 GB RAM and all in proximity with the users. To empirically verify this, we consider a cloud setup wherein the broker nodes remains in the Azure UK-South datacenter, whereas the worker nodes are now initialized in the East-US datacenter. For a fair comparison, we consider a setup where no splitting is required and end-to-end models are used in the cloud nodes. The response time and SLA violation rates are shown in Figure 5.17. The figure demonstrates that in a Cloud-AI setup, the same worker nodes are unable to conform to the strict deadlines when compared to the distributed Edge-AI setup. In terms of the initial one-time communication for transferring the containers, SplitPlace method on edge takes 30 seconds, whereas the cloud-only approach

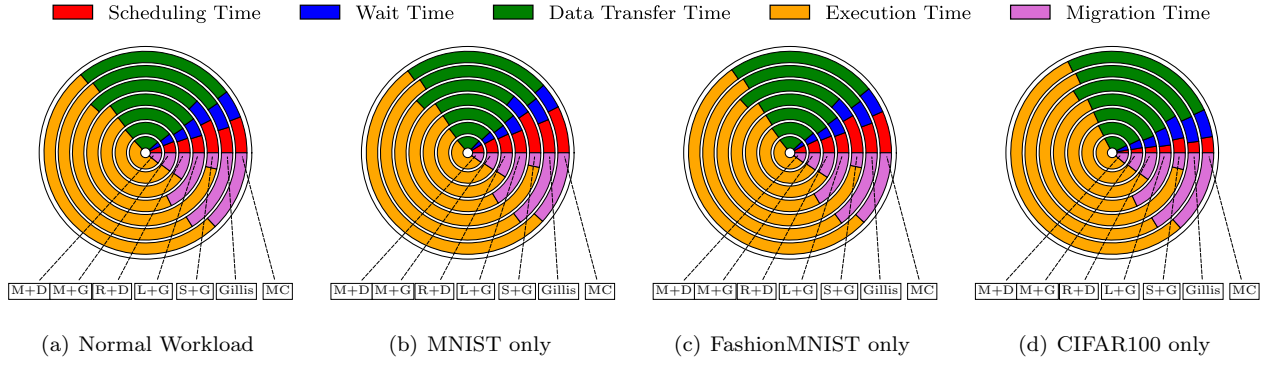


Figure 5.16: Distribution of response time in Normal and MNIST only, FashionMNIST only, CIFAR100 only workloads.

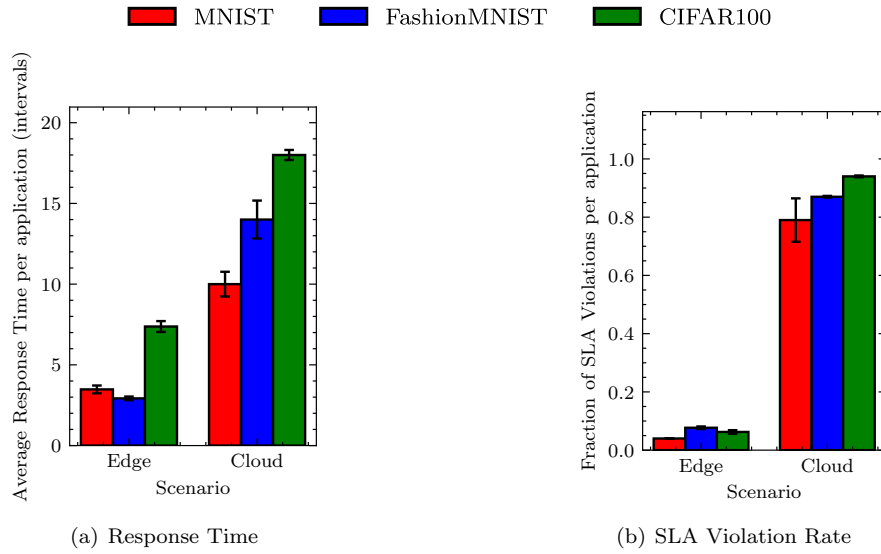


Figure 5.17: Comparing the response time and SLA violation rates for SplitPlace for edge and cloud setups.

takes 72 seconds. The single-hop communication topology among the broker-worker nodes foregoes the need for sending the container images via the Internet.

5.3.12 Comparing the Impact of Splitting and Placement Decisions on Response Time

The decomposition of decision making in SplitPlace is based on the hypothesis that the MAB models do not need split placement decisions to decide the splitting decisions. The major contributing factor for response time of a task is the split decision type as shown in Figure 5.1. As the figure demonstrates, the deviation in the response time is much higher when we compare layer vs semantic. The deviation is low when comparing across placement decisions. This is demonstrated by Figure 5.18. Even though we consider a heterogeneous edge computing environment, with edge devices having different resource capacities, the difference in resources is not sufficient to give significant deviation in terms of response time as per the placement decisions.

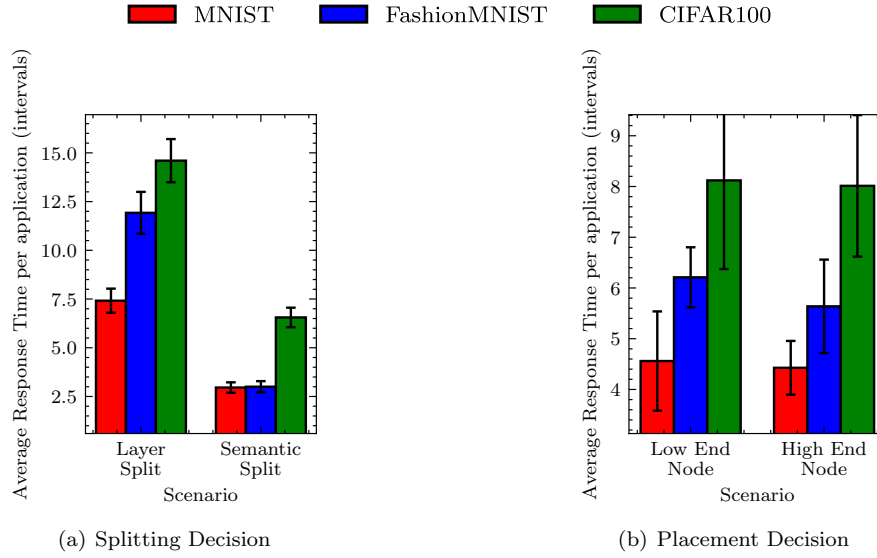


Figure 5.18: Comparing the response time across splitting and placement decisions.

5.4 Summary

In this chapter, we present SplitPlace, a novel framework for efficiently managing demanding neural network based applications. SplitPlace exploits the trade-off between layer and semantic split models where the former gives higher accuracy, but the latter gives much lower response times. This allows SplitPlace to not only manage tasks to maintain high inference accuracy on average, but also reduce SLA violation rate. The proposed model uses a Multi-Armed-Bandits based policy to decide which split strategy to use according to the SLA deadline of the incoming task. Moreover, it uses a decision-aware learning model to take appropriate placement decisions for those neural fragments on mobile edge workers. Further, both MAB and learning models are dynamically tuned to adapt to volatile scenarios. All these contributions allow SplitPlace to out-perform the baseline models in terms of average response time, SLA violation rate, inference accuracy and total reward by up to 46.3%, 69.2%, 3.1% and 12.1% respectively in a heterogeneous edge environment with real-world workloads. Thus, using the ideas of co-simulation and gradient-based optimization using surrogate models in Chapter 3 enable us to achieve objective 1(b) as mentioned in Chapter 1.

This page intentionally left blank.

Chapter 6

AI Augmented Task and Workflow Scheduling

This chapter describes novel task scheduling approaches, namely GOSH and MCDS and their extensions, to achieve optimal scheduling (objective 2) as described in Chapter 1. In Chapters 4 and 5, we discussed how to optimally provision resources and deploy applications in a fog environment. We also provide preliminary discussions on how to utilize a gradient-based optimization approach (GOBI) for task placement in Chapter 3. However, GOBI faces limitations in highly stochastic environments or with scientific workflow like tasks that have long-range computational run-times. Considering we have a provisioned resources and deployed applications, we now shift our focus on optimal task placement in dynamic Fog environments. We extend GOBI to incorporate more challenging scenarios of task and workflow scheduling.

Extending to GOSH. Advances like deterministic surrogate models, deep neural networks (DNN) and gradient-based optimization allow low energy consumption and response times to be reached. This is due to their ability to quickly adapt to diverse scenarios in real-time. However, due to the lack of state-space exploration or uncertainty modeling, such methods can perform poorly in settings unseen during model training [Panda et al., 2015]. Moreover, QoS optimization surfaces are known to be highly non-convex, and ignoring their curvature information can lead such methods to get stuck in local optima, saddle points or plateaus [Tychogiorgos et al., 2013]. To overcome the challenges of unseen settings, it is important to consider uncertainty in the model predictions for a guided exploration of the state-space. In general, uncertainty may arise from several factors, including model approximations, measurement imprecision and fluctuations of parameters over time [Tchernykh et al., 2015]. Specifically, having a deterministic model does not always give a complete picture of the distribution, due to the unknown nature or inherent stochasticity of some parameters like network latency, temperature and hardware faults [Ye et al., 2018]. Thus, in this work, we use heteroscedastic surrogate models, specifically the Natural Parameter Network (NPN) that uses twice the parameters of a traditional DNN to output both mean and variance estimates. Further, it is crucial to model the curvature of the QoS hyper-surface to converge quickly to an optima [Kochenderfer and Wheeler, 2019, § 6.1]. This entails using higher-order derivatives of the surrogate output while running decision optimization. Further, when modeling uncertainty, we may leverage this to trade-off between the exploration and exploitation to converge to the optimal scheduling decision [Russell and Norvig, 2009]. However, even after feeding more data, the performance of deep neural networks tends to

saturate and we use recently proposed co-simulation based bootstrapping schemes to circumvent this.

Drawing these insights together, we present **GOSH: Gradient Based Optimization using Second Order derivatives and Heteroscedastic Surrogate Model**. We perform extensive empirical experiments on a real-life fog computing testbed to compare and analyze GOSH against state-of-the-art methods. Our experiments show that GOSH performs *best* in terms of QoS metrics, reducing the energy consumption, response time and Service Level Agreement (SLA) violations by up to 18, 27 and 82 percent, respectively. It achieves this by being one of the *first* methods to mitigate volatility by fully taking into account the uncertainty in DNN model predictions. Specifically, the technical contributions of this chapter include the use of a heteroscedastic neural network based surrogate model of the QoS objective score, uncertainty-based exploration and an adapted version of higher-order optimization. An extension of GOSH, namely GOSH*, further improves model predictions by running a discrete-event co-simulator to generate QoS estimates, but at the cost of increasing the scheduling time.

Extending to MCDS. Unlike standalone workloads, workflows constitute several resource intensive and inter-dependent tasks, with their precedence constraints being modeled as a directed acyclic graph (DAG) [Adhikari et al., 2019]. Modern computing offers a new range of computationally heterogeneous workflows involving data gathering at the edge and processing at the cloud backend. Such workflows have recently been adapted to hybrid computing platforms like edge-cloud environments for their cost saving benefits [Ding et al., 2018, Matrouk and Alatoun, 2021]. The workflow scheduling problem aims to efficiently map the tasks onto the available resources to optimize the Quality of Service (QoS) metrics. Many approaches have been proposed in the past (details in Chapter 2). However, such methods are myopic as they optimize short-term objective scores, leading them to often get stuck at a local optimum [Sutton and Barto, 2018, Chapter 7]. Further, such methods often struggle to produce accurate QoS estimates in dynamic settings (with volatile tasks and mobile hosts) to execute apt decision optimization [Song and Tilevich, 2020]. To mitigate the risks of short-term goal optimization and generate a myopic QoS estimate, we develop on some recent ideas. In the past, schedulers have used deep neural networks to approximate the optimization objectives [Ismayilov and Topcuoglu, 2020]. Such neural networks are called deep surrogate models (DSMs) as they act as approximators for the QoS scores that are otherwise time-consuming to obtain (requiring physical execution of the scheduling decisions). Deep surrogate models provide an improved QoS due to their ability to accurately map objective scores and quickly adapt to diverse scenarios. However, due to the poor state-space exploration, such methods can perform poorly in settings unseen during model training. As surrogate models only provide a single-step objective estimate, optimizing using such models can adversely affect long-term QoS scores. Thus, some reinforcement learning methods use look-ahead search schemes to get a long-term score estimate [Sutton and Barto, 2018]. One such method is Monte Carlo learning that runs several multi-step executions to get an unbiased estimate of the long-term effect of an immediate action [Sutton and Barto, 2018]. Running such executions in a real environment is infeasible due to the large execution times. As scheduling decisions need to be taken in near real-time, prior work does not use such look-ahead schemes and hence performs myopic optimization. We use co-simulation, which allows schedulers to leverage event-driven simulators to quickly get a QoS estimate instead of executing decisions on a physical platform. Combining Monte Carlo learning and deep surrogate models for QoS optimal workflow scheduling is non-trivial. This is because vanilla Monte Carlo learning requires the computationally expensive execution of several multi-step simulation runs with random scheduling actions. This may be infeasible when frequent decision making is required, especially for volatile environments [Hausmann et al., 2019]. Further,

traditional deep surrogate models trained on historical data only provide the QoS score of the system state immediately after executing an action and not a long-term QoS. To address these issues, instead of randomly choosing the actions to simulate, we train a deep surrogate model to approximate long-term QoS scores within a Monte Carlo learning framework. To discover potentially better actions, we constantly explore the state-action search space. To start with, we use the action output of a short-term optimization scheme.

Drawing all these insights together, we present MCDS: Monte Carlo Learning using Deep Surrogate models for workflow scheduling. MCDS is an AI augmented scheduler that uses Upper-Confidence-Bound (UCB) exploration [Sutton and Barto, 2018, Chapter 2] and short-term optimization action output as domain knowledge. Running co-simulations on several multi-step decisions allows MCDS to get a long-term QoS estimate of each immediate action choice. Once we have a trained deep surrogate model, we employ a gradient optimization using the back-propagation to input (GOBI) scheme to reach optimal decisions quickly (see Chapter 3). To test the efficacy of the MCDS policy, we perform experiments on a physical mobile edge-cloud setup with ten hosts and a simulated setup with fifty hosts with real-world application workloads. The set of schedulers used in our experiments includes MCDS and four other state-of-the-art baselines. Our evaluations show that MCDS performs *best* in terms of QoS parameters, giving at least 6.13%, 4.56%, 45.09% and 30.71% better energy consumption, response time, SLA violation rate and execution cost respectively. These results are obtained with up to 13.19% lower scheduling time than the baseline with the second-best QoS scores.

6.1 Extending GOBI to Stochastic Neural Models and Higher Order Optimization

To model the variance of the QoS objective score in our system, we characterize its uncertainties into two classes defined below.

Aleatoric Uncertainty. The aleatory uncertainty of a collection of datapoints refers to an inherent variation in the physical system to be modeled, also known as irreducible uncertainty [Shaker and Hüllermeier, 2020].

Epistemic Uncertainty. The epistemic uncertainty of a collection of datapoints refers to the deficiencies caused by a lack of knowledge or information, also known as reducible uncertainty [Shaker and Hüllermeier, 2020].

Both uncertainties are used in GOSH. We now describe the system, workload and the QoS objective models that are used in this work.

6.1.1 System Model

We consider a similar system model as in Chapter 3, with the t -th interval denoted as I_t . Moreover, we consider new tasks being generated at the beginning of every interval. For all tasks in the system in I_t , the broker takes a decision $\mathcal{D}_t$ which includes allocations for new and migrations for existing tasks (with matrix form denoted as $\phi(\mathcal{D}_t)$). The matrix form of a scheduling decision consists of one-hot vectors for each task allocation, each of the size of the number of hosts. Tasks unable to be allocated are kept in a wait queue W_t . The objective score at I_t is denoted as $\mathcal{O}_t$. This objective score quantifies the QoS of the system. The state of the system including the

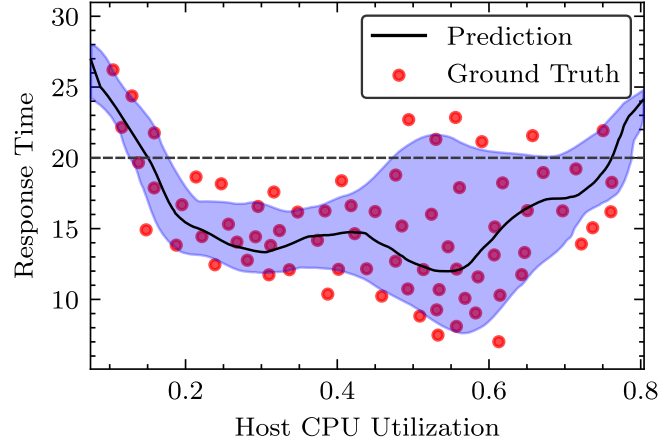


Figure 6.1: Heteroscedastic Surrogate for Response Time.

utilization metrics of hosts and tasks is denoted as S_t . Moreover, we train and use a Long-Short-Term-Memory (LSTM) network to predict the next state S_{t+1} from the previous history of states $\{S_0, \dots, S_t\}$ [Hochreiter and Schmidhuber, 1997]. This is due to the ability of LSTMs to accurately model sequential data like utilization traces [Ouhamme et al., 2021].

Users generate tasks that are relayed via gateway devices to a fog broker. The broker uses the utilization state of the system (S_t) to generate an LCB estimate using a surrogate model. This estimate is then optimized using a second-order gradient optimizer, and the final scheduling decision is executed by the container-orchestrator, discussed in Section 6.1.4. We now describe the neural model used by GOSH to model the QoS score $\mathcal{O}_t$.

As discussed in the introduction, it is crucial to model the complete distribution to characterize the uncertainty in metrics like response time to reduce SLA violations. To achieve this, as discussed in the following subsection, we use deep neural models of NPN due to their superior modeling performance to other methods like Gaussian Processes, BNNs or FCNs. Our optimization paradigm is motivated by energy-based models (EBMs) that use a deep neural network to model an energy function that is minimized using gradient-based optimization [LeCun et al., 2007]. Here, the energy function takes as inputs the observed (like host and workload characteristics) and the decision variables (like a schedule for tasks). We now describe how we model this using NPNs, LCB and second-order optimization methods.

6.1.2 NPN as Heteroscedastic Surrogate Model

Motivation. Predicting just the mean response time using a deterministic surrogate model and ignoring the uncertainty in the distribution is insufficient for an accurate forecast of Service Level Agreement (SLA) violations [Panda et al., 2015]. Moreover, our tests show that in many cases (more than 70%), taking a scheduling decision according to the minimum mean latency (but high uncertainty) gives a higher SLA violation rate than a decision with higher mean latency (but lower uncertainty). Consider the real-world experiments we have carried out in Figure 6.1 where response time (seconds) is plotted against the CPU utilization of a host in a fog environment. Here, based on a dataset (red points), we train a heteroscedastic Gaussian process surrogate model (mean in black and confidence interval shaded). Considering the mean prediction model, there are two minima of the curve, 0.3 (local minimum) and 0.55 (global minimum) CPU utilization of a host. However, the probability of exceeding the SLA deadline (20 seconds) is higher for 0.55 than 0.3. Thus, both mean and

statistical uncertainty need to be modeled to minimize the best objective value that can be guaranteed with some given probability viz the *value at risk* (VaR) [Kochenderfer and Wheeler, 2019, § 17.3].

Natural Parameter Networks. To model uncertainty, recent works have used models like Gaussian Processes (GP) or Bayesian Neural Networks (BNN) [Zinnen and Engel, 2011, Jamshidi and Casale, 2016]. However, prior work has shown Natural Parameter Networks (NPNs) to be more accurate in modeling heteroscedastic uncertainty, also corroborated by our experiments (Section 6.3.5.2). NPNs form a class of probabilistic-neural networks that provide a lightweight Bayesian treatment of traditional neural models [Wang et al., 2016]. To do this, NPNs express neural network parameters as exponential-family distributions, giving a stochastic output of each layer. Such networks are trained using an adapted version of the backpropagation approach [Wang et al., 2016]. Compared to standard BNNs, this allows NPNs to avoid executing time consuming sampling. Typically for an exponential distribution, the output can be characterized with a set of natural parameters (for instance, mean and variance in a normal distribution). Exploiting this, say for mapping a normal distribution, NPNs use twice the parameters of a traditional neural network to output both mean and variance estimates. In contrast to a traditional NN, an input goes through multiple layers of stochastic transformation, producing an output distribution that is matched with a target distribution. A major advantage of NPNs is that they model aleatoric uncertainty more accurately than the more popular Bayesian Neural Networks (BNNs) [Wang et al., 2016]. However, inference using an NPN scales linearly with the dimension of the output [Wang et al., 2016]. Considering a uni-variate objective function, NPNs become a natural choice for stochastic modelling for our problem when compared to BNNs.

Heteroscedastic Modelling. An NPN model cannot be used directly in our setting. This is because, unlike standard NPNs, we do not have the uncertainty information for each input state and scheduling decision. Instead, we have a single datapoint corresponding to each input of the network. To resolve this, we consider the underlying datapoint that corresponds to a probabilistic distribution instead of being static. Specifically, we consider a NPN neural approximator $f(x_t; \theta)$ of the objective score $\mathcal{O}$, with θ as the network parameters and x_t as the state of the fog environment and scheduling decision $[S_t, \phi(\mathcal{D}_t)]$. The output of the NPN is a pair $(\mu_t, \sigma_t) = f(x_t, \theta)$. The parameters θ are learnt using a dataset $\Lambda = \{[S_t, \phi(\mathcal{D}_t)], \mathcal{O}_t\}_{t=0}^T$, and the aleatoric loss $\mathcal{L}$ defined as:

$$\mathcal{L}(f, \Lambda) = \frac{1}{T} \sum_{(x_t, y_t) \in \Lambda} \frac{(\mu_t - y_t)^2}{2\sigma_t^2} + \frac{1}{2} \ln \sigma_t^2, \quad (6.1)$$

where $(\mu_t, \sigma_t) = f(x_t; \theta)$. This is the same as Kullback–Leibler divergence loss between $\mathcal{N}(\mu, \sigma)$ and $\mathcal{N}(y_t, 1)$ [Wang et al., 2016] of the ground-truth datapoint. Using this loss function, we train our NPN model to predict an estimate μ and aleatoric uncertainty σ .

Now, we calculate the *value at risk* using the prediction of the NPN model. The value at risk (VaR) is defined as the best objective value that can be achieved with α probability, which is equivalent to the α quantile of the distribution. Considering a Gaussian distribution and 95% confidence bound as per [Wang et al., 2016, Kochenderfer and Wheeler, 2019, Rezaei et al., 2020], the parameterized VaR for interval I_t is given by

$$VaR_t = \mu + 1.65\sigma. \quad (6.2)$$

Optimizing this metric in GOSH allows it to minimize the mean latency (μ) as well as reduce uncertainty (σ) in response time, leading to a lower probability of overshooting deadline and hence lower SLA violation rates.

6.1.3 LCB Exploration

Motivation. Prior work shows that exploration of the state space based on the uncertainty estimates of heteroscedastic models can provide a performance boost in dynamic scenarios [Jamshidi and Casale, 2016]. In particular, methods like ϵ -greedy, softmax and Lower-confidence bound (LCB) have been popular [Jamshidi and Casale, 2016, Liu et al., 2020]. However, due to the adaptability of LCB exploration, we use LCB to trade off between greedy minimization of the mean latency and error reduction by employing uncertainty-based exploration [Kochenderfer and Wheeler, 2019, § 17.3.4]. However, it is known that uncertainty modeling in such NPNs only includes the aleatoric uncertainty [Sahlin et al., 2021]. Thus, in this work, we use the LCB metric with epistemic uncertainty.

Teacher-Student Network. To implement the LCB exploration strategy, we need estimates of the epistemic uncertainty for each input $[S_t, \phi(\mathcal{D}_t)]$. Prior work leverages models like BNNs or Monte-Carlo Dropout to achieve this by sampling the output multiple times and quantifying the variance in the output to determine the epistemic uncertainty [Postels et al., 2019]. However, the measure of epistemic uncertainty obtained from this method is not in closed form and hence cannot be differentiated. Having closed-form solutions is better for second-order optimization as finite difference methods tend to be slow and imprecise [Kochenderfer and Wheeler, 2019, § 2.3.2], having a negative effect on the scheduling time.

To solve this problem, we employ a teacher-student learning model, similar to the one in [Matiisen et al., 2019], with an overview given in Figure 6.2. Compared to other methods like approximated variance propagation or sampling based propagation [Postels et al., 2019], a teacher-student model is able to adapt quickly to changing scenarios. **The teacher network**, denoted as $g(x_t; \theta')$, is a FCN model that takes $[S_t, \phi(\mathcal{D}_t)]$ as an input and outputs an estimate of $\mathcal{O}_t$. However, in this model, we use Monte-Carlo Dropout (MCD) for Bayesian inference at test time [Gal and Ghahramani, 2016]. Unlike conventional dropout, MCD enables dropout at inference time as well. This allows us to run inference multiple times (say M) and obtain a stochastic output. We can then calculate the standard-deviation of the resultant output samples. This deviation quantifies the epistemic uncertainty (denoted as ξ) of the model as described in [Kendall and Gal, 2017]. In contrast, **the student network**, denoted as $h(x_t; \theta'')$, is another identical FCN model without the MC Dropout. It takes $[S_t, \phi(\mathcal{D}_t)]$ as an input and directly outputs an estimate of the epistemic uncertainty ξ (denoted as $\hat{\xi}$). Unlike the teacher, the output epistemic uncertainty of the student is differentiable, allowing quick and precise second-order optimization. Both teacher (g) and student (h) networks are trained using the Mean Square Error (MSE) loss such that for a dataset Λ as described previously,

$$\mathcal{L}(g, \Lambda) = \frac{1}{T} \sum_{(x_t, y_t) \in \Lambda} (g(x_t; \theta') - y_t)^2 \quad (6.3)$$

$$\mathcal{L}(h, \Lambda) = \frac{1}{T} \sum_{(x_t, y_t) \in \Lambda} (h(x_t; \theta'') - \xi_t)^2. \quad (6.4)$$

Assuming we use affine activations, the student-network is a differentiable closed-form function; its output $\hat{\xi}$ is differentiable with respect to the input.

Now that we have the VaR from the NPN model and an estimate of the epistemic uncertainty, we use the

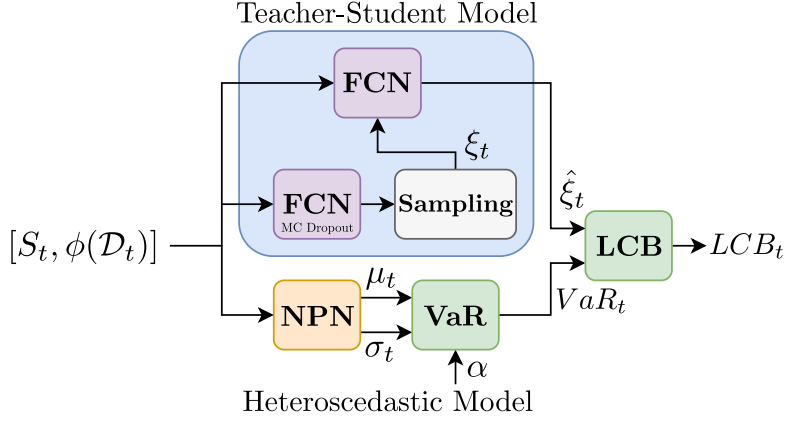


Figure 6.2: LCB metric calculation in GOSH.

LCB optimization where the final objective function to minimize becomes

$$LCB_t = \underbrace{VaR_t}_{\text{exploitation}} - \underbrace{k \cdot \hat{\xi}_t}_{\text{exploration}}, \quad (6.5)$$

where k is the exploration factor.

Minimizing the first term aims to reduce the chance of an SLA violation (considering response time as part of $\mathcal{O}_t$) and can be termed the *exploitation* part of the algorithm. Minimizing the second term, *i.e.*, maximizing the epistemic uncertainty helps the model *explore* in cases with limited data or knowledge. This interactive dynamic between the exploration and exploitation allows the GOSH scheduler to quickly adapt while maintaining low SLA violation rates.

6.1.4 Second-Order Optimization

We now describe how we optimize the scheduling decision. To reduce the number of iterations, avoid saddle points and shorten the scheduling time, we employ a second-order stochastic optimization approach which dynamically incorporates the curvature of the surrogate function [Yao et al., 2021]. Exploiting the improved convergence properties of the second order algorithms over first order ones, our approach gives *low scheduling times* and *improved QoS objective scores*.

For a trained neural approximator f , at the start of each interval I_t , a typical second-order approach (Newton methods) optimizes the decision matrix $\phi(\mathcal{D})$ by the following rule (γ is the learning rate)

$$\phi(\mathcal{D}_{n+1}) \leftarrow \phi(\mathcal{D}_n) - \gamma \cdot [\nabla_{\phi(\mathcal{D}_n)}^2 f]^{-1} \cdot \nabla_{\phi(\mathcal{D}_n)} f,$$

where f is a shorthand for $f([S_t, \phi(\mathcal{D}_n)]; \theta)$. This operation is iteratively performed until the absolute value of the gradient is more than the convergence threshold ϵ .

Computing the second-order derivative, or Hessian in a multivariate case, allows us to make a quadratic approximation of the objective function and estimate an appropriate step size to reach a local minimum [Kochenderfer and Wheeler, 2019, § 6]. For instance, it is known that if f is quadratic and its Hessian is positive definite, one can reach its global minimum in a single step using the Newton method. Moreover, second-order methods provide resilience against ill-conditioned objective function landscapes and guarantee convergence to second-

order critical points [Yao et al., 2021].

However, computing the inverse of the Hessian is known to be computationally expensive (cubic time-complexity with matrix size). In this endeavor, prior work proposed Hessian free approaches like secant or Quasi-Newton methods [Kochenderfer and Wheeler, 2019, § 6.3]. Moreover, it has been shown that the exact computation of Hessian is not required and an approximate of the diagonal of the Hessian is sufficient for low-cost iterations [Martens and Grosse, 2015]. One such method for computing this Hessian diagonal for neural networks is the Hutchinson’s method with a linear computational complexity [Yao et al., 2018]. However, even an approximate Hessian can give local curvature information, which is misleading. This is alleviated by adapting the Hutchinson’s method and using the moving average of the approximate Hessian diagonal to speed up convergence. Thus, for Hutchinson’s estimate of the Hessian diagonal of $f([S_t, \phi(\mathcal{D}_n)]; \theta)$ at I_t , denoted as $H_t(f)$, we compute the moving average $\hat{H}_t(f)$ as

$$\hat{H}_t(f) \leftarrow \omega \cdot \hat{H}_t(f) + (1 - \omega) \cdot H_t(f),$$

where ω is the momentum parameter. Hence, the update equation becomes

$$\phi(\mathcal{D}_{n+1}) \leftarrow \phi(\mathcal{D}_n) - \gamma \cdot \hat{H}_t^{-1}(f) \cdot \nabla_{\phi(\mathcal{D}_n)} f. \quad (6.6)$$

Unlike prior efforts that utilize Hutchinson’s method for lightweight second-order optimization like [Yao et al., 2021, 2020], we perform iteration based moving averages and not spatial averaging. This allows our model to be robust to sudden variations in reward metrics in volatile workload settings. The main overhead of this approach is the computation of the approximate of the Hessian diagonal using Hutchinson’s method. Also, it is possible to improve the estimate of the Hessian diagonal by increasing the iteration count of Hutchinson’s method. However, one-step iteration performs as well in practice, giving low per-iteration computational overhead compared to the GOBI approach [Yao et al., 2021]. Comparisons of the scheduling overheads are shown later in Section 6.3.5.2.

6.1.5 The GOSH and GOSH* Schedulers

In the GOSH approach, we combine the LCB metric given by (6.5) and second-order optimization in (6.6). Thus, our update equation becomes

$$\phi(\mathcal{D}_{n+1}) \leftarrow \phi(\mathcal{D}_n) - \gamma \cdot \hat{H}_t^{-1}(LCB_t) \cdot \nabla_{\phi(\mathcal{D}_n)} LCB_t, \quad (6.7)$$

where $\hat{H}_t(LCB_t)$ denotes the moving average of the Hessian diagonal estimate of LCB_t . As we utilize the output of the student network for the epistemic uncertainty, we can generate a closed-form second-order Hessian estimate (see Figure 6.2).

To optimize the QoS parameters, we consider a standard objective function that focuses on energy consumption and response time as is done in prior work [Basu et al., 2019]. For interval I_t ,

$$\mathcal{O}_t = \alpha \cdot AEC_t + \beta \cdot ART_t, \quad (6.8)$$

where AEC_t and ART_t are the average energy consumption of the fog environment in I_t and average response

time of tasks leaving at the end of I_t . Here, α and β (such that $\alpha + \beta = 1$) are hyper-parameters that can be set by users in line with the application requirements. Typically, in real-life settings, these values are around 0.5. For an energy-constrained setting, a higher α value is used, whereas for latency-critical tasks, a higher β value is used.

We also extend GOSH to benefit from co-simulated results. GOSH* uses a simulator to execute the scheduling decision predicted by GOSH and the utilization metrics of the next state using an LSTM (Section 6.1). This allows GOSH* to make more informed decisions and hence reach better objective scores than GOSH, spending additional computational overhead in return for increased accuracy. We will show that GOSH is more suitable for environments with resource-constrained fog brokers, whereas GOSH* reaches a better objective score but requires heavier computational resources.

6.1.6 Model Training

As described previously in Section 6.1.2, the models f, g and h are trained using a dataset $\Lambda = \{[S_t, \phi(\mathcal{D}_t)], \mathcal{O}_t\}_{t=0}^T$ as discussed in Section 6.1. The model f is trained using the aleatoric loss (6.1), whereas g, h are trained using MSE loss given in (6.3) and (6.4). We train using the AdamW optimizer [Loshchilov and Hutter, 2018]. Using these models, we obtain the LCB metric (6.5).

For GOSH*, we train models f^*, g^* and h^* which take input $[S_t, \bar{\mathcal{O}}_t, \phi(\mathcal{D})]$, where $\bar{\mathcal{O}}_t$ is a simulated estimate of $\mathcal{O}_t$. We also train a next-state predictor using a LSTM network (Section 6.1). This model uses the state history to predict the next state, where the predicted state is denoted as

$$\bar{S}_t \leftarrow LSTM(\{S_{t'} \forall t' < t\}).$$

This LSTM model is also trained using the Λ dataset.

6.1.7 Dynamic Exploration Weight

As we show in Section 6.3.5.2, keeping a static value of the exploration weight k is not ideal as the objective scores are very sensitive to its value. Hence, GOSH and GOSH* use a dynamic exploration weight k , which is updated based on the epistemic uncertainty value in each scheduling interval. Thus, we update the k parameter such that it increases when the epistemic uncertainty increases, indicating more exploration required and decreases otherwise, while at the same time remaining fairly insensitive to the step-size parameter.

We use a feedback based weight control scheme similar to the one in [Maroti, 2019]. At run-time, we maintain an exponential moving average of the epistemic uncertainty metric ξ (denoted as ξ_{MA}). Thus, at each interval I_t , $\xi_{MA} \leftarrow \psi \cdot \xi_{MA} + (1 - \psi) \cdot \xi_t$, where ψ is the exploration momentum parameter. Now, the value of k is updated as per the rule

$$k = \begin{cases} (1 + \delta) \cdot k, & \text{if } \xi > (1 + \delta) \cdot \xi_{MA} \\ (1 - \delta) \cdot k, & \text{if } \xi < (1 + \delta) \cdot \xi_{MA} \\ k, & \text{otherwise,} \end{cases} \quad (6.9)$$

where δ is the step-size.

Algorithm 6.1 The GOSH scheduler**Require:**

Pre-trained stochastic function approximator $f(x; \theta)$
 Pre-trained Teacher-Student models $g(x; \theta'), h(x; \theta'')$
 Dataset used for training Λ ; Convergence threshold ϵ
 Iteration limit L ; Learning rate γ ; LCB parameter k
 Value at Risk parameter α

```

1: procedure MINIMIZE( $\mathcal{D}$ ,  $f$ ,  $g$ ,  $h$ ,  $z$ )
2:   Initialize decision matrix  $\phi(\mathcal{D})$ ;  $i = 0$ 
3:   do
4:      $x \leftarrow [z, \phi(\mathcal{D})]$  ▷ Concatenation
5:      $\mu, \sigma \leftarrow f(x; \theta)$  ▷ Stochastic Prediction
6:      $\hat{\xi} \leftarrow h(x; \theta'')$  ▷ Uncertainty Prediction
7:      $LCB \leftarrow VaR(\mu, \sigma, \alpha) - k \cdot \hat{\xi}$  ▷ LCB metric
8:      $\phi(\mathcal{D}) \leftarrow \phi(\mathcal{D}) - \gamma \cdot \hat{H}^{-1}(LCB) \cdot \nabla_{\phi(\mathcal{D})} LCB$ 
9:      $i \leftarrow i + 1$ 
10:  while  $|\hat{H}^{-1}(LCB) \cdot \nabla_{\phi(\mathcal{D})} LCB| > \epsilon$  and  $i \leq L$ 
11:  Convert matrix  $\phi(\mathcal{D})$  to scheduling decision  $\mathcal{D}^*$ 
12:  return  $\mathcal{D}^*$ 
13: end procedure
14: procedure INITIALIZE(scheduling interval  $I_t$ )
15:   if ( $t == 0$ )
16:     return random decision  $\mathcal{D}$ 
17:   return  $\mathcal{D}_{t-1}$  ▷ Output for the previous interval
18: end procedure
19: procedure TUNE(scheduling interval  $I_t$ )
20:    $x_t \leftarrow [S_t, \phi(\mathcal{D}_t)]$ 
21:    $\xi_t \leftarrow Sample(g(x_t; \theta'))$ 
22:    $k, \xi_{MA} \leftarrow Update(\xi, \psi)$ 
23:    $\Lambda_{t-1} \leftarrow \{(x_{t-1}, \mathcal{O}_{t-1})\}$ 
24:   Fine-tune  $\theta, \theta'$  with  $\Lambda_{t-1}$  and loss as in (6.1), (6.3)
25:   Fine-tune  $\theta''$  with  $\xi_t$  and loss as in (6.4)
26: end procedure
27: procedure GOSH(scheduling interval  $I_t$ )
28:    $\mathcal{D} \leftarrow INITIALIZE(I_t)$ 
29:   Get  $S_t$ 
30:    $\mathcal{D}_t \leftarrow MINIMIZE(\mathcal{D}, f, g, h, S_t)$ 
31:   TUNE( $I_t$ )
32:   return  $\mathcal{D}_t$ 
33: end procedure

```

6.1.8 Scheduling

An overview of the GOSH scheduler is given in Algorithm 6.1. After training models f, g and h and obtaining LCB metric, we optimize $\phi(\mathcal{D}_t)$ as per the rule (6.7) until the step-size $\hat{H}_t^{-1}(LCB_t) \cdot \nabla_{\phi(\mathcal{D}_t)} LCB_t$ is greater than the convergence threshold ϵ (line 10). We also use Implicit Gradient Regularization in the $\phi(\mathcal{D}_t)$ optimization steps for numerical stability in high aleatoric uncertainty cases [Barrett and Dherin, 2020]. This adds a loss term that imposes structural constraints to the gradient values making them more robust to noisy parameter perturbations [Barrett and Dherin, 2020].

For interval I_0 , we initialize a random decision and for $I_t, t > 0$ we initialize the decision to be the same as $\mathcal{D}_{t-1}$ (lines 16-17). We obtain the current state of the system which includes the IPS, RAM, Disk and Bandwidth utilizations of the hosts and tasks as S_t (line 29). We utilize the MINIMIZE function, given in Algorithm 6.1, to find the decision at each interval I_t (line 30). We then fine-tune the models f, g using the latest datapoint

Algorithm 6.2 The GOSH* scheduler**Require:**

Pre-trained stochastic function approximator $f^*(x; \theta)$
 Pre-trained Teacher-Student models $g^*(x; \theta'), h^*(x; \theta'')$
 Pre-trained LSTM model $LSTM(\{S_{t'} \forall t' < t\})$

```

1: procedure GOSH*(scheduling interval  $I_t$ )
2:    $\mathcal{D} \leftarrow \text{INITIALIZE}(I_t)$ 
3:   Get  $S_t$ 
4:    $\bar{S}_t \leftarrow LSTM(\{S_{t'} \forall t' < t\})$ 
5:    $\bar{\mathcal{D}}_t \leftarrow GOSH(I_t)$ 
6:    $\bar{\mathcal{O}}_t \leftarrow \mathcal{S}(\bar{\mathcal{D}}(A_t), \bar{S}_t)$  ▷ Simulation
7:    $\mathcal{D}_t \leftarrow \text{MINIMIZE}(\mathcal{D}, f^*, g^*, h^*, [S_t, \bar{\mathcal{O}}_t])$ 
8:   TUNE( $I_t$ )
9:   return  $\mathcal{D}_t$ 
10: end procedure

```

$(x_{t-1}, \mathcal{O}_{t-1})$ (line 24). Finally, we sample and obtain the epistemic uncertainty from g as ξ_t and fine-tune h using MSE loss (line 25). This continuous fine-tuning allows the models to adapt in dynamic scenarios.

An overview of the working GOSH* scheduler is given in Algorithm 6.2. Here, we employ GOSH, next-state predictor (LSTM model) and a simulator (denoted as $\mathcal{S}$). Using the predicted next-state $\bar{S}_t$ and an action of interest $\bar{\mathcal{D}}_t = GOSH(I_t)$, we use the simulator to predict the objective score at the end of the current interval $\bar{\mathcal{O}}_t \leftarrow \mathcal{S}(\bar{\mathcal{D}}(A_t), \bar{S}_t)$ (line 6).

6.2 Extending GOBI to Long-Term Optimization

6.2.1 Workload Model

We assume scheduling as a discrete-time control problem as standard in prior work [Mahmud et al., 2018, Basu et al., 2019]. We assume that there are a fixed number of host machines, the set of which is denoted as H . Each workflow is modeled as a directed acyclic graph $G = (V, E)$, where V denotes the tasks as nodes and E denotes the execution precedence constraints as directed edges. Each edge (i, j) in E is an ordered pair such that task i must complete execution before task j is scheduled. We also consider that new workflows created at the interval I_t are denoted as N_t , with all active workflows being denoted as A_t . A workflow is considered to be active if at least one task of that workflow is being executed in the edge-cloud environment. If no task of a workflow $G \in N_t$ can be allocated to an edge or cloud node then it is added to a wait queue W_t . All created workflows that are not active and are not in the wait queue are considered to be completed workflows and we can calculate their QoS metrics like response time and SLA violation.

6.2.2 Problem Formulation

We denote the QoS objective score for interval I_t as O_t that we need to maximize. We denote the utilization metrics of all hosts in the interval I_{t-1} as U_t . We collect the DAGs of all active workflows A_{t-1} and denote them as G_t (to represent the workflow constraint graph). Now using U_t and G_t , we need to predict a scheduling decision S_t . All tasks with no incomplete dependent tasks in $N_t \cup W_t \cup A_t$ are called feasible tasks. G_t includes allocation decisions for feasible and migration decisions for active tasks in the system. Thus the problem can

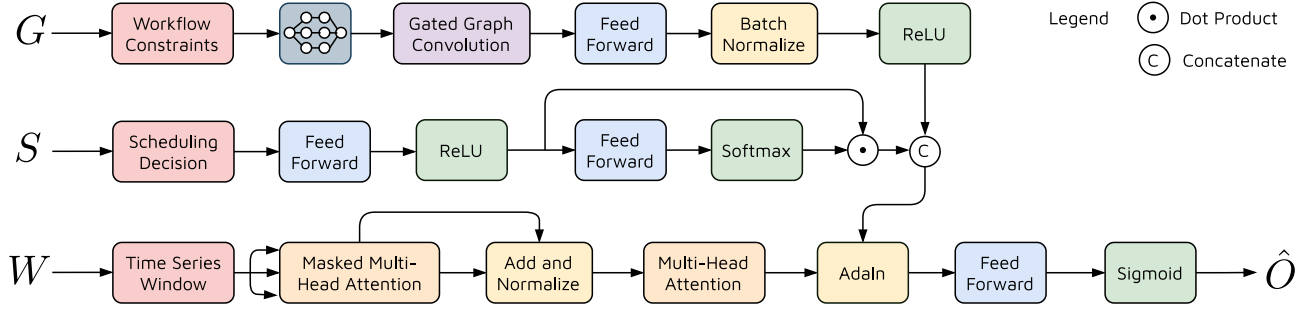


Figure 6.3: Deep Surrogate Model used in the MCDS Scheduler.

be formulated as:

$$\begin{aligned}
 & \underset{S_t}{\text{maximize}} \quad \sum_{t=0}^{\infty} O_t \\
 & \text{subject to} \quad \forall t, S_t : P_t \cup Q_t \rightarrow H, \\
 & \quad \forall t, P_t = \text{set of feasible tasks in } N_t \cup W_t \cup A_t, \\
 & \quad \forall t, Q_t = \text{set of active tasks in the system.}
 \end{aligned} \tag{6.10}$$

6.2.3 Deep Surrogate Model

Our MCDS scheduler uses a deep neural network as a surrogate model. The MCDS scheduler lies in the edge-cloud broker. We first describe the working of this model and how we use it to obtain an estimate of the objective score. To do this, we encapsulate the state of the system using the host characteristics U_t and the workload constraints G_t . To predict an estimate of the objective score, we provide the model with the system state and the scheduling decision. Instead of using U_t , we use a time-series sliding window of size k , denoted as W_t , to capture temporal trends of the host utilization characteristics. Thus, at the start of the interval I_t , the input to the surrogate model is $[G_t, S_t, W_t]$. Without loss in generality, we drop the subscript t for the rest of the discussion. An overview of the deep neural network used as a surrogate model in the MCDS scheduler is shown in Figure 6.3. The constituent model components are detailed next.

6.2.4 Workflow Constraint Graph Encoding

To generate a QoS estimate, we encode the workflow constraint graph G using a gated-graph convolution network (GGCN). This is because GGCN has been shown to be effective in capturing dynamically changing graph like inputs [Li et al., 2015]. We first form an embedding for each node in the graph; for active tasks we use the SLA deadline and the resource utilization characteristic and for inactive tasks we use the zero-vector. We denote this embedding for a task i as e_i . Following Chapter 2, the gating stage is realized as a Gated Recurrent Unit (GRU) resulting in *graph-to-graph* updates as:

$$\begin{aligned}
 r_i^0 &= \text{Tanh}(w e_i + b), \\
 x_i^k &= \sum_{j \in n(i)} w^k r_j^{k-1}, \\
 r_i^k &= \text{GRU}(r_i^{k-1}, x_i^k).
 \end{aligned} \tag{6.11}$$

Here, w and b are the weight matrix and bias parameter of the neural network and k is the iteration index that varies from 1 to p , each corresponding to a graph convolution operation. Further, the messages for task i are aggregated over one-step connected neighbors $n(i)$ over p convolutions, resulting in an embedding r_i^p for each task node in the graph. The stacked representation for all tasks is represented as r^p and is used in the downstream model. We generate the workflow graph encoding e^G by passing r^p through a feed-forward layer with ReLU activation as

$$e^G = \text{ReLU}(\text{BatchNorm}(\text{FeedForward}(r^p))), \quad (6.12)$$

where BatchNorm standardizes the output of the FeedForward layer for the input batch.

6.2.5 Scheduling Decision Encoding

We also encode the scheduling decision for p tasks, first as a matrix M^S and then pass it through a feed-forward layer. Then we apply *Bahdanau style* self-attention [Bahdanau et al., 2015] to reduce its dimensions size and generate the scheduling decision encoding e^S . Thus,

$$\begin{aligned} M &= \text{ReLU}(\text{FeedForward}(M^S)), \\ \text{attn} &= \text{Softmax}(\text{FeedForward}(M)), \\ e^S &= \sum_{i=1}^p \text{attn}_i \cdot M_i. \end{aligned} \quad (6.13)$$

Here, M_i is the one-hot vector corresponding to the i -th tasks out of p tasks in the system. This operation takes a convex combination of the multiple scheduling decisions to create a single encoding that encapsulates the inter-task dependencies. This relieves the downstream predictors from discriminating among scheduling decisions.

6.2.6 Time Series Window Encoding

For any three input matrices Q , K and V , we define Multi-Head Self Attention [Vaswani et al., 2017] as passing it through h (number of heads) feed-forward layers to get Q_i , K_i and V_i for $i \in \{1, \dots, h\}$, and then applying attention as

$$\begin{aligned} \text{MultiHeadAtt}(Q, K, V) &= \text{Concat}(H_1, \dots, H_h), \\ \text{where } H_i &= \text{Attention}(Q_i, K_i, V_i), \end{aligned} \quad (6.14)$$

where $\text{Attention}(Q_i, K_i, V_i)$ is the scaled-dot product attention operation [Vaswani et al., 2017]. For a time-series input of host resource utilization characteristics, we perform transformer like multi-head attention operations as:

$$\begin{aligned} W^1 &= \text{Mask}(\text{MultiHeadAtt}(W, W, W)), \\ W^2 &= \text{LayerNorm}(W + W^1), \\ e^W &= W^2 + \text{MultiHeadAtt}(W^2, W^2, W^2). \end{aligned} \quad (6.15)$$

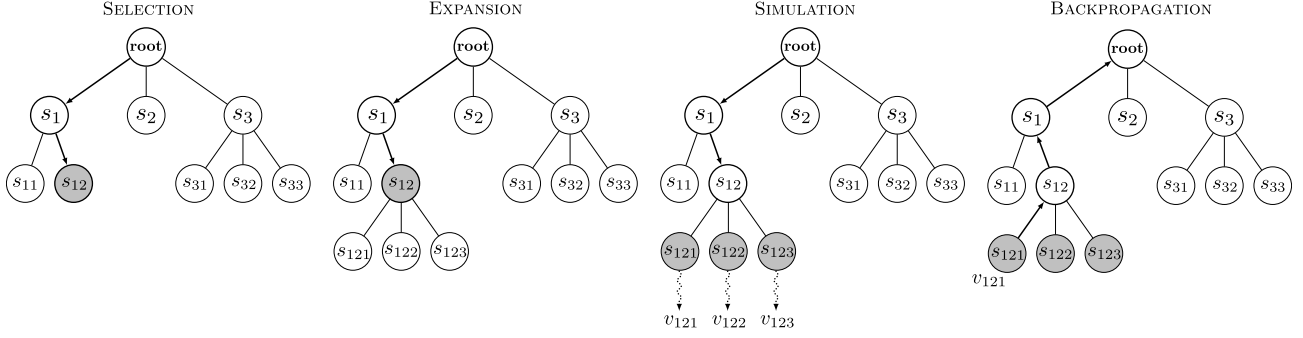


Figure 6.4: Building the Monte Carlo Tree.

The first attention operation is masked to prevent the encoder from looking at the datapoints for future timestamps at the time of training as all time-series windows are given at once to allow parallel training. LayerNorm, unlike BatchNorm, performs normalization across the values in the input instead of the batch. Multi-head attention allows the model to make the subsequent prediction computationally simpler, enabling higher model scalability and generalizability [Lee et al., 2019]. The final window encoding obtained is e^W .

6.2.7 Estimating the Objective Score

Now that we have encoded representations of the host characteristics (e^W), scheduling decision (e^S) and the workflow tasks (e^G), we use Adaptive Instance Normalization (AdaIn) [Huang and Belongie, 2017] to normalize our window encoding as:

$$\begin{aligned} e^{GS} &= [e^G; e^S], \\ e &= \text{AdaIn}(e^W, e^{GS}). \end{aligned} \tag{6.16}$$

Here, e^{GS} is the concatenation of e^G and e^S . Motivated from style-transfer neural networks, AdaIn adaptively normalizes the time-series encoding using the workflow and scheduling decision encodings. This allows the model to maintain robustness in scenarios with highly time-varying host characteristics. The final encoding is then passed through a feed-forward layer to generate the objective score estimate

$$\hat{O} = \text{Sigmoid}(\text{FeedForward}(e)). \tag{6.17}$$

Unlike traditional deep surrogate model based schedulers, the output of our model is an estimate of the long-term objective score instead of the short-term (single scheduling interval) score. This allows the scheduler to avoid local optima when making sequential scheduling decisions [Sutton and Barto, 2018]. We denote this surrogate model as a function $f(G, W, S) = \hat{O}$. We use a Monte Carlo approach to train this model as described next.

6.2.8 Monte Carlo Learning

We develop a modified version of the Monte Carlo Tree Search (MCTS) to train our deep surrogate model and also actively take scheduling decisions. MCTS is a method of building a search tree considering the current state as the root node [Silver and Tesauro, 2009]. The nodes in this tree denote states of the system and edges

denote actions. Each node is assumed to be either a leaf node or have one or more child nodes corresponding to the feasible actions from this node¹. For our Monte Carlo tree, we consider that each node is defined by the following set of variables: $[G, W, S, q, v, n]$. Here, G , W and S represent the workflow constraints, time-series window and last scheduling decision; q, v, n denote the short-term reward, *value function* and the number of visits to the node. The value function is the long-term estimate of the QoS for the node. Building this tree involves finding the feasible actions from a state, executing several simulation steps and iteratively updating the root node with the best action.

6.2.9 Building the Monte Carlo Tree using Co-Simulations

The process of building the Monte Carlo tree involves iteratively performing four operations: Selection, Expansion, Simulation and Backpropagation. We start with only the root node (initial state), with q, v initialized as 0 and n initialized as 1. An overview of these operations is presented in Figure 6.4.

Selection. This operation selects a leaf-node of the tree. Starting at the root node, we recursively select the child node that has the highest Upper-Confidence-Bound (UCB). For the current node $[G, W, S, q, v, n]$, we select one of the child nodes $\{[G_i, W_i, S_i, q_i, v_i, n_i]\}_i$ using the rule [Russell and Norvig, 2009]

$$\text{selected node} = \arg \max_i v_i + \sqrt{\frac{c_1 \times \ln n}{n_i}}. \quad (6.18)$$

Here the first term corresponds to the exploitation part of the search algorithm and the second term corresponds to exploration. $c_1 \in [0, 1]$ is the exploration parameter. The n value of each node that is selected in this recursive process is incremented by 1. When only the root node is present in the tree, we return that as the selected node.

Expansion. After the selection process, we reach a leaf node of the tree. We use a co-simulator to find out all possible actions $\{S_i\}_i$ from this node and expand it to generate a new node for each such decision. The co-simulator also provides us the G and W for the new nodes. For each new node in the tree, v, n are initialized as 0, 1, respectively. The q value is set as the QoS objective score returned by the co-simulator for this decision in one scheduling interval.

Simulation. For the new nodes in our tree, we use the co-simulator to execute a sequence of ϕ actions to get a long-term QoS estimate of this node, where ϕ is called the roll-out parameter. This QoS value is set as the v value of this node. While training, the actions used in these simulations are based on a random scheduler. At test time, we use the deep surrogate model based scheduler. More details on this in Sections 6.2.10 and 6.2.11.

Backpropagation. Now that we have the updated v values of the new leaf nodes, we backpropagate these up to the root for updating the value functions. For every node $([G, W, S, q, v, n])$ other than the leaf-nodes in the path to the root node with child nodes $\{[G_i, W_i, S_i, q_i, v_i, n_i]\}_i$, we propose the weighted update rule

$$v = q + \frac{\sum_i n_i \cdot v_i}{\sum_i n_i}, \quad (6.19)$$

i.e., the sum of q and the weighted average of the v values of child nodes. The q value here gives us the immediate reward and weighted-average gives us the estimate of the long term reward once the action corresponding to

¹In our discussion, we use action and scheduling decision interchangeably.

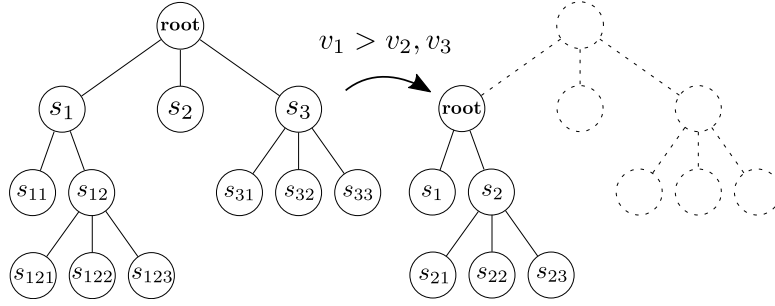


Figure 6.5: Choosing best action and updating the Monte Carlo Tree after simulations.

Algorithm 6.3 Training the deep surrogate model**Require:**Number of simulation steps ψ ; Rollout parameter ϕ ; Dataset used for training Λ ; Deep Surrogate Model f

```

1: Initialize root node
2: procedure MCTS-TRAIN(root)
3:   for  $i \leftarrow 0$  to  $\psi$  do
4:     Recursively select child nodes using Eq. (6.18)
5:     Expand leaf-node  $\forall$  possible actions
6:     Using random policy, run  $\phi$ -step co-simulations
       to generate QoS scores
7:     Recursively back-propagate scores using Eq. (6.19)
8:   end procedure
9: procedure UPDATE(root)
10:   $[G, W, S, q, v, n] \leftarrow \text{root}$ 
11:   $\{[G_j, W_j, S_j, q_j, v_j, n_j]\}_j \leftarrow \text{child nodes of root}$ 
12:   $i = \arg \max_j v_j$ 
13:  Add  $(f(G, W, S), O_t + v_i)$  to  $\Lambda$ 
14:   $\text{root} \leftarrow [G_i, W_i, S_i, q_i, v_i, n_i]$ 
15:  return  $S_i$ 
16: end procedure
17: procedure TRAIN(scheduling interval  $I_t$ )
18:   MCTS-TRAIN(root)
19:    $S_i \leftarrow \text{UPDATE}(\text{root})$ 
20:   Tune  $f$  using the dataset  $\Lambda$  using (6.20) as loss
21:   return  $S_i$ 
22: end procedure

```

this node is taken [Russell and Norvig, 2009].

We perform the four operations: selection, expansion, simulation and backpropagation a fixed number of times (ψ) to generate the Monte Carlo Tree. The expansion operation adds possibly one or more nodes to the tree in each iteration. The UCB metric in the selection phase trades off between exploration of new action choices and exploitation of value function estimates in each iteration. As ψ increases, the value function estimates become closer to the expected empirical objective scores.

6.2.10 Model Training

To train the deep surrogate model, at every interval I_t we build the tree as described in Section 6.2.9 as shown by the MCTS-TRAIN function in Algorithm 6.3. (we start with a single node in I_0). We use a random scheduler to execute the *simulation* operation. The random scheduler gives us an unbiased estimate of the long-term value function of a leaf-node (line 4 in Alg. 6.3). After performing the operations ψ number of times, we choose the action S_i corresponding to child node of the root with the highest value-function v_i (line 12 in Alg. 6.3). For

Algorithm 6.4 MCDS Scheduler**Require:**

Number of simulation steps ψ ; Learning rate γ ;
 Convergence threshold ϵ ; Iteration limit L
 Pre-trained Deep Surrogate Model f

- 1: Initialize *root* node
- 2: **procedure** MCTS-TEST(*root*)
- 3: **for** $i \leftarrow 0$ to ψ **do**
- 4: Recursively select child nodes using Eq. (6.22)
- 5: Expand leaf-node $\forall$ possible actions
- 6: Use $f(G, W, S)$ as generate QoS scores
- 7: Recursively back-propagate scores using Eq. (6.19)
- 8: **end procedure**
- 9: **procedure** GOBIGRAPH(f, G, S, W)
- 10: Initialize $i = 0$
- 11: **do**
- 12: $S \leftarrow S + \gamma \cdot \nabla_S f(G, W, S)$
- 13: $i \leftarrow i + 1$
- 14: **while** $|\nabla_S f(G, W, S)| > \epsilon$ and $i \leq L$
- 15: **return** S
- 16: **end procedure**
- 17: **procedure** MCDS(scheduling interval I_t)
- 18: MCTS-TEST(*root*)
- 19: $S_i \leftarrow \text{UPDATE}(\text{root})$
- 20: Tune f with the new datapoint using (6.20) as loss
- 21: **return** S_i
- 22: **end procedure**

training, instead of using the q_i value of this node, we get the QoS score in the next interval I_{t+1} *i.e.* O_t and add the datapoint $(f(G, W, S), O_t + v_i)$ to the dataset (line 13 in Alg. 6.3). For the next interval, we update the Monte Carlo tree by changing the root node to the selected child node (see Figure 6.5 and UPDATE function in Alg. 6.3). This allows us to re-use co-simulation data in the subsequent scheduling intervals. The MCTS-TEST function recursively selects child nodes. Assuming GOBIGRAPH takes unit time, the MCTS-TEST function has a time and space complexity of $O(\phi)$ each, where ϕ is the maximum depth of the tree as we limit the MCTS iterations by ϕ .

Using the collected dataset, we use experience-replay [Rolnick et al., 2019] to decouple training from the execution of the workloads on our edge-cloud setup. We update the weights of the deep surrogate model using the loss:

$$\text{Loss} = \text{MSE}(f(G, W, S), O_t + v_i), \quad (6.20)$$

where MSE denotes mean-square-error and G, W are the workflow constraints and time-series host characteristics of the root node state. We use a cross-validation style training process with early-stopping as the convergence criterion.

6.2.11 Scheduling using the Trained Model

Now that we have trained the deep surrogate model, for an input $[G, W, S]$, we use the GOBI technique introduced in Chapter 3 to update S to maximize the output $f(G, W, S)$ using gradient-ascent rule

$$S \leftarrow S + \gamma \cdot \nabla_S f(G, W, S). \quad (6.21)$$

To differentiate our workflow-graph based surrogate model from the vanilla feed-forward neural network, we call our optimization strategy as GOBIGRAPH that returns an action in the form of a scheduling decision (see GOBIGRAPH function in Alg. 6.4). At test time we build the Monte Carlo Tree with slight modifications. In the selection operation, we choose the node using the rule

$$\begin{aligned} \text{selected node} = \arg \max_i v_i + \sqrt{\frac{c_1 \times \ln n}{n_i}} \\ + c_2 \times \mathbb{1}(S_i = \text{GOBIGRAPH}(f, G, S, W)). \end{aligned} \quad (6.22)$$

Here, the third term uses the deep surrogate model to bias our selection choice, like a domain knowledge term commonly used in machine learning [Silver and Tesauro, 2009]. $c_2 \in [0, 1]$ is the bias parameter. When performing the simulation operation, we use the value estimate $v = f(G, W, S)$ instead of running expensive co-simulations, reducing the overall scheduling time (line 6 in Alg. 6.4). Using the new datapoint in each interval, we also fine-tune the deep surrogate model f in MCDS to adapt in dynamic settings (line 20 in Alg. 6.4).

6.3 Performance Evaluation

We now provide details on how we evaluate the proposed schedulers against state-of-the-art methods.

6.3.1 Evaluation Setup

We use two-test beds to test the scheduling strategies: physical setup and a simulated platform. The former allows us to perform a precise comparison closer to the real-deployments, while the latter lets us perform large-scale experiments. This is a standard testing mechanism in prior work [Basu et al., 2019].

Physical Testbed: We use Microsoft Azure virtual machines (VMs) to create a heterogeneous PDC edge-cloud testbed². Specifically, we use the B2s, B4ms and B8ms machine types. The configuration consists of 10 VMs, 6 light-weight VMs on the edge and 4 powerful ones on a geographically distant cloud location. All machine types use a different number of Intel Haswell 2.4 GHz E5-2673 v3 processor cores. To keep storage costs consistent, we keep Azure P15 Managed disk with 125 MB/s disk throughput and 256 GB size³. We use the Microsoft Azure pricing calculator to obtain the cost of execution per hour (in US Dollars)⁴. The power consumption models are taken from the SPEC benchmarks repository⁵. The Million-Instruction-per-Second (MIPS) of all VMs are obtained using the `perf-stat`⁶ tool.

The Azure B2s machines consist of two cores (4029 MIPS), B4ms have four (8102 MIPS) and B8ms have eight cores (1601 MIPS) each. Moreover, we have 4GB, 16GB and 32GB RAM sizes in B2s, B4ms and B8ms machines, respectively. In our settings, we have considered the response time of edge nodes to be 1 ms and that of cloud nodes to be 10 ms based on the empirical studies using the Ping utility in an existing edge-cloud framework, namely FogBus [Tuli et al., 2019]. To factor in the possible mobility of the edge nodes, we use the `NetLimiter` tool to tweak the communication latency with the broker node using the mobility model described

²Azure general purpose B-type VMs: <https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable>

³Azure Managed Disk types <https://docs.microsoft.com/en-us/azure/virtual-machines/disks-types#premium-ssd>

⁴Microsoft Azure pricing calculator for South UK <https://azure.microsoft.com/en-gb/pricing/calculator/>

⁵Standard Performance Evaluation Corporation (SPEC) benchmark repository https://www.spec.org/cloud_iaas2018/results/

⁶`perf-stat` tool <https://man7.org/linux/man-pages/man1/perf-stat.1.html>

in [Tuli, 2022]. Specifically, for the edge nodes in our setup, we use the latency and bandwidth parameters of hosts from the traces generated using the Simulation of Urban Mobility (SUMO) tool [Krajzewicz et al., 2012] that emulates mobile vehicles in a city like environment. SUMO gives us the parameters like `ping` time and network bandwidth to simulate in our testbed using `NetLimiter`. The edge-cloud broker node was a system situated in London, UK and with configuration: Intel i7-10700K CPU, 64GB RAM, and Windows 11 OS. The simulator also runs within the broker node.

Simulated Testbed: We consider 50 host machines as a scaled up version of the physical setup. Here, each category has 5 times the instance count, *i.e.*, 30 edge nodes and 20 cloud nodes. This is to test models in a larger-scale edge-cloud environment as considered in prior art [Basu et al., 2019, Ahmed and Pierre, 2018]. We use a trace driven simulation model that emulates real execution of tasks, but without actual job execution, allowing us to perform experiments on scale.

We run all experiments for 200 scheduling intervals, with each interval being 300 seconds long, giving a total experiment time of 16 hours 40 minutes. We average over five runs and use diverse workload types to ensure statistical significance in our experiments.

6.3.2 Workloads

6.3.2.1 Tasks

We use popular and real-world workloads to test GOSH and GOSH* models. For our simulator, we use the *Bitbrain*⁷ workload traces dataset [Shen et al., 2015]. These traces are generated by running computational analysis applications on 1750 VMs. The workloads consist of data intervals of 5 minutes, including the CPU, RAM, Disk and Bandwidth utilization of all tasks. We use both *random* and *sequential* computation traces, sampling them uniformly at random when generating workload containers.

For our physical setup, we use *DeFog* benchmark applications [McChesney et al., 2019]. We use the COSCO framework, introduced in Chapter 3, to map these fog benchmark applications to containers run in our physical setup. These include the popular deep learning bag-of-task applications: Yolo, PocketSphinx and Aeneas. At the beginning of each scheduling interval we create *Poisson*(λ) tasks with $\lambda = 1.2$ tasks for physical and $\lambda = 5$ tasks for the simulated setup.

6.3.2.2 Workflows

For our physical setup, we use *WFCommons* benchmark applications [Coleman et al., 2021] which are derived from Pegasus workflows [Bharathi et al., 2008]. These include the popular non-preemptive scientific applications: BLAST, Cycles and Montage, each having 15 to 100 tasks sampled uniformly at random. The BLAST workflow is a high-throughput compute-intensive application for genome based bioinformatics calculations and has applications in smart health monitoring systems at the edge [Rafique et al., 2020]. The Cycles benchmark performs several agro-economic modelling based calculations and is popular in edge computing based smart-agriculture [O’Grady et al., 2019]. Montage is a benchmark derived from astronomical imaging systems and similar to other edge computing based space exploration applications [Semenov, 2021]. The Pegasus workflows are popular benchmark applications that are representative of a large number of edge and cloud computing

⁷The BitBrain dataset can be downloaded from: <http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains>

environments. These have been used in several previous works that test workflow scheduling in edge-cloud environments [Lin et al., 2019, Shao et al., 2019, Xie et al., 2019]. Due to these benchmarks being standard in most prior work, we use them in our experiments.

For our simulator, we use the execution traces of the three applications and the precedence graphs from the benchmark specifications. These traces are generated by running the experiments on the 10 node physical setup. The traces consist of data intervals of 5 minutes, including the CPU, RAM, Disk and Bandwidth utilization of all tasks. We sample monitoring data uniformly from the three applications as done in the *WFCommons*'s workload generator.

At the beginning of each scheduling interval we create $Poisson(\lambda)$ workflows with $\lambda = 1.2$ workflows for physical and $\lambda = 5$ workflows for the simulated setup. These values have been taken directly from traces generated from real-world computing applications deployed on Bitbrains datacenters with 10 and 50 servers [Shen et al., 2015]. The 1.2 and 5 values have been obtained after normalizing the computational requirements of tasks using the total instructions executed for each task, calculated as the product of average MIPS and execution time. This is done to ensure that the experimental testbed is under a similar load as typical real-world physical deployment.

6.3.3 Model Training and Assumptions

6.3.3.1 GOSH Model

For the GOSH and GOSH* algorithms, we use standard NPN models with the following characteristics. We use the following non-affine activation functions for our neural approximators to be differentiable for all input values. Consider the number of hosts as N and the maximum number of tasks as M .

1. Input layer of size $|S_t| + M \times N$ for GOSH and $|S_t| + M \times N + 1$ for GOSH*. The non-linearity used here is `softplus`⁸. Note, N may vary from 10 in tests in the physical environment to 50 in the simulator.
2. FC/NPN layer of size 128 with `softplus` activation.
3. FC/NPN layer of size 64 with `tanhshrink` activation.
4. FC/NPN layer of size 1 with `sigmoid` activation.

The f model has NPN layers with same sizes as above. The g and h models had FC layers with the g model using 50% dropout for Bayesian inference as in prior work [Wang et al., 2016].

We generate training data for GOSH by running a random placement scheduler for 2000 scheduling intervals to capture diverse state-space. For GOSH*, we create a separate dataset by running a random scheduler with pre-trained models of the GOSH scheduler to include $\bar{O}_t$ in the datapoints. To implement and test GOSH and GOSH* models, we use the COSCO framework introduced in Chapter 3. The gradient calculation and optimization (Equation 6.7) are performed using the Autograd package [Paszke et al., 2017]. At training time, we use 50% dropout in the teacher network and learning rate 10^{-5} in the AdamW optimizer. We perform five-fold cross-validation for statistical significance, where we randomly sample 80% of data to get the training set and the rest as the cross-validation set. All models are trained using the PyTorch 1.8 library. To prevent

⁸The definitions of these activation functions can be seen at the PyTorch web-page: <https://pytorch.org/docs/stable/nn.html>.

model overfitting, we use decoupled adaptive weight regularization from [Loshchilov and Hutter, 2018] using the weight decay parameter of 10^{-4} . The values of the learning rate and the regularization hyperparameters are obtained via grid search using the `Ray Tune` API in PyTorch [ray]. We also use the early stopping criterion to train the models [Prechelt, 1998]. The hyperparameter search is performed on the mean five-fold cross-validation accuracy.

6.3.3.2 MCDS Model

To optimize the QoS parameters, we consider a standard objective function that focuses on energy consumption and response time as in Chapter 3. For interval I_t ,

$$\mathcal{O}_t = 1 - \alpha \cdot AEC_t - (1 - \alpha) \cdot ART_t, \quad (6.23)$$

where AEC_t and ART_t are the normalized average energy consumption of the environment in I_t and average response time of workflows leaving at the end of I_t . Here, α is a hyper-parameter that can be set by users as per the application requirements.

For our experiments, we keep $\alpha = 0.5$ in (6.23). We also set $\phi = 10$ (number of actions in simulations for model training) and $\psi = 50$ (number of MCTS iterations). We also use $c_1 = 0.1, c_2 = 0.5$. All hyper-parameter values are obtained using grid-search. In a realistic setup, if performing grid-search on the real system is infeasible, we may perform grid-search on the co-simulator. We get similar trends for other values. We present a sensitivity analysis on ψ later in Section 6.3.6.3.

6.3.4 Baselines

6.3.4.1 GOSH Model

We evaluate the performance of the GOSH/GOSH* model against the state-of-the-art approaches discussed in Chapter 2. These include the pessimistic-optimistic UCB optimization approach *POND* [Liu et al., 2020], deterministic surrogate based methods *GOBI* and *GOBI**, *A3C* [Tuli et al., 2020] and the *HGP* model, where we use the LBFSG method with a HGP surrogate model for optimization as done in [Jamshidi and Casale, 2016, Bui et al., 2017].

We also compare the proposed methods with ablated models, where we replace either one of second-order optimization or heteroscedastic modeling components with simpler versions as described below.

- HGOBI/HGOBI*: GOSH with first-order optimization (as in GOBI) instead of second-order. Thus, LCB_t was optimized using first-order gradient descent. HGOBI* is the co-simulator aided extension of HGOBI.
- SGOBI/SGOBI*: GOSH with updating the output of an FCN neural model (as in GOBI). Thus, the optimization was done as per (6.6). SGOBI* is the co-simulator aided extension of SGOBI.

6.3.4.2 MCDS Model

We compare MCDS against the state-of-the-art baselines: ESVR, DNSGA, IMPSO and Closure as described in Chapter 2. We also add a vanilla GOBI method as a reference point for our results to compare against the state-of-the-art deep surrogate model based scheduler.



Figure 6.6: Comparison of GOSH and GOSH* against baselines and ablated models on physical setup with 10 hosts

Table 6.1: Comparison of GOSH and GOSH* against baselines and ablated models on simulator with 50 hosts.

Model	Energy	Response Time	SLA violation	Scheduling Time	Fairness	Migration Time	Wait Time
POND	326.21	2083.81±176.25	2.93E-01	18.94±0.01	0.56±0.01	1.10±0.07	114.28±0.32
A3C	270.28	2017.52±164.76	3.29E-01	15.77±0.03	0.51±0.02	0.72±0.06	206.34±0.26
HGP	364.20	2219.36±202.75	1.32E-01	21.56±0.00	0.60±0.01	0.21±0.01	60.64±0.12
GOBI	214.29	1767.29±126.00	1.51E-01	6.19±0.00	0.53±0.01	0.84±0.01	26.98±0.07
GOBI*	211.05	1487.06±123.98	7.02E-02	13.73±0.00	0.59±0.01	0.31±0.02	4.13±0.04
HGOBI	209.22	1557.84±137.79	5.22E-02	11.22±0.01	0.53±0.01	0.38±0.02	24.90±0.11
HGOBI*	215.29	1295.32±157.90	3.22E-02	17.33±0.01	0.59±0.01	0.31±0.02	20.81±0.15
SGOBI	182.21	1324.99±84.07	6.40E-02	5.46±0.00	0.48±0.02	0.41±0.03	14.41±0.08
SGOBI*	179.09	1291.31±96.20	4.99E-02	10.86±0.00	0.46±0.01	0.35±0.01	12.73±0.05
GOSH	179.10	1275.80±89.89	3.09E-02	7.98±0.01	0.42±0.01	0.19±0.01	8.31±0.07
GOSH*	176.66	1202.39±77.55	2.37E-02	15.98±0.01	0.51±0.02	0.33±0.02	12.23±0.09

6.3.5 Results for GOSH

6.3.5.1 Baseline Comparison and Ablation Analysis

We now provide comparative results showing the performance of the GOSH and GOSH* schedulers against the baselines and ablated models. Figure 6.6 shows results for the physical setup with 10 Azure hosts and Table 6.1 for the simulation platform with 50 hosts.

Figure 6.6(a) shows the average energy consumption per task for the Azure hosts, with Figure 6.6(i) showing energy consumption with execution time. GOSH and GOSH* (150.9 – 157.6 KW-hr respectively) have lower energy consumption than the GOBI* approach by 17.91% – 21.40% where GOBI* has the least consumption among the baselines. This is because of the ability of the second-order optimization approach to reach better objective scores and hence reduce the AEC_t metric, as the SGOBI/SGOBI* models also reach low energy consumption values. Further, the reduced average response time for GOSH and GOSH* models gives a higher number of completed tasks (Figure 6.6(l)) and hence lower average energy consumption. A similar trend is also seen for the simulation setup (Table 6.1) where GOBI* achieves the least energy consumption among the baselines (211.05 KW-hr) and GOSH/GOSH* improve upon this by 15.13% – 16.29%.

Figure 6.6(b) compares the average response time for each scheduler. Again, the GOBI* scheduler has the least average response time of 154 ± 29 seconds. Here, GOSH* has much lower response times (28.57% lower than GOBI*) and the GOSH scheduler has 28.29% lower than the GOBI model. Again, this is because of the second-order optimization, which gives low scheduling overhead and can reach better objective scores of ART_t . We see here that the ablated model HGOBI* has higher response times than the GOBI* model. Specifically, HGOBI* gives 16.88% higher response time than GOBI*. On the contrary, the HGOBI* model has a significantly lower SLA violation rate compared to the best baseline GOBI* (relative improvement is 44.21%) as seen in Figure 6.6(d). This is because the HGOBI* model aims to minimize the value-at-risk instead of the mean response time, reducing the chance of violating the SLA deadline. Among the baselines, the GOBI* model has the lowest SLA violation rate (5.82%) with the HGP model coming second (8.21%). The GOSH and GOSH* models beat the GOBI* scheduler giving SLA violation rates of just 3.04% – 1.11%. Similar trends are seen in the case of 50 simulated hosts where the GOSH and GOSH* models give 14.25% – 19.16% lower average response time and 55.98% – 66.23% lower SLA violation rate.

For the three applications, the average response time and SLA violation for each scheduler are shown in Figures 6.6(g) and 6.6(h). The high response time of the POND baseline is mostly attributed to frequent

Table 6.2: Loss values of various models on test data. The values are averaged over the five-fold cross-validation runs.

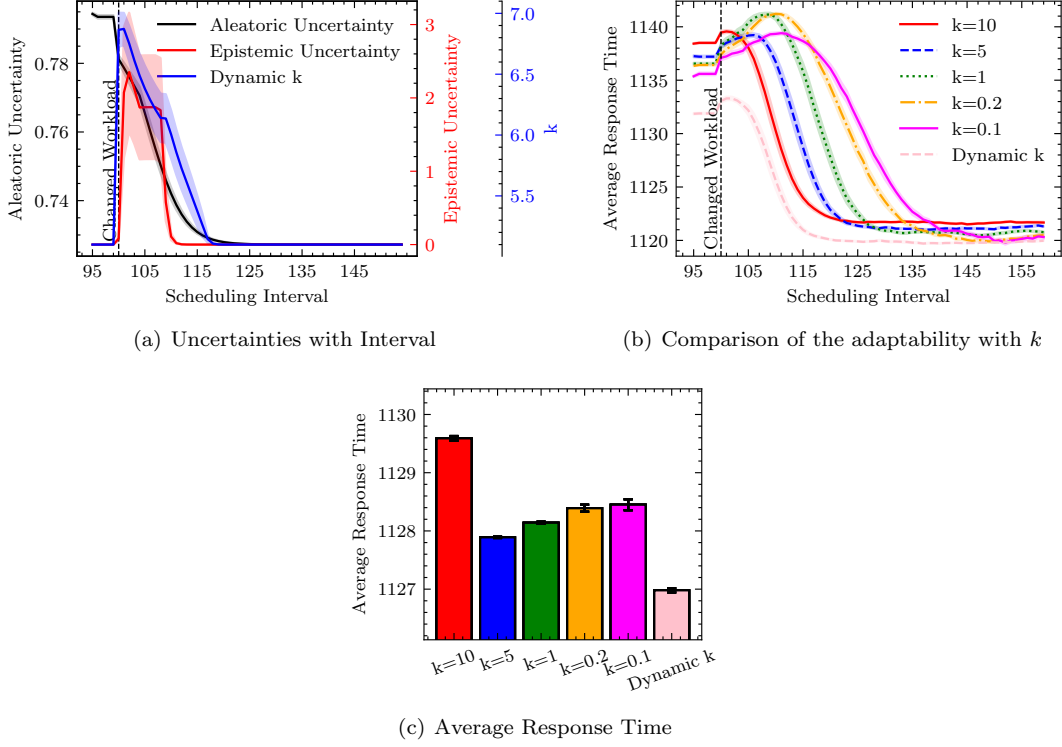
Scheduler	Model	MSE	KLD
GOBI	FCN	0.2582	2.3344
GOBI*	FCN	0.1176	1.9837
HGP	HGP	0.3821	1.0928
GOSH	NPN	0.2812	0.9236
GOSH*	NPN	0.1182	0.3542

migrations performed by this method (see Figure 6.6(e)). On the other hand, GOSH* is the most parsimonious strategy in task migrations, allowing lower response times than other approaches. The deviations from the 5% mark in Figure 6.6(h) is due to the stochasticity in the workloads, motivating us to fine-tune the model at each interval (line 8 in Alg. 6.2). Clearly, among the DeFog benchmark applications, Yolo takes the maximum time. As seen previously, Figure 6.6(h) shows the SLA violations rates for each application. The SLA violation rates of the SGOBI and SGOBI* methods are higher than the heteroscedastic models GOSH, GOSH*, HGOBI and HGOBI*. Even the HGP baseline has a lower SLA violation rate than the SGOBI model for the Yolo application. This shows the importance of having heteroscedastic modeling in estimating QoS scores. These improvements are at the cost of task execution fairness.

In terms of the scheduling time, as seen in Figures 6.6(f) and 6.6(j), the co-simulation based variant of GOBI, HGOBI, SGOBI and GOSH have higher scheduling times. This is due to the additional step that calls the co-simulator to estimate the QoS scores of the next interval (see line 6 in Alg. 6.2). GOBI has the least scheduling time among baselines, showing the importance of gradient-based optimization. However, the second-order optimization further reduces the scheduling time of SGOBI and SGOBI* by 11.79% – 20.90% compared to GOBI and GOBI* respectively in the large-scale setting of 50 hosts (see Table 6.1). However, having a more sophisticated neural network (NPN) in HGOBI and HGOBI* increases the scheduling time compared to GOBI and GOBI* due to a slight increase in the time taken to fine-tune the model (line 31 in Alg. 6.1 and line 8 in Alg. 6.2). A more detailed comparison of the model overheads is given in Section 6.3.5.2. Figure 6.6(c) shows that the GOBI* baseline is the fairest among all models with a Jain’s fairness index of 0.75. GOSH* has the second highest fairness of 0.57, higher than all baselines other than GOBI*. The average wait times for applications with execution is shown in Figure 6.6(k). A3C has high wait times due to the inability of the RL baseline to quickly adapt in cases of hosts running and capacity. Comparing other metrics like fairness, GOSH and GOSH* have lower Jain’s fairness index value than the GOBI and GOBI* models. Among all models, the GOBI* model has the highest fairness index, with GOSH and GOSH* having only 0.26 and 0.57 respectively. The scheduling time is lowest for the SGOBI ablated model of 2.33 seconds in the physical setup and 5.46 seconds in the simulated setup. Moreover, the scheduling overhead of GOSH compared to GOBI is 12.72% and that of GOSH* compared to GOBI* is 15.22%. Similar overheads are observed for the 50 simulated hosts case of 16.39% – 28.9%.

6.3.5.2 Additional Experiments

Modelling Accuracy. Experiments on the dataset collected using a random scheduler show that the NPN model is able to give lower KL divergence between the predicted output and the ground-truth compared to the

Figure 6.7: Sensitivity Analysis for k

baseline FCN model, even though the Mean-Square Error is slightly higher (0.5 – 8.9%, see Table 6.2). The NPN models perform even better than the HGP models used in prior work [Jamshidi and Casale, 2016, Bui et al., 2017], giving more accurate predictions and hence improving QoS.

Sensitivity Analysis. We now show the importance of dynamically updating the value of the k parameter based on the epistemic uncertainty. We conduct our experiments on the simulated setup with the Bitbrain workload traces. We run the GOSH model for 200 intervals with static values of k and one with $k = 5$ initialization being updated at each interval as per (6.9). We keep the workloads as Bitbrain random traces for the first 100 intervals and the sequential traces for the next 100 intervals. The “Changed Workload” vertical line in Figure 6.7 highlights the instant of this workload transition. We average our results over 20 runs for 5 cases of static k with the dynamic k setup.

As is apparent from Figure 6.7(a), the aleatoric uncertainty in the random traces is higher than the sequential traces. The sharp fall in the epistemic uncertainty (Figure 6.7(a)) is due to the fact that the model explores and quickly reaches a local region of “well known” state space. Figure 6.7(b) shows the average response time per interval of leaving tasks. The converged response time (before the 100th and after the 150th interval) is lower for low values of k as a higher relative weight is given to the value-at-risk metric. However, as k increases, the model takes more time to adapt to the new scenario, as shown by the slopes in Figure 6.7(b), indicating that higher k enforces higher exploration and swift adaptability. Finally, Figure 6.7(c) shows that the average response time over the complete 200 interval duration is also minimum for the dynamic k setup. This indicates that dynamically adapting the k parameter is crucial for quick adaptability in volatile environments.

Agility Testing. Figure 6.8 compares how quickly GOSH is able to adapt to the new setting compared to baselines. As we see, GOSH, HGOBI and HGP are able to adapt quickly to new settings compared to GOBI or SGOBI. Both SGOBI and GOBI take twice the time (40 intervals) to converge to the new scenario compared

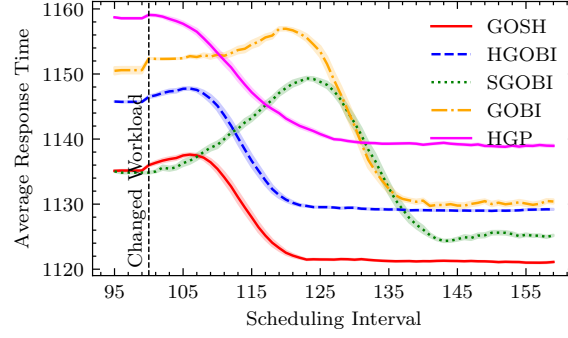


Figure 6.8: Comparison of the adaptability of various models.

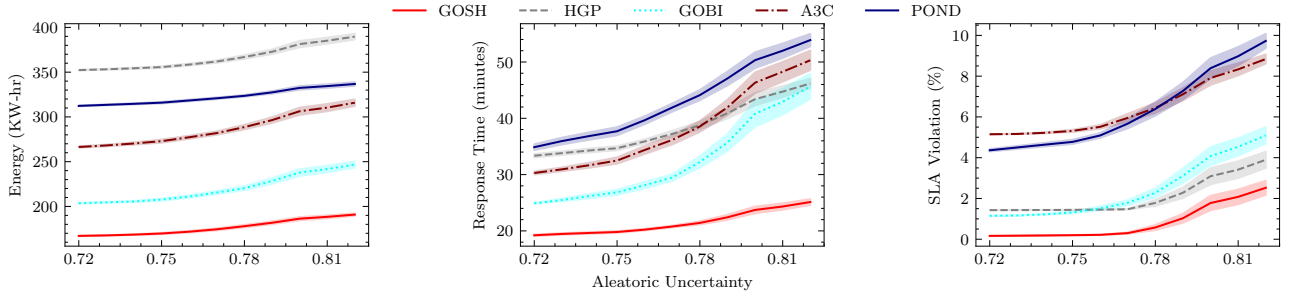


Figure 6.9: Comparison of the performance with increasing uncertainty.

to GOSH. Thus, heteroscedastic modeling aids quick adaptability in the proposed methods.

Performance with Uncertainty: Figure 6.9 compares the performance of the GOSH approach with the baseline models as the aleatoric uncertainty of the system increases. We exclude the co-simulated extensions to focus on the effects of second-order optimization and heteroscedastic modeling. To test this, we vary the ratio of the *sequential* and *random* traces of the Bitbrain dataset (with the former having lower uncertainty than the latter). We run them sequentially with the *sequential* only workloads first. The *sequential* only workload traces give the lower bound (0.72) and *random* only traces give the upper bound (0.81) of the uncertainty considered across the x axis in Figure 6.9. In terms of energy consumption, response time and SLA violation rate, GOSH not only has better metric scores, but also does not face excessive performance drop in highly uncertain cases, thanks to its gradient regularization.

Overhead Comparison: Figure 6.10 compares the cost per iteration (in milliseconds), average number of iterations and the overall scheduling time for GOSH and baseline models. The per-iteration cost of SGOBI is higher due to the additional step of Hutchinson’s method and moving average computation. However, the improvement due to the reduction in the number of iterations is more than this excess cost, so much so that the overall scheduling time of SGOBI is lower than GOBI. This allows the overall overhead of GOSH due to the heteroscedastic modeling to not be as high as GOBI (33.65%) compared to the HGOBI approach (40.12%).

6.3.6 Results for MCDS

We show the results of our experiments on the physical and simulated testbeds in Figures 6.11 and 6.12 respectively. Figures 6.11(a) and 6.12(a) show that the MCDS model has the lowest energy consumption compared to the baseline methods. MCDS scheduler gives an average interval energy consumption of 228.21 KW-hr for 10 hosts and 1326.98 KW-hr for 50 hosts. As the number of hosts increases so does the power consumption of the

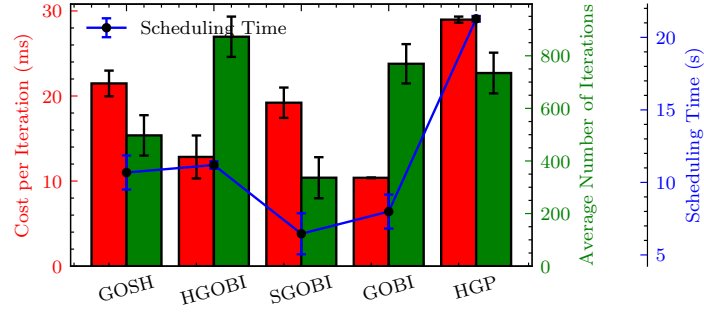


Figure 6.10: Comparison of the scheduling overhead of various models.

setup. However, MCDS has up to 6.13% lower energy consumption than the second lowest value of the Closure scheduler. The major reason for this is the low response time of each application, which leads to lower number of active tasks in the system at any point giving low CPU utilization and consequently low energy consumption. Figures 6.11(b) and 6.12(b) show the execution times of the workflows, *i.e.*, the time these workflows spend in computation. All schedulers have similar execution times for both setups. Figures 6.11(c) and 6.12(c) show the waiting times of workflows. Here, IMPSO has the highest waiting times as using this scheduler leads to the maximum number of resource contentions in the system. The scheduling times of all schedulers are less than 10 seconds (3.33% of the scheduling interval duration of 5 minutes), with DNSGA and GOBI having the lowest scheduling times (Figures 6.11(d) and 6.12(d)). MCDS has 13.19% lower scheduling time than the best performing baseline, *i.e.*, Closure. A higher scheduling time compared to the meta-heuristic approach DNSGA is due to the multiple Monte-Carlo iterations. However, the model is faster than other approaches like Closure and IMPSO, showing that MCDS is scalable to larger testbeds. More details on scalability with workload size in Section 6.3.6.2.

Figures 6.11(e) and 6.11(e) show the average response times for both setups. Among the baselines, Closure has the lowest average response time of 27.21 intervals in the physical setup and 12.67 intervals in the simulated case (lower response time in the simulated setup due to the larger parallelization capacity). MCDS has a response time up to 4.56% lower than Closure. This is due to the long-term optimization in MCDS, unlike other methods. Myopic optimization in prior work typically prevents them from considering future system states that could cause an increase in response time due to resource contention or migrations. This allows MCDS to be parsimonious in terms of number of preemptive migrations (Figs. 6.11(l) and 6.12(l)). Comparing the number of migrations across baselines, we see that IMPSO and ESVR aggressively migrate tasks, whereas the other methods try to minimize migration overheads to optimize QoS. Low response times of the MCDS scheduler also results in low SLA violation rates as seen in Figures 6.11(g) and 6.12(g). MCDS has an SLA violation percentage of 0.25%-1.12%, lower than the best values among the baselines, which are 1.12%-2.04% for the Closure scheduler. IMSO has a high SLA violation rates due to the high waiting and migration times as seen in Figures 6.11(k) and 6.12(k). This is due to the aggressive migration in IMPSO as seen in Figures 6.11(l) and 6.12(l). ESVR has relatively low SLA violation rates for its response times due to the deadline-first prioritization in the scheduling decisions. Figures 6.6 and 6.12 also show the response times and SLA violation rates for each application type (BLAST, Cycles and Montage). The SLA violation rates for each application is shown in Figures 6.11(h) and 6.12(h). The deviations from the 2% mark in these figures is due to the stochasticity in the workloads, motivating us to fine-tune the model at each interval (line 20 in Alg. 6.4).

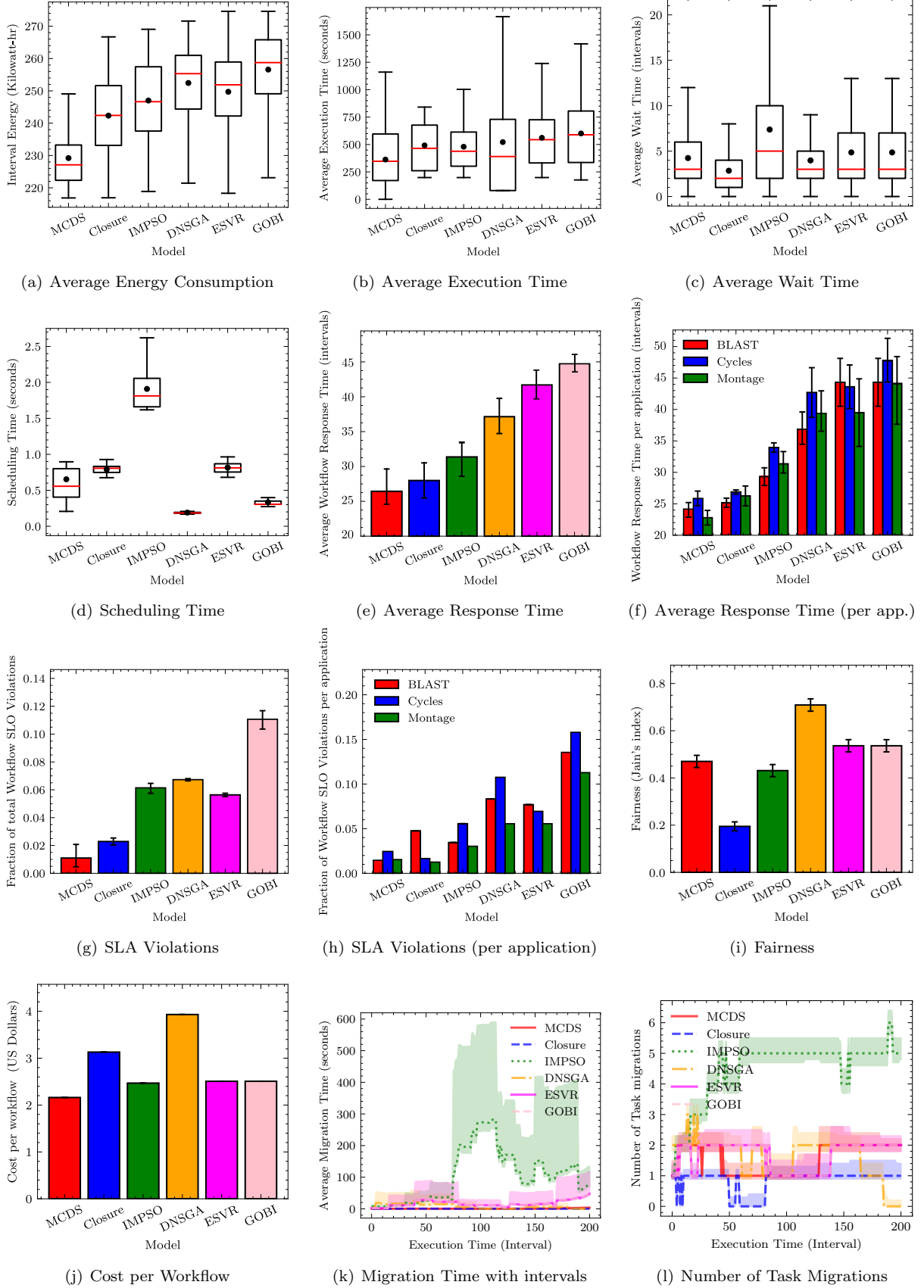


Figure 6.11: Comparison of MCDS against baselines on physical setup with 10 hosts.

Comparing the fairness indices of the schedulers (Figures 6.11(i) and 6.12(i)), we see that the DNSGA has the maximum fairness index of 0.34-0.72. MCDS has a fairness index of 0.28-0.52. This is due to the long-term

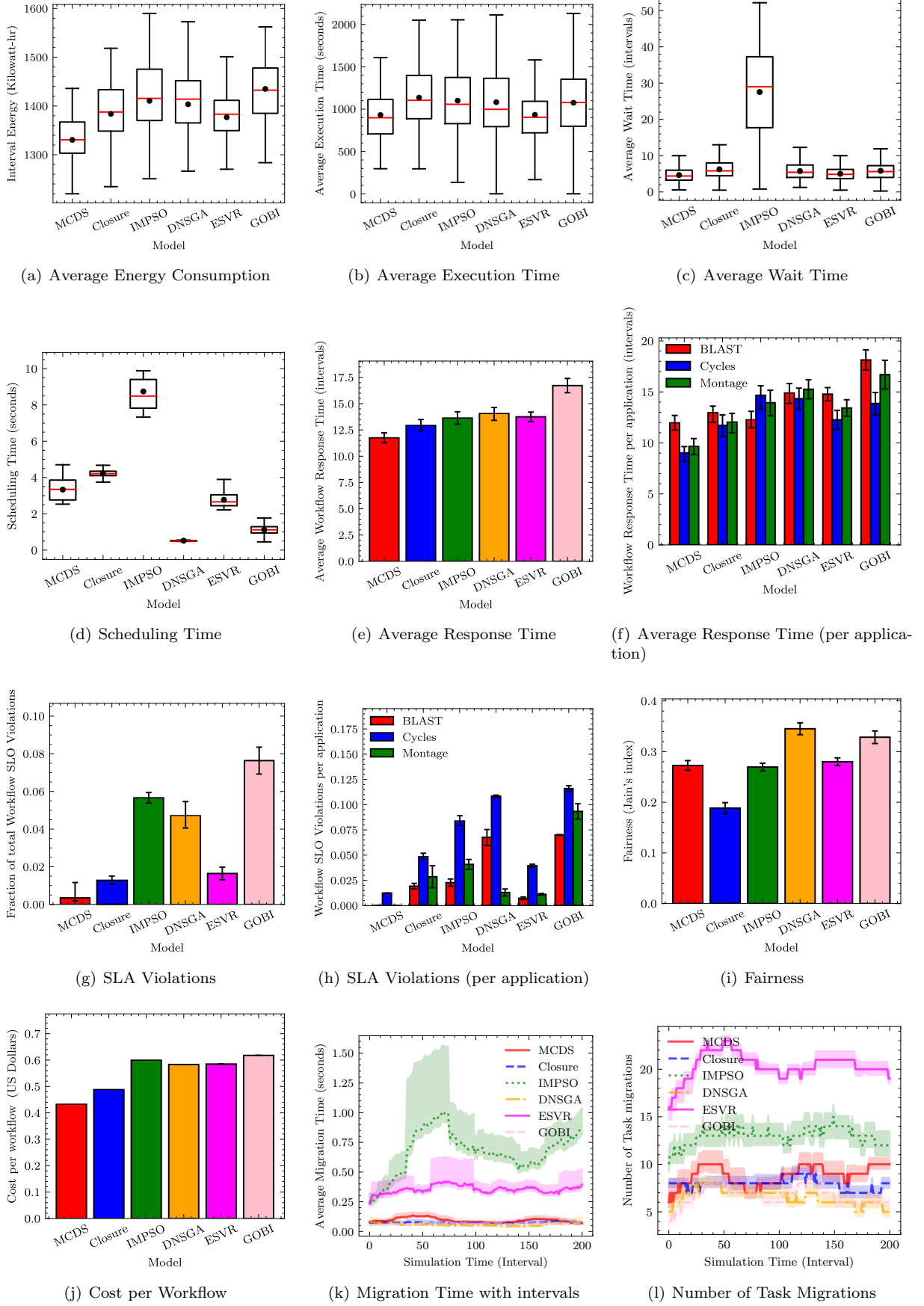


Figure 6.12: Comparison of MCDS against baselines on simulated setup with 50 hosts.

Table 6.3: Ablation Analysis. Units: Energy - *KW-Hr*, Response Time - scheduling intervals, SLA Violations - percentage, Scheduling Time (cumulative) - seconds.

Model	Energy	Resp. Time	SLA Violation	Sched. Time
MCDS	1326	12.08	0.468	3.3627
w/o MCTS	1479	14.21	4.283	0.3123
w/o DSM	1401	13.20	3.027	7.4231
w/o Exploration	1392	12.98	0.832	2.6283
w/o Domain Kn.	1532	19.41	0.930	2.0318

optimization in MCDS, preventing resource contention and allocation of resources to tasks corresponding to their requirements. As MCDS is able to complete the maximum number of workloads in the duration of 200 intervals, the amortized cost per workflow of MCDS is lowest among all models (0.43 and 2.21 USD/workflow). MCDS gives 16.12%-30.71% lower cost than the baselines with ten hosts (Figure 6.11(j)). Similar trends are seen in the simulated setup with 50 hosts (7.62%-70.14%).

6.3.6.1 Ablation Analysis

To study the relative importance of each component of the model, we exclude every major one and observe how it affects the performance of the scheduler. An overview of this ablation analysis is given in Table 6.3. First, we consider the MCDS scheduler without the Monte-Carlo tree search, *i.e.*, only using the pre-trained deep surrogate model to get the GOBIGraph output (*w/o MCTS* model). Second, we consider a model without the deep surrogate model, *i.e.*, we only use MCTS simulations with a random scheduler to get the QoS estimates and generate the scheduling decision at each interval (*w/o DSM* model). The other two models we consider are modifications in the selection formula. In the *w/o Exploration* model we ignore the UCB exploration metric. In the *w/o Domain Kn.* we ignore the GOBIGraph based indicator term in (6.22). We reach the following findings:

- Without the MCTS search scheme, the model performs a myopic optimization and suffers the maximum increase in the SLA violation rate. This demonstrates the need for a long-term objective estimation and bootstrapping is crucial to reach optimal QoS.
- Without the deep-surrogate model, the scheduling time significantly increases. This is due to the large number of co-simulation steps being performed in the MCTS iterations. It also affects the QoS values due to the inaccurate estimates obtained from the simulations based on the random scheduler.
- Without the exploration term, the model is unable to find possibly better scheduling decisions. Also, the domain knowledge term, *i.e.*, the GOBIGraph based indicator term in (6.22) helps the model to use the deep surrogate model for exploration. Both these terms help improve the scheduling decisions and reach better QoS values.

We observe that Monte-Carlo learning and deep surrogate model have the maximum impact on the performance. UCB exploration and domain knowledge are additional features that help improve QoS.

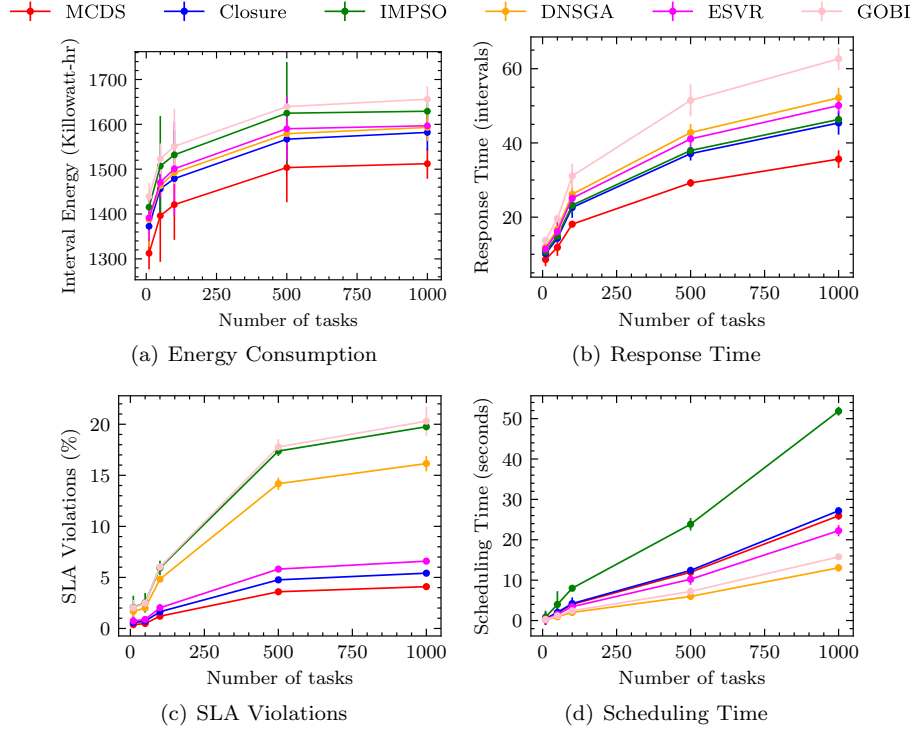


Figure 6.13: Scalability Analysis for MCDS and baseline methods with the number of tasks in each workflow as 10, 50, 100, 500 and 1000.

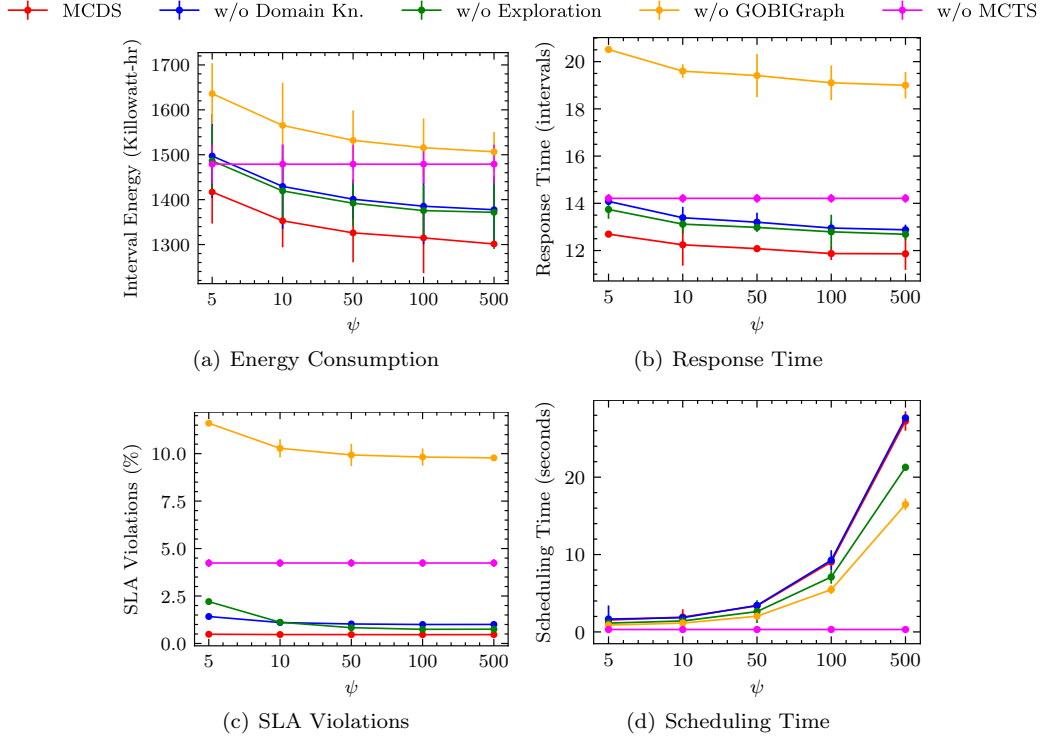


Figure 6.14: Sensitivity Analysis for ablated models with number of Monte Carlo iterations (the ψ parameter in our formulation).

6.3.6.2 Scalability Analysis

The *WfGen* utility within *WfCommons* benchmark suite allows us to change the number of tasks in each workflow DAG.⁹ In our main experiments, we sample the number of jobs uniformly at random between 15 and 100. We now use this feature to test the scalability of the MCDS model and compare it against the baselines on the simulated setup with 50 hosts. Figure 6.13 shows the energy consumption, response time, SLA violations and scheduling time when we vary the number of tasks in each workflow from 10 to 1000. Comparing across models, the DNSGA, ESVR and IMPSO do not scale well in terms of SLA violation rates as the number of tasks grows. Only models that use deep surrogate networks, *i.e.*, MCDS and GOBI, and the attack-defense based scalable baseline Closure are able to scale with increasing task count. Similarly, other parameters, *i.e.*, energy consumption, response time and scheduling time increase with the number of tasks. In terms of energy consumption, response time, SLA violation rate and scheduling time, MCDS not only has better metric scores, but also does not face from excessive performance drop in cases where workflows have a large number of tasks, thanks to its gradient graph convolution based adaptive normalization. The deep surrogate model in MCDS is able to handle large-scale workflows also due to the *Bahdanau* attention that brings down the dimensional size of the input size windows.

6.3.6.3 Sensitivity Analysis

We check the sensitivity of the performance parameters to the ψ parameter (number of Monte-Carlo iterations), as shown in Figure 6.14. The figure shows the energy consumption, response time, SLA violations and scheduling time when ψ varies from 5 to 500. As the number of iterations increases, the model can cover larger state-action space and get a much better look-ahead estimate. Thus, the energy consumption, response time and SLA violations decrease as ψ increases (except the *w/o MCTS* model which is unaffected as MCTS is not performed here). Although, after $\psi = 50$ the improvement in the QoS performance is minimal. However, increasing the simulation steps has an adverse impact on the scheduling time of the model. We choose $\psi = 50$ in our experiments to balance the trade-off between QoS and scheduling overhead.

6.4 Summary

In this work, we have presented multiple contributions including heteroscedastic modeling for aleatoric uncertainty estimation and value-at-risk calculation, a teacher-student model for epistemic uncertainty estimation, efficient second-order optimization and feedback based exploration control. All these contributions allow GOSH and GOSH* to out-perform the baseline models in terms of QoS metrics such as energy, response time and SLA violation by up to 18%, 27% and 82% respectively in a heterogeneous fog environment with real-world workloads. As shown in Figure 6.6(f), between GOSH and GOSH*, the former is more suitable for environments with a resource-constrained fog broker, whereas the latter reaches a better objective score but requires heavier computational resources.

We have also presented MCDS, a novel scheduler for efficiently scheduling demanding workflow applications on mobile edge-cloud computing environments. Unlike prior work that performs myopic optimization of QoS using reinforcement learning, evolutionary methods or heuristics, MCDS trains a long-term QoS estimation

⁹WfGen - WfCommons <https://wfcommons.org/generator>.

surrogate model. To do this, MCDS uses a modified version of the Monte-Carlo-Tree-Search that uses UCB exploration and gradient-optimization based biased exploitation. This allows MCDS to manage workflows to maintain high inference objective scores on an average and reduce the scheduling time as it does not need to perform several multi-step simulations at test time. All these contributions allow MCDS to be scalable and outperform the state-of-the-art models in terms of energy consumption, response time, SLA violation rate and cost by up to 6.13%, 4.56%, 45.09% and 30.71% respectively in a heterogeneous mobile edge-cloud environment with real-world workloads. Thus, extending the ideas of co-simulation and gradient-based optimization (in Chapter 3) to higher-order and long-term optimization allow us to achieve objective 2 as mentioned in Chapter 1.

This page intentionally left blank.

Chapter 7

AI Augmented Fault-Tolerance

This chapter describes novel fault-tolerance approaches, namely PreGAN and DeepFT and their extensions, to achieve fault-tolerance (objective 3(a)) as described in Chapter 1. In Chapters 4, 5, and 6 we discussed how to optimally provision resources, deploy applications and place tasks in a fog environment. However, to ensure robust task or workflow execution, it is crucial to detect and diagnose faults and run appropriate remediation steps to maintain fault-tolerant services. This is particularly of importance in edge computing environments where the nodes may be resource constrained and prone to contention related faults. To address the adverse effects of resource contentions on system QoS, we extend ideas of GOBI (introduced in Chapter 3) with generative and classification models for optimal fault-tolerance in mobile edge environments.

The paradigm of mobile edge computing follows the data gravity principle, wherein the processing occurs close to the source of data, which typically are the sensors and actuators in an Internet of Things (IoT) framework [Gill et al., 2019]. This entails having compute devices geographically spread to cater to the large number of users, say within a smart-city or a smart-grid like environment with a large number of interconnected nodes [Tuli et al., 2022]. In such contemporary use cases, workloads typically demand low latency execution and usually have non-stationary resource requirement characteristics exacerbating the challenge further [Gill et al., 2019]. When performing computation at such scales it becomes financially infeasible to have computationally heavy nodes as edge devices. In such settings, practitioners and service providers prefer instead to deploy resource limited nodes at the edge. Within such constraints it is common to have resource contentions that lead to system downtimes, which could lead to significant financial losses for service providers [Franke, 2011, Nicoletti, 2013]. The increasing amounts of data requiring immediate processing and the high level of volatility in modern mobile systems is pushing edge resources to their limits [Liu et al., 2016, Dastjerdi et al., 2016]. This gives rise to the need for developing fault-tolerance mechanisms that not only are able to quickly adapt to changes in edge systems, but also accurately predict contentions for optimal remediation.

Challenges. The problem of fault-tolerant computing is challenging. A part of fault-tolerance is the ability to proactively predict faults and diagnose root-cause issues to run apt remediation steps. For mobile edge computing, this entails the prediction of the specific nodes where resource contention might occur and also identifying the specific resource type (such as CPU, memory or disk) that would get overloaded. Fault predictions can then be utilized to inform resource management and decision-making to proactively avoid overloads or faults. These steps should be carried out in near real-time for such systems to be effective [Park et al., 2018]. Further, to

ensure efficacy, such systems need to be able to predict and resolve diverse types of faults [Malik and Huet, 2011]. However, this is challenging due to the volatile nature of workloads and resources, which have characteristics that may change unexpectedly [Ristov et al., 2020]. Additionally, in mobile computing environments, the network bandwidth is limited. Performing remediation in the form of task migration may cause long transfer times, consequently increasing the average response times. Thus, we need approaches with low overheads to effectively trade off between remediating faults and having high overheads. Specifically, this requires fault predictions to have low false positives to avoid running unnecessary remediation. Avoiding significant adverse effects of contentions in the form of system downtimes additionally requires such methods to have low false negatives, requiring such prediction models to be extremely accurate. To do this, some existing methods leverage machine learning models due to their high accuracy [Sharif et al., 2020, Negi et al., 2021]. However, in such cases, the annotated log traces available to brokers are limited, leading to restricted size of the labelled fault classification data. This makes it hard to train supervised machine learning models for fault detection and classification. In many cases labels are absent, which requires us to also develop unsupervised fault-tolerance models.

Existing Solutions. Several approaches have been proposed in the past few years for fault-tolerant computing in edge or cloud platforms [Liu et al., 2016, Sivagami and Easwarakumar, 2019, Kumari and Kaur, 2018]. One of the common approaches is to provide node duplication or redundancy and network contingencies to avoid service downtime of any element of a mobile edge framework. However, in large-scale resource-constrained deployments, having a redundant node for each edge host is infeasible [Zhou et al., 2010]. Another approach is to replicate the running instances of the tasks on separate nodes [Zheng et al., 2011]; however, this doubles the compute requirements in an already resource limited setting [Hong and Varghese, 2019]. A solution that is commonly used by prior work is instead *preemptive migration* [Engelmann et al., 2009] where the execution state of a running task is saved as a *checkpoint* which is then transferred to another node to resume execution. This is enabled by virtualization services such as containerization. However, in volatile environments with several tasks running in the same edge system, frequently checkpointing and transferring tasks could impose excessive stress on both system and network. We need parsimonious migration strategies that migrate only the critical tasks. In recent years, many different methods have been proposed to decide the appropriate migration decisions to enhance service reliability. These range from Particle Swarm Optimization (PSO) [Liu et al., 2016], Integer Linear Programming [Sivagami and Easwarakumar, 2019, Tian et al., 2018] and Bayesian Classification [Sharif et al., 2020] to using Neural Network and clustering algorithms like k-means [Negi et al., 2021]. However, such methods are often not generalizable or struggle to adapt in volatile settings quickly and accurately, as discussed later in Chapter 2. Further, recent methods offer unsupervised schemes like clustering [Liu et al., 2016], sparse neural networks [Hu et al., 2021], autoencoders [He et al., 2020] or other meta-heuristic formulations [Liu et al., 2016, Sivagami and Easwarakumar, 2019]. Such methods extract patterns from historical log data without any fault labels, mostly using data reconstruction techniques. However, such methods struggle to adapt in volatile environments as they need several datapoints corresponding to operational and performance characteristics of the system [He et al., 2020]. Moreover, these methods wait for the latest data to detect faults, diminishing their appeal. Thus, we propose novel fault-tolerance models and demonstrate their efficacy against these state-of-the-art baselines [Negi et al., 2021, Sharif et al., 2020].

Developing a Supervised Fault-Tolerance Model. For effective fault-tolerance, it is critical to accurately detect, diagnose and classify faults online for a mobile edge environment. To do this, we utilize in this work,

a neural network based state reconstruction model. A prototypical network is utilized to classify faults into the specific resource type that could be overloaded, such as CPU, memory or disk [Snell et al., 2017]. To efficiently model the temporal trends and the cross-correlations of performance and resource utilization metrics among different edge nodes, we employ graph attention network (GAT) and gated recurrent units (GRU) [Veličković et al., 2018, Chung et al., 2014]. We then use a Generative Adversarial Network (GAN) to generate informed preemptive migration decisions that are utilized to update the task placement decision from an underpinning scheduler [Li et al., 2019]. To train the GAN model, we use a coupled-simulator that, akin to a digital-twin, provides us a simulated platform to test whether the original or the updated decision gives higher Quality of Service (QoS) [Ismayilov and Topcuoglu, 2020]. We call this model that remains static over task execution PreGAN. However, to adapt the model in dynamic settings, we extend PreGAN to PreGAN+ that leverages semi-supervised learning to periodically tune the classification model and leverages a Transformer [Vaswani et al., 2017] based model for time-efficient tuning. PreGAN+ can be configured to run identically as PreGAN, *i.e.*, without online fine-tuning. This chapter leverages co-simulation and self-supervised learning to predict preemptive migration decisions for fault-tolerant edge computing. We perform extensive empirical experiments on real-life edge testbed to compare and analyze PreGAN and PreGAN+ against the state-of-the-art methods. Our experiments show that both methods outperform baselines in terms of QoS metrics, reducing the energy consumption, response time and SLO violations by up to 8%, 7% and 15%, respectively for PreGAN+. It achieves this by up to 12.5% more accurate fault detection, 13.9% lower task migrations and 23.8% lower overhead than the most accurate baseline. PreGAN too outperforms baselines, but the improvements are lower compared to PreGAN+. Although PreGAN+ gives better QoS scores compared to PreGAN, it imposes higher overheads and is more suitable only in settings with high capacity broker nodes.

Developing an Unsupervised Fault-Tolerance Model. A drawback of not using any human domain knowledge is that unsupervised learning approaches can vastly deviate from the ground truth [Pecht and Kang, 2019, § 6.5]. A class of unsupervised learning, called self-supervised learning, is a refactored supervised scheme where the labels are generated by the model itself [Goodfellow et al., 2016]. However, without a trained model, generating such labels is hard. Thus, we propose to obviate the lack of supervised data by means of online simulation, enabled and driven by system data (as per a digital modeling twin [Santos et al., 2020]). A co-simulated digital twin, referred to as a co-simulator in the rest of the discussion, is a software that models the behavior of a physical system, which in our case is an edge computing platform. Recent developments, in co-simulator design, enable a model to quickly run discrete-event simulations to get an estimate of a future state of the system, allowing proactive fault remediation. However, running a co-simulator for every preemptive migration decision is computationally expensive. A key ingredient in our work is the use of a surrogate model that mimics the behavior of these expensive simulations [Kochenderfer and Wheeler, 2019]. Unlike a co-simulator, our surrogate model captures the fundamental aspects of the system dynamics observed from the data without incurring a state space explosion. We propose to use a co-simulator to generate self-supervision labels to fine-tune a reconstruction-based model on-the-fly for fault detection. Using self-generated data permits model training with limited historical data and adaptation to dynamic settings. To obviate volatility, we cast the problem as a few-shot learning classification [Wang et al., 2020], whereby a small number of samples are needed to train a neural network. This allows for faster learning under highly volatile workloads. In this work, we propose DeepFT, a scalable Deep surrogate model based Fault-Tolerance method. DeepFT utilizes a neural network as

a surrogate model to predict a future system state, the possible presence of faults affecting it and their class. It uses co-simulation to generate fault labels and trains the surrogate in a few-shot setting. With a trained surrogate model, DeepFT generates preemptive migration decisions for fault tolerance. Extensive experiments on a Raspberry-Pi based edge cluster show that DeepFT outperforms five recently proposed different baselines in terms of fault-detection and QoS metrics. Specifically, DeepFT reduces response times and latency-based service level objective (SLO) violations by 9.41% and 37.21%, respectively, thanks to its higher fault prediction scores and lower overheads.

7.1 System Model and Problem Formulation

We now provide details of the system model assumptions and formulate the fault-tolerance problem.

7.1.1 System Model

Akin to a system considered in Chapter 3, we consider a distributed edge environment with heterogeneous nodes in a broker-worker fashion.

- The *Compute Layer* consists of a fixed set of m edge hosts (worker nodes), denoted as $H = \{h_1, \dots, h_m\}$.
- The *Management Layer* receives all incoming tasks and includes a broker node with sufficient compute capacity to perform fault-tolerant tasks scheduling.

We consider that an execution in this environment runs for a bounded time and that the timeline is divided into fixed-sized scheduling intervals, where I_t denotes the t -th interval (t ranging from 0 to T). The edge broker can sample the resource utilization metrics of all hosts at any time (e.g., CPU, RAM, disk/network bandwidth, some additional fault-related metrics including consumption of the swap space, disk buffers, network buffers, disk and network I/O waits [Tanenbaum and Woodhull, 1997]). We assume a scheduler is already present in the broker. The scheduling decision and host characteristics at this interval are denoted as S_t and $x_t \in \mathbb{R}^{m \times n}$ (each host has n features). The time-series $\{x_0, \dots, x_{t-1}\}$ is denoted as $\mathcal{T}_t$.

Fault Model. We consider resource contention type faults, which frequently occur in edge workers, while executing data-intensive applications [Vasilakos et al., 2020]. Here, a worker node may become unresponsive due to resource over-utilization [Vasilakos et al., 2020], we thus consider faults of the form of CPU, RAM or Disk contention that we raised when their utilization is greater than dynamically set thresholds. As in prior work, edge hosts are connected to the same power supply and unrecoverable faults like outages are ignored [Sivagami and Easwarakumar, 2019, Javed et al., 2018, Bagchi et al., 2019]. We aim to prevent over-utilization faults by adjusting the task scheduling and migration decisions.

Workload Model. We assume a bag-of-tasks workload model, where a set of independent tasks enter the system at the start of each scheduling interval. These are generated from the users and are transferred to the edge broker via gateway devices or IoT sensors. To evaluate the proposed method in a controlled environment, we abstract out the users and IoT layers in our experiments and apply a discrete probability distribution to realize tasks as container instances (see Section 7.4). Each task has an associated SLO deadline.

7.1.2 Formulating a Supervised Problem

We divide the problem of fault-tolerance into two sub-problems, defined as:

Fault Prediction: For an input multi-variate time-series $\{x_0, \dots, x_{t-1}\}$, we need to predict fault labels for each host in I_t as $y_t \in \{0, 1, \dots, c\}^m$. Here, $y_{t,i}$ denotes the output for h_i ; $y_{t,i} = 0$ means no fault and $y_{t,i} = j$ means fault of class j among the user-specified c classes $j \in \{1, \dots, c\}$.

Preemptive Migration Decision: For an input scheduling decision S_t (set of task and host pairs) and a solution of the prediction problem y_t , we wish to predict a migration decision Δ_t . To model the dependence of a data point x_t at a timestamp t , we consider the sliding window of length k , denoted as

$$W_t = \{x_{t-k}, \dots, x_{t-1}\},$$

as is done in prior work [Audibert et al., 2020]. Thus, in lieu of using just x_t , we use W_t as it allows us to capture the local context. For model robustness, we normalize the input windows. For the sake of simplicity and without loss of generality, we drop the t index whenever this is not ambiguous and use x , S , W , Δ .

7.1.3 Formulating an Unsupervised Problem

Formally, at the start of the interval I_t , the edge broker needs to make a decision denoted as S_t , that is a mapping of new tasks in the interval I_t as well as the active tasks from interval I_{t-1} to hosts $H = \{h_1, \dots, h_m\}$. For new tasks, this becomes a scheduling decision and for existing active tasks, it acts as a preemptive migration decision (if the target host is different from the current host). Now, at the beginning of interval I_t , for an input time-series of system states $\{x_0, \dots, x_t\}$, the broker needs to predict the next system state, *i.e.*, x_{t+1} . Here, the system state (x_t) consists of values for an arbitrary set of resource usage metrics for all active tasks and hosts at the start of I_t . Instead of directly using the system states, we consider a sliding window of length k to capture the temporal contextual trends:

$$W_t = \{x_{t-k+1}, \dots, x_t\}.$$

We use replication padding [Liu et al., 2018] for the first k intervals to ensure the time-series window is always of the same size. Also, to ensure robust predictions in the DeepFT model, we normalize these time-series windows using min-max scaling to be in the range $[0, 1]$.

Using the input window W_t and an input scheduling decision S_t , the model needs to predict whether there is likely to be a fault in the next system state, *i.e.*, x_{t+1} . In lieu of directly predicting whether the next system state is faulty, we predict a fault score f_t using a predicted reconstruction of the next window W_{t+1} , denoted as $\hat{W}_{t+1}$. The fault score f_t is obtained by calculating the deviation between the true window W_{t+1} and its reconstruction $\hat{W}_{t+1}$. We also predict the type of fault (CPU/RAM/Disk contention), referred to as the fault class, of the next state in the form of a prototype embedding P_{t+1} . A prototype embedding [Snell et al., 2017] is a dense vector representation of the fault class a particular input belongs to. Without loss in generality, whenever unambiguous, we drop the subscripts for simplicity. Hence, we only refer to the inputs and outputs as W , S , $\hat{W}$, P and f .

attention as

$$\begin{aligned}\text{MultiHeadAtt}(Q, K, V) &= \text{Concat}(H_1, \dots, H_h) \\ H_i &= \text{Attention}(Q_i, K_i, V_i).\end{aligned}\tag{7.3}$$

Here $\text{Attention}(Q_i, K_i, V_i)$ is the scaled-dot product attention operation [Vaswani et al., 2017]. We apply this on the GRU and GAT outputs (with task SLO requirements and host characteristics) to obtain

$$O = \text{MultiHeadAtt}(S, S, [O_1; O_2]).\tag{7.4}$$

The late-fusion of the two outputs allows the downstream models to exploit them independently. The output O is a collection of attention based encodings for each host. We pass this encoding through two decoders for each host h_i ,

$$\begin{aligned}D_i^A &= \text{softmax}(\text{FeedForward}(O_i)), \\ P_i &= \text{sigmoid}(\text{FeedForward}(O_i)),\end{aligned}\tag{7.5}$$

where $D_i^A \in \mathbb{R}^2$ denotes the fault prediction. The model predicts a fault in h_i if $D_i^A[1] \geq D_i^A[0]$. $P_i \in \mathbb{R}^E$ is a prototype embedding for each host corresponding to the fault class, irrespective of whether a fault is detected or not. This factored prediction for each host allows our model to be agnostic to the number of hosts in the system. We define the embedding for host h_i as

$$E_i^F = \begin{cases} P_i, & \text{if } D_i^A[1] \geq D_i^A[0] \\ [0]_{1 \times E}, & \text{otherwise.} \end{cases}\tag{7.6}$$

We stack all host embeddings to generate E^F . This auto-regressive style of using fault predictions allows us to generate a representation of the fault prediction that only consists of class prototypes for the hosts where a fault is detected.

Preemptive Migration GAN. The embedding output from the FPE is passed through the Generator

$$\Delta = \tanh(\text{LayerNorm}(S + \text{MultiHeadAtt}(S, S, E^F))),\tag{7.7}$$

where the $\Delta \in \mathbb{R}^{p \times n}$ denotes the preemptive migration decision. This is added to the original scheduling decision to get $N = S + \Delta$. The final decision for each task p is calculated as $\text{argmax}(N_p)$. The output of the generator is passed to a discriminator decoder

$$D = \text{softmax}(\text{FeedForward}(N)),\tag{7.8}$$

where $D \in \mathbb{R}^2$ denotes predicted likelihood scores for using S and N as schedules. The final scheduling decision of PreGAN is defined as

$$\text{Final Schedule} = \begin{cases} N, & \text{if } D[1] \geq D[0] \\ S, & \text{otherwise.} \end{cases}\tag{7.9}$$

Summarizing, the FPE detects faults in hosts and generates class prototype embeddings for each host. These

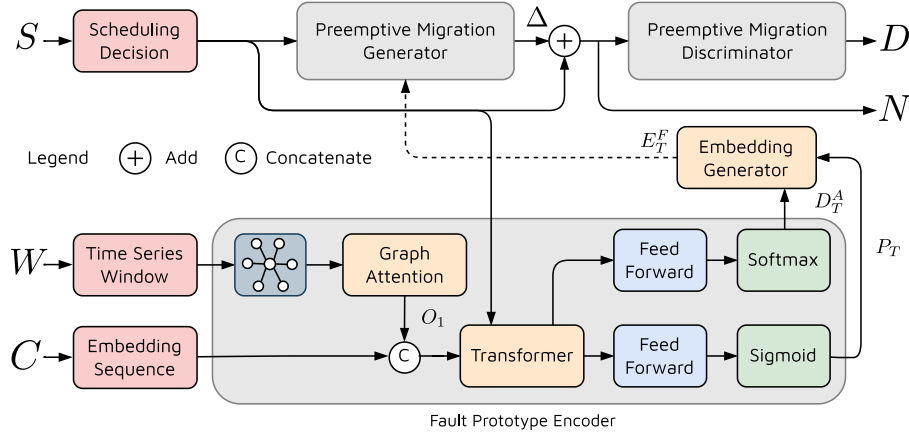


Figure 7.2: The PreGAN+ Model. Generator and Discriminator models are same as in Figure 7.1.

embeddings are stacked (E^F) and sent to a generator network that outputs a preemptive migration vector (Δ). The discriminator then provides a likelihood score to the new (N) and original (S) scheduling decisions. The final output is then the decision with the highest likelihood score.

7.2.1 PreGAN+ Model

The PreGAN model assumes the presence of fault labels and classes to be available to tune its FPE. However, obtaining supervision labels may be expensive in many settings as we describe later in Section 7.4.3.1. Further, recurrent models like GRUs are expensive to fine-tune as they require data to be given in a recurrent fashion. This makes it difficult to frequently tune PreGAN to dynamically adapt its decision making policy. However, once trained, PreGAN is a lightweight model that when kept fixed imposes very low decision making overheads. We perform a more exhaustive analysis of the load on the broker node. To allow using partial or no supervision and facilitate frequent model tuning, we extend PreGAN's FPE in PreGAN+. A diagrammatic representation of the PreGAN+ model is shown in Figure 7.2.

Akin to PreGAN, in PreGAN+ we perform the graph attention operation on the input time-series window W to obtain O_1 . However, instead of the just providing the current graph embedding, we also provide a sequence of all past embeddings of O_1 up to the current timestep t , which we denote as C , to re-use GAT inferences from past intervals. Unlike PreGAN, this enables us to provide the complete historic context, and not just the host characteristics for the last k intervals as in W , for more informed decision making. We concatenate the historic embedding sequence and GAT output $C_1 = [C, O_1]$, and pass it through a Transformer Encoder [Vaswani et al., 2017]. Unlike a recurrent neural network such as GRU or LSTM, a Transformer can take the complete sequence as input. It then uses attention operations to infer patterns across timesteps [Tuli et al., 2022]. Specifically, we position encode the complete sequence C_1 using the trigonometric sin function to obtain C_2 [Vaswani et al., 2017]. The encoder then infers over C_2 conditioned by the scheduling decision S defined as

$$\begin{aligned} C_3 &= \text{LayerNorm}(C_2 + \text{MultiHeadAtt}(C_2, C_2, S)), \\ C_4 &= \text{LayerNorm}(C_3 + \text{FeedForward}(C_3)). \end{aligned} \quad (7.10)$$

Algorithm 7.1 Offline FPE training algorithm**Require:**

Fault Prototype Encoder E
 Dataset used for training $\{W_t, S_t, \hat{y}_t\}_{t=1}^T$
 Step size α , Evolutionary hyperparameter ϵ
 Iteration limit L

- 1: Initialize weights in E . Set $l \leftarrow 0$
- 2: Randomly initialize class prototypes $\{P_0^C, \dots, P_c^C\}$
- 3: **do**
- 4: **for** ($t = 1$ to T)
- 5: $D^A, P \leftarrow E(S_t, W_t)$
- 6: $L_1 = \sum_{i=1}^m (\mathbb{1}(\hat{y}_{t,i} = 0) \log(D^A[0]) + \mathbb{1}(\hat{y}_{t,i} > 0) \log(D^A[1]))$
- 7: $L_2 = \sum_{i=1}^m \mathbb{1}(\hat{y}_{t,i} > 0) (\|P_i - P_{\hat{y}_{t,i}}^C\|_2 - (\sum_{j \neq \hat{y}_{t,i}} \|P_i - P_j^C\|_2))$
- 8: **for** ($i = 1$ to m)
- 9: **if** ($\|P_i - P_{\hat{y}_{t,i}}^C\|_2 = \min_j \|P_i - P_j^C\|_2$)
- 10: $P_{\hat{y}_{t,i}}^C \leftarrow (1 - \alpha) \cdot P_{\hat{y}_{t,i}}^C + \alpha \cdot P_i$
- 11: Update weights of E using L_1, L_2
- 12: $\alpha \leftarrow (1 - \epsilon) \cdot \alpha$
- 13: $l \leftarrow l + 1$
- 14: **while** $l < L$

We then pass this encoding through two decoders for each host h_i ,

$$\begin{aligned}
 D_{T,i}^A &= \text{softmax}(\text{FeedForward}(C_{4,i})), \\
 P_{T,i} &= \text{sigmoid}(\text{FeedForward}(C_{4,i})),
 \end{aligned} \tag{7.11}$$

where $D_{T,i}^A$ denotes the fault prediction and $P_{T,i}$ is a prototype embedding for each host. Finally, we define the embedding for host h_i as

$$E_{T,i}^F = \begin{cases} P_{T,i}, & \text{if } D_{T,i}^A[1] \geq D_{T,i}^A[0] \\ [0]_{1 \times E}, & \text{otherwise.} \end{cases} \tag{7.12}$$

We stack all host embeddings to generate E_T^F as the output of the FPE of PreGAN+. Unlike PreGAN that has GRU based temporal inference, a Transformer based model not only allows us to train FPE in a data and time efficient manner, it also enables us to give a broader context as C_1 of the historic host characteristics in the edge environment albeit with higher memory overheads. This allows us to frequently tune the FPE in an online fashion, possibly at the start of each interval.

7.2.2 Offline FPE Training in PreGAN and PreGAN+

We now describe the training process for the FPE to detect and classify faults, summarized in Algorithm 7.1. We utilize a supervised training approach assuming sufficient compute time available at training. To train the encoder, we first collect a dataset of input windows, scheduling decisions and fault class labels $\{W_t, S_t, \hat{y}_t\}_{t=1}^T$ by running the system without any preemptive migration. Here, $\hat{y}_{t,i}$ is the class label for h_i , with 0 indicating no fault. Now, the FPE encoder (denoted as E) generates the outputs D^A and P from the inputs W_t and S_t (line 4 in Alg. 7.1). We use the cross-entropy loss for fault detection (line 6 in Alg. 7.1). Here, $\hat{y}_{t,i}$ is used to

generate the ground-truth labels and the loss is summed over all hosts:

$$L_1 = \sum_{i=1}^m \left(\mathbb{1}(\hat{y}_{t,i}=0) \log(D^A[0]) + \mathbb{1}(\hat{y}_{t,i}>0) \log(D^A[1]) \right). \quad (7.13)$$

This loss allows us to modify the weights of the FPE encoder such that, at convergence, the model correctly assigns $D^A[1] > D^A[0]$ that denotes the presence of a fault as assumed in Equations (7.6) and (7.12).

For fault-classification, we use a triplet loss. We first define class prototype vectors for each fault class $\{P_0^C, \dots, P_c^C\}$, that are randomly initialized from $[0, 1]^E$. Just like the means in a k-means clustering approach, these vectors denote the centroids of the embeddings of their respective classes [Snell et al., 2017]. As is common in prototypical networks used in few-shot learning, the new embedding generated by the model belongs to the class which has the least Euclidean distance from the class prototypes. The triplet loss aims at reducing the distance of the output embedding from the true class prototype and increasing it from the false class ones (line 7 in Alg. 7.1). Thus, for all those hosts with a true fault, the loss includes the L2-norm between the output embedding and the true class prototype and negative L2-norm between the output and false class prototypes:

$$L_2 = \sum_{i=1}^m \mathbb{1}(\hat{y}_{t,i} > 0) \left(\|P_i - P_{\hat{y}_{t,i}}^C\|_2 - \sum_{j \neq \hat{y}_{t,i}} \|P_i - P_j^C\|_2 \right). \quad (7.14)$$

Another step in the training process is to update the class prototypes. If an output embedding belongs to the correct class, we can update the prototype of that class as

$$P_{\hat{y}_{t,i}}^C \leftarrow (1 - \alpha) \cdot P_{\hat{y}_{t,i}}^C + \alpha \cdot P_i, \quad (7.15)$$

where α is the step-size. We exponentially decay our step-size to ensure that the model learns class prototypes quickly initially and converges to a stable prototype set using a hyperparameter based decay (line 11 in Alg. 7.1). The triplet loss in conjunction with the prototype updates allow the prototypes to be distant from one another, making each prototype a characteristic identifier for that class.

7.2.3 Online FPE Tuning in PreGAN+

We now describe how we periodically fine-tune the FPE in the PreGAN+ model, summarized in Algorithm 7.2. We leverage a semi-supervised approach, where we generate labels for a fraction of the datapoints (denoted by ξ) to guide unsupervised clustering. Our approach is motivated from *constrained K-Means* [Basu et al., 2002, Shukla et al., 2020]. We assume that within the ξ fraction of labeled datapoints, there is at least one from each class. We impose an assumption, common in prior work [Rawat et al., 2021, Liu et al., 2016, Sharif et al., 2020], that the fraction of faulty datapoints is consistent across data. Thus, for all labelled data, we find the fraction of no-fault datapoints as

$$\Gamma = \frac{\mathbb{1}(\hat{y}_{t,i}=0)}{\mathbb{1}(\hat{y}_{t,i}>0)} \quad (7.16)$$

Now, for each prototype vector P_u without any label, we calculate the sum distance with each class prototype.

$$f_d(P_u) = \sum_{j=1}^c \|P_u - P_j^C\|. \quad (7.17)$$

Algorithm 7.2 Online PreGAN+ FPE Tuning algorithm**Require:**

Fault Prototype Encoder E
 For timestep t , labeled L and unlabeled U datasets
 Step size α , Evolutionary hyperparameter ϵ
 Iteration limit L
 1: Initialize weights in E . Set $l \leftarrow 0$
 2: Initialize class prototypes $\{P_0^C, \dots, P_c^C\}$ to labelled vectors
 3: **do**
 4: $D^A, P \leftarrow E(S_t, W_t)$
 5: $L_1 = \sum_{i \in L} (\mathbb{1}(\hat{y}_{t,i} = 0) \log(D^A[0])$
 $\quad + \mathbb{1}(\hat{y}_{t,i} > 0) \log(D^A[1]))$
 $+ \sum_{i \in U} ((1 - f_l(P_i)) \log(D^A[0])$
 $\quad + f_l(P_i) \log(D^A[1]))$
 6: $L_2 = \sum_{i \in L} \mathbb{1}(\hat{y}_{t,i} > 0) (\|P_i - P_{\hat{y}_{t,i}}^C\|_2$
 $\quad - (\sum_{j \neq \hat{y}_{t,i}} \|P_i - P_j^C\|_2))$
 $+ \sum_{i \in U} f_l(P_i) (\|P_i - P_{f_c(P_i)}^C\|_2$
 $\quad - \sum_{j \neq f_c(P_i)} \|P_i - P_j^C\|_2)$
 7: **for** ($i = 1$ to m)
 8: **if** ($\|P_i - P_{\hat{y}_{t,i}}^C\|_2 = \min_j \|P_i - P_j^C\|_2$)
 9: $P_{\hat{y}_{t,i}}^C \leftarrow (1 - \alpha) \cdot P_{\hat{y}_{t,i}}^C + \alpha \cdot P_i$
 10: Update weights of E using L_1, L_2
 11: $\alpha \leftarrow (1 - \epsilon) \cdot \alpha$
 12: $l \leftarrow l + 1$
 13: **while** $l < L$

We then take the top Γ fraction of the datapoints with the highest f_d output. We give these as fault-label 0 and the rest as 1. This function is represented as $f_l(P_u)$ for a vector P_u . The intuition is as follows. Considering that Γ fraction of datapoints in a dataset have no faults, the prototype vector for such datapoints should not belong to any fault class. This has been shown to translate to having the highest distances from the class representatives, namely, class prototypes [Shukla et al., 2020]. Thus, the top Γ fraction of datapoints with the highest distance as calculated in equation (7.17) denote datapoints with fault label 0. We assume that at each timestep the hosts with known labels have the index set L and those with unlabelled classes U , such that $L \cup U = \{1, \dots, m\}$. Thus, in such settings, the loss function for fault-detection becomes

$$\begin{aligned}
 L_1 = & \sum_{i \in L} \left(\mathbb{1}(\hat{y}_{t,i} = 0) \log(D^A[0]) + \mathbb{1}(\hat{y}_{t,i} > 0) \log(D^A[1]) \right) \\
 & + \sum_{i \in U} \left((1 - f_l(P_i)) \log(D^A[0]) + f_l(P_i) \log(D^A[1]) \right).
 \end{aligned} \tag{7.18}$$

Thus, we use the prototypical vectors to *estimate* the fault class of the unlabeled datapoints.

For classification, in lieu of initializing the class prototypes randomly, we use the labeled datapoints and sample uniformly at random to set the initial class prototype vectors. Now, for each prototype vector P_u without any label, we define the corresponding class as

$$f_c(P_u) = \arg \min_j \|P_u - P_j^C\|, \tag{7.19}$$

which translates to the class for which the class prototype is the closest as per L2 distance. We use this *estimated*

Algorithm 7.3 The GAN training algorithm**Require:**

-
- Pretrained FPE Encoder E
 - Generator and Discriminator networks $Gen, Disc$
 - Co-Simulator Sim
 - 1: Initialize weights in $Gen, Disc$.
 - 2: **for** ($t = 1$ to T)
 - 3: $D^A, P \leftarrow E(S_t, W_t)$
 - 4: Get E^F by (7.6)
 - 5: $\Delta = Gen(S_t, E^F)$ ▷ Generative preemptive migration
 - 6: $N = S_t + \Delta$ ▷ Form new scheduling decision
 - 7: $D = Disc(S_t, N)$ ▷ Compare the two decisions
 - 8: Calculate L_D using (7.21)
 - 9: Update weights of $Disc$ using L_D
 - 10: Calculate L_G using (7.22)
 - 11: Update weights of Gen using L_G
-

class label for unlabeled datapoints, which gives us the classification loss as

$$\begin{aligned}
 L_2 = & \sum_{i \in L} \mathbb{1}(\hat{y}_{t,i} > 0) \left(\|P_i - P_{\hat{y}_{t,i}}^C\|_2 - \sum_{j \neq \hat{y}_{t,i}} \|P_i - P_j^C\|_2 \right) \\
 & + \sum_{i \in U} f_l(P_i) \left(\|P_i - P_{f_c(P_i)}^C\|_2 - \sum_{j \neq f_c(P_i)} \|P_i - P_j^C\|_2 \right).
 \end{aligned} \tag{7.20}$$

Thus, we use the loss function as shown in equation (7.14) for the labeled datapoints and the estimated fault labels for the unlabeled datapoints.

7.2.4 Online GAN Training using Co-Simulation

We now describe how we use the FPE encoder and a co-simulation engine to train the GAN model both for the PreGAN and PreGAN+ models, summarized in Algorithm 7.3. As event-based co-simulators can provide us QoS estimates quickly, they allow us to avoid running decisions on the physical setup, saving time and execution costs. Thus, unlike a traditional GAN model, we train our discriminator in a self-supervised manner using a co-simulator.

Our GAN training runs in an online fashion instead of using a pre-collected dataset. For an input pair (S, W) , we run the PreGAN model to obtain N and S . We run the co-simulator with the two scheduling decisions S and N and obtain the QoS scores. These scores may be calculated using a combination of metrics like energy consumption, response time and SLO violations [Basu et al., 2019]. We denote the co-simulated QoS scores by $Sim(S)$ and $Sim(N)$. To make the likelihood score output of the discriminator correspond to the co-simulated scores, we use the binary cross-entropy loss as:

$$\begin{aligned}
 L_D = & \frac{1}{2} \sum_{i=1}^m \left(\mathbb{1}(Sim(N) \geq Sim(S)) \left(\log(D[1]) + \log(1 - D[0]) \right) \right. \\
 & \left. + \mathbb{1}(Sim(N) < Sim(S)) \left(\log(D[0]) + \log(1 - D[1]) \right) \right),
 \end{aligned} \tag{7.21}$$

where the loss does not propagate to the generator (fixing N). This pushes the discriminator to give a higher likelihood score to the decision with a higher simulated QoS score. To train the generator, we use the adversarial

Algorithm 7.4 The PreGAN/PreGAN+ testing algorithm**Require:**

```

  Pretrained models  $E, Gen, Disc$ 
1: for ( $t = 1$  to  $T$ )
2:    $D^A, P \leftarrow E(S_t, W_t)$ 
3:   Get  $E^F$  by (7.6)
4:    $\Delta = Gen(S_t, E^F)$  ▷ Generative preemptive migration
5:    $N = S_t + \Delta$  ▷ Form new scheduling decision
6:    $D = Disc(S_t, N)$  ▷ Compare the two decisions
7:   if PreGAN+
8:     Fine-Tune  $E$  using Algorithm 7.2
9:     Fine-Tune  $G$  and  $D$  using Algorithm 7.3
10:  Execute final schedule obtained from (7.9)
11:  if ( $D[1] \geq D[0]$ )
12:    return  $N$  ▷ Return new decision if higher score
13:  return  $S$ 

```

loss

$$L_G = \frac{1}{2} \sum_{i=1}^m \left(\log(D[1]) + \log(1 - D[0]) \right), \quad (7.22)$$

where the loss propagates to the generator with the discriminator weights kept fixed. This is equivalent to Eq. (7.21) but with $\text{Sim}(N) \geq \text{Sim}(S)$, pushing the generator to output a migration decision Δ which gives a schedule N better than the original schedule S . To do this, the generator gets fault class embeddings from the FPE encoder. Akin to prior work [Basu et al., 2019], we use a convex combination of energy consumption and average response time as the Sim score. Specifically, for a decision D , we denote the average energy consumption by AEC_D and average response time as ARC_D . Thus,

$$\text{Sim}(D) = \beta \cdot AEC_D + (1 - \beta) \cdot ARC_D. \quad (7.23)$$

This style of training a discriminator model has two benefits: it allows us to (1) run our model inference without the co-simulation at test time, eliminating the computationally expensive generation of simulated traces, (2) train our preemptive migration generator by backpropagating the adversarial loss to the generator network.

7.2.5 Online Inference

We now describe the inference procedure using the trained PreGAN or PreGAN+ models (summarized in Algorithm 7.4). For any input time-series window and scheduling decision W, S , the PreGAN model outputs D, N . Based on the likelihood scores of the discriminator D , PreGAN predicts if the preemptive migration decision would improve QoS or not. If it does, *i.e.*, $D[1] \geq D[0]$, N is executed, otherwise the original decision S is executed (lines 11-13 in Alg. 7.4). Also, in case of PreGAN+, we fine-tune the FPE encoder using new data with fixed ξ fraction as labeled (process described in Section 7.4.3.1, where ξ is a hyper-parameter). ξ controls the overhead imposed by the fault labelling engine and the amount of labelled data available to tune the FPE encoder of PreGAN+. We also tune GAN using the co-simulating both decisions S and N .

Overall, the FPE in PreGAN allows us proactively predict faults and guide the GAN model to generate a preemptive migration decision over the schedule generated by a policy oblivious to future system faults. The discriminator of the GAN then performs a cost-benefit analysis over the original scheduling decision to avoid

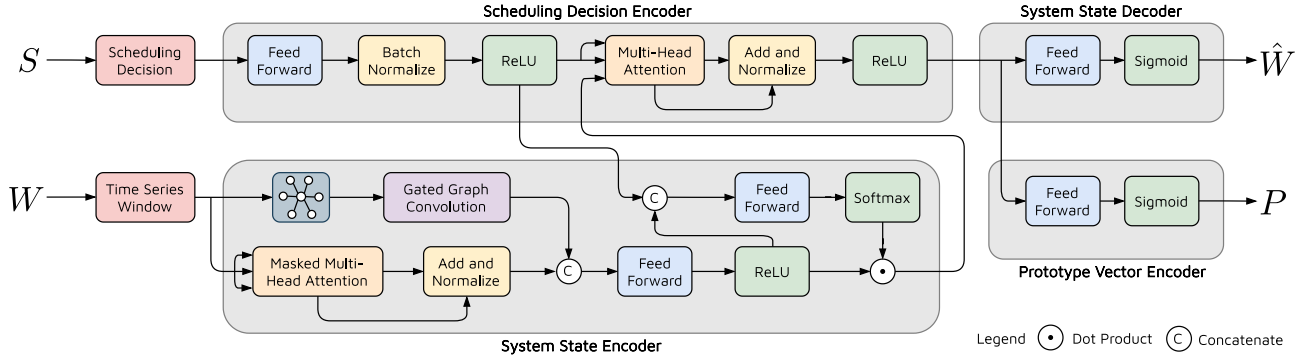


Figure 7.3: The DeepFT Surrogate Model.

excessive migration overheads. This allows PreGAN to optimize scheduling decisions and cut down execution costs.

7.3 DeepFT Unsupervised Model

As discussed in the previous subsection, for an input system sliding window W and scheduling decision S , the surrogate model of DeepFT predicts a future system state $\hat{W}$ and fault classification prototype vector P . The surrogate model used in DeepFT is a composite neural network presented in Figure 7.3. The DeepFT model has four main components: (1) Scheduling Decision Encoder, (2) System State Encoder, (3) System State Decoder and (4) Prototype Vector Encoder.

Considering we have p tasks in the system as a sum of new and active tasks, the scheduling decisions of these are encoded as one-hot vectors of size m (number of hosts). We thus get a matrix of scheduling decisions (S) of size $[p \times m]$. Also, the n features of m hosts and p tasks in the form of resource utilization metrics, each state window (W) is encoded as a $[(m + p) \times n \times k]$ tensor. All operations are performed in a factored fashion, *i.e.*, the neural network operates on S as a batch of p vectors, each having dimension m , and on W as a batch of size $m + p$ tensors of size $n \times k$.

Scheduling Decision Encoder. The decision encoder encapsulates the scheduling decision in the form of a compressed vector representation. This representation uses an encoding of the time-series window (E^W generated by the state encoder) as an input to facilitate the prediction of the next window and the fault class prototype embeddings. To do this, we utilize a feed-forward layer to bring down the dimension size of the input with batch-normalization and rectified linear unit (ReLU) activation. Thus,

$$E^S = \text{ReLU}(\text{BatchNorm}(\text{FeedForward}(S))). \quad (7.24)$$

Here, a feed-forward layer performs a matrix-multiplication operation with a tunable set of parameters. A batch-normalization layer normalizes the input data across each dimension to zero mean and unit variance, which ensures faster and more robust training [Ioffe and Szegedy, 2015].

We then use a multi-head self-attention operation [Vaswani et al., 2017] with the window encoding E^W to provide a representation that focuses on only a subset of the tasks and hosts that may result in faults, relieving the downstream predictors from inferring over tasks/hosts that have low fault probability. Self-attention passes the input through a feed-forward layer to generate a set of weights, which are then used to take a convex

combination of the encodings of multiple tasks in E^W . In the multi-headed version, the input is passed through multiple such feed-forward layers to capture the correlations across multiple such task sets. This self-attention technique allows *the designed model to scale efficiently* with the number of hosts or active tasks in the system (demonstrated with a visualization in Section 7.3.0.1). This with the layer-normalization operation gives

$$\begin{aligned} E_1^S &= \text{Norm}(E^S + \text{MultiHeadAtt}(E^S, E^S, E^W)), \\ E_2^S &= \text{ReLU}(E_1^S). \end{aligned} \quad (7.25)$$

System State Encoder. The state encoder creates a succinct encoding of the system state time-series window, conditioned on the task scheduling decision to facilitate the prediction of the next system state window. We first form a graph using the task schedule S , such that there is an edge from host h_i to host h_j if there is a task migration from h_i to h_j in schedule S . The n characteristics of each host in x_t are then used to populate the feature vectors of the nodes in the graph. We denote the feature vector of host h_i as e_i . We then pass the graph through a gated-graph convolution network to capture the inter-host dependencies rising from the new task allocation S . Here, the features for host h_i are aggregated over one-step connected neighbors $n(i)$ in the graph over r convolutions, resulting in an embedding e_i^r for each host node in the graph. Specifically, the gating stage is realized as a Gated Recurrent Unit (GRU) resulting in *graph-to-graph* updates [Ruiz et al., 2020] as:

$$\begin{aligned} e_i^0 &= \tanh(W e_i + b), \\ x_i^q &= \sum_{j \in n(i)} W^q e_j^{q-1}, \\ e_i^q &= \text{GRU}(e_i^{q-1}, x_i^q), \end{aligned} \quad (7.26)$$

where the second equation performs the convolutions of the features of immediate neighbors in the graph. However, for large-scale graphs, to ensure that we capture the inter task and host correlations, we perform the above convolution step r times. Here, a GRU is a recurrent neural network that decides the weightage of the output of the previous convolution iteration with respect to the latest iteration. This allows the model to efficiently scale with the size of input graph without significantly losing performance. The stacked representation for all hosts is represented as E^H . We also pass the time-series window through a multi-head self attention network.

$$\begin{aligned} E_1^W &= \text{Mask}(\text{MultiHeadAtt}(E^W, E^W, E^W)), \\ E_2^W &= \text{Norm}(W + E^W). \end{aligned} \quad (7.27)$$

The first attention operation is masked to prevent the encoder from looking at the datapoints for future time-series values at the time of training as all time-series windows are given at once to allow parallel training. The resulting window encoding is denoted as E_2^W . We then apply dot-product attention [Bahdanau et al., 2015] conditioned on the decision encoding E^S , giving the final window encoding as

$$\begin{aligned} E_3^W &= \text{ReLU}(\text{FeedForward}([E_2^W, E^H])), \\ \text{attn} &= \text{softmax}(\text{FeedForward}([E_2^W, E^S])), \\ E^W &= \text{attn} \cdot E_3^W. \end{aligned} \quad (7.28)$$

This attention is similar to the multi-headed attention operation without multiple heads and also uses E^S as the input of the feed-forward layer, allowing us to generate an output conditioned on the input scheduling decision S .

System State Decoder. Now that we have the embedding that captures both decision and state E_2^S , we generate the reconstructed window as

$$\hat{W} = \text{sigmoid}(\text{FeedForward}(E_2^S)). \quad (7.29)$$

Here, the sigmoid function allows us to bring the output in the range $[0, 1]$, the same as that of the normalized true window (W). To generate the fault score we only consider upward spikes of the true window from the reconstructed window. This is due to the nature of our state data, *i.e.*, resource utilization metrics leading to faults only when there is a sudden increase in CPU/RAM/disk/network consumption. The ReLU activation is apt for this as it gives a zero fault score when the true utilization metrics are lower than the predicted ones. Moreover, the fault detection label is decided by comparing the fault score with a threshold. We use Peak-Over-Threshold (POT) [Siffer et al., 2017] to generate the fault label for input W . POT is a dynamic thresholding technique that uses extreme value theory to set thresholds for each input dimension. Thus, our fault score and label (1 if faulty and 0 otherwise) are calculated as

$$\begin{aligned} f &= \|\text{ReLU}(W - \hat{W})\|^2, \\ l &= \mathbb{1}(f > \text{POT}(W)). \end{aligned} \quad (7.30)$$

Prototype Vector Encoder. Our prototype encoder is motivated by prototypical networks proposed in prior work [Snell et al., 2017]. Such networks are standard few-shot learning models for supervised clustering of the input data. Few-shot learning is a paradigm in which models aim to achieve the required task using only a few data points. This allows the DeepFT model to adapt to changing environments in a few scheduling intervals. Prototypical networks perform clustering by maintaining representative prototype vectors for each class. Such networks generate a prototype embedding for an input and ensure that the output vector is close to the prototypes for the correct class and far from the prototypes of the incorrect classes. The prototype embedding is generated from the E_2^S embedding as

$$P = \text{sigmoid}(\text{FeedForward}(E_2^S)). \quad (7.31)$$

7.3.0.1 Offline Model Training

We now describe the training process for the DeepFT surrogate model, summarized in Algorithm 7.5. To do this, we collect a dataset of scheduling decisions and time-series window pairs $\{W_t, S_t\}_{t=1}^T$. To do this, we use a random scheduler to cover as much of the decision space as we can. We use the model to find the prototype vectors of each data pair and calculate their mean and standard deviations for each class c_i as $\{(\mu_i, \sigma_i)\}_{i=0}^j$ (line 3). For an input pair (W_t, S_t) , the model generates an output $(\hat{W}_t, P_t)$ (line 5). Now, for the state decoder, we define the reconstruction loss as the mean-square-error

$$L_R = \|\hat{W}_t - W_{t+1}\|^2. \quad (7.32)$$

Algorithm 7.5 The DeepFT offline training algorithm**Require:**

```

  Deep Surrogate Model  $M$ 
  Dataset used for training  $\{W_t, S_t\}_{t=1}^T$ 
  Iteration limit  $L$ 
1: Initialize weights in  $M$ . Set  $l \leftarrow 0$ 
2:   do
3:     Calculate  $(\mu_i, \sigma_i)$  for each class  $\{c_i\}_{i=0}^j$ 
4:     for  $(t = 1 \text{ to } T)$  do
5:        $\hat{W}_t, P_t \leftarrow M(W_t, S_t)$ 
6:       Calculate  $l_t$  using (7.30)
7:       if  $(l_t)$  do
8:          $\phi = \arg \min_{i=1}^j D(P_t, c_i)$ 
9:       else
10:         $\phi = 0$ 
11:         $L_R = \|\hat{W}_t - W_{t+1}\|^2$ 
12:         $L_T = D(P_t, c_\phi) - \sum_{i \neq \phi} D(P_t, c_i)$ 
13:        Update weights of  $M$  using  $L_R + L_T$ 
14:         $l \leftarrow l + 1$ 
15:   while  $l < L$ 
16: return trained  $M$ 

```

Minimizing this loss ensures that our model generates a precise prediction of the next state.

The prototype encoder outputs the embedding P_t . In our case, true class labels are *a-priori* unknown. To circumvent this issue, we use the fault prediction from the state decoder. Thus, we first generate the fault prediction label l_t using (7.30). If a faulty state is predicted, we label it as one of the j classes $\{c_1, \dots, c_j\}$. Here, j is a hyperparameter corresponding to the different types of faults in an edge worker. Motivated by [Medina et al., 2020], to generate the ground-truth class in the case that an anomaly is detected by the state decoder, we choose the class corresponding to the closest class prototype from the output P_t (line 8). This class label may not correspond to any semantically defined fault, but helps the model implicitly segregate faults and improve learnability [Medina et al., 2020]. If there is no fault, we call it the No-Anomaly-Prototype (NAP) class, denoted as c_0 (line 10). This training style requires only a few datapoints to generate a class prototype as the centroid of the points in that class; hence, it is called few-shot learning.

As the fault distribution is highly skewed in typical executions with most states belonging to the NAP class, we use the Kullback–Leibler divergence between the output P_t and a class representative pair (μ_i, σ_i) :

$$D(P_t, c_i) = \frac{(\mu_i - P_t)^2}{2\sigma_i^2} + \frac{1}{2} \ln \sigma_i^2. \quad (7.33)$$

Unlike the Euclidean distance between the predicted prototype P_t and the mean of the class μ_i , this divergence accounts for the overall distribution of the class vectors. The ground-truth class (c_ϕ) is then determined as

$$c_\phi \text{ s.t. } \phi = \begin{cases} \arg \min_{i=1}^j D(P_t, c_i), & \text{if } l_t \\ 0, & \text{otherwise.} \end{cases} \quad (7.34)$$

To train the model, we apply the triplet loss [Snell et al., 2017] to ensure the predicted prototype embedding is closer to the true class c_ϕ (minimizing $D(P_t, c_\phi)$) and far from the incorrect class vectors $c_i \forall i \neq \phi$ (maximizing

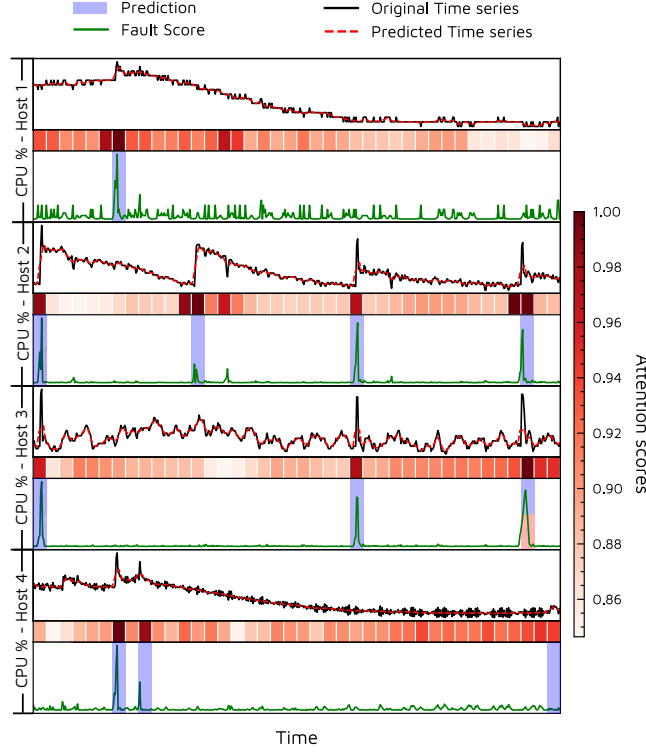


Figure 7.4: Visualization of attention weights, truncated with the CPU utilization of four hosts in the system.

Algorithm 7.6 The DeepFT scheduling algorithm

Require:

- Trained Deep Surrogate Model M
 - Use converged $\{(\mu_i, \sigma_i)\}_{i=0}^j$ from training
 - 1: **for** $(t = 1 \text{ to } T)$
 - 2: $S_t \leftarrow S_{t-1}$ **if** $t > 0$ **else** initialize randomly
 - 3: $\hat{W}_t, P_t \leftarrow M(W_t, S_t)$
 - 4: $W_{t+1} \leftarrow \text{Sim}(W_t, S_t)$
 - 5: $L_O = \|\text{ReLU}(W_{t+1} - \hat{W}_t)\|^2 + D(P_t, c_0)$
 - 6: Optimize S_t using L_O and Adam optimizer
 - 7: Update weights of M using $L_R + L_T$ in Eqs.(7.32),(7.35)
 - 8: **return** S
-

$$\sum_{i \neq \phi} D(P_t, c_i):$$

$$L_T = D(P_t, c_\phi) - \sum_{i \neq \phi} D(P_t, c_i). \quad (7.35)$$

Visualization of attention weights. Figure 7.4 visualizes the attention weights averaged across the heads of the attention operation in the scheduling decision encoder. The model is trained using a dataset collected from a physical setup (see details in Section 7.4) using a random scheduler. We show the ground-truth (in black) and predicted time-series (in red) using the first state vector of the reconstructed window. The fault scores are also shown in green. The red heatmap shows the average attention weights for each window. The attention operation compresses the scheduling decision matrix to a fixed size vector allowing the model to scale to higher number of hosts or workloads in the system. For instance, the number of parameters of the model increases by only 3% or 1% for each new task or host in the system. Moreover, there is a high correlation between the fault scores and attention weights, showing how attention facilitates focusing on only those dimensions that may result in faults. The fault labels are also highlighted using the POT thresholds.

7.3.1 Scheduling and Online Model Fine-Tuning using Co-Simulation

We now describe how we use the surrogate model to generate scheduling decisions for fault-tolerant computing, summarized in Algorithm 7.6. Our method is motivated by GOBI approach introduced in Chapter 3. At the beginning of interval I_t , we start from S_{t-1} (if $t > 0$ otherwise initialize randomly) and optimize it using DeepFT’s surrogate model, denoted as M (line 2). Compared to initializing S_t randomly, this helps us to reduce the number of migrations. Also, to generate our self-supervision next-state W_{t+1} at test time, we use a co-simulator to find the next state using the current state and scheduling decision, denoted as $W_{t+1} \leftarrow \text{Sim}(W_t, S_t)$ (line 4). Our co-simulator runs a single-step execution trace of the scheduling decision S_t using the workload characteristics W_t to estimate the system state at the start of the next interval, *i.e.*, W_{t+1} . For input (W_t, S_t) and model output $\hat{W}_t, P_t \leftarrow M(W_t, S_t)$, we define optimization loss as

$$L_O = \|\text{ReLU}(W_{t+1} - \hat{W}_t)\|^2 + D(P_t, c_0). \quad (7.36)$$

Optimizing using this loss minimizes the fault score and updates the prototype vector to move towards the NAP class. As we update the scheduling decisions to minimize L_O , when converged, we hypothesize that this decision should facilitate fault avoidance. To optimize the scheduling decision S_t , we can use the stochastic gradient descent

$$S_t \leftarrow S_t - \nabla_{S_t} L_O. \quad (7.37)$$

However, advances like root-mean-square propagation and adaptive gradients facilitate faster convergence; hence, we utilize the Adam optimizer [Kingma and Ba, 2014] with cosine annealing [Loshchilov and Hutter, 2016] to optimize the scheduling decision. Thus, running an optimization over the scheduling decision S_t to minimize L_O should, in principle, give us a scheduling decision, say S_t^* , such that P_t with S_t^* as input belongs to the NAP class and hence, it does not lead to a contention-like fault in the subsequent interval. The final converged decision is applied to schedule and migrate tasks. As we do not need the ground-truth labels, we can fine-tune the surrogate model on-the-fly at test time (line 7 of Alg. 7.6).

7.4 Performance Evaluation

7.4.1 Evaluation Setup

Our evaluation testbed is a cluster with 16 Raspberry Pi 4B nodes (see Figure 7.5). Our cluster contains 8 nodes with 4-GB RAM and another 8 but with 8-GB RAM. The power consumption models are taken from the commonly-used SPEC benchmarks repository [Standard Performance Evaluation Corporation]. We run all experiments for 100 scheduling intervals, with each interval being 300 seconds long, giving a total experiment time of 8 hours 20 minutes. We average over 5 runs and use diverse workload types to ensure statistical significance.

For our workloads, we consider the *DeFog* applications: Yolo, PocketSphinx and Aeneas [McChesney et al., 2019]. DeFog is a fog computing benchmark suite that consists of various real-world application instances. The three specific applications used in our experiments were considered due to their heterogeneous resource requirements and volatile characteristics. At the beginning of every scheduling interval, we create $Poisson(\lambda)$



Figure 7.5: Raspberry Pi Cluster as evaluation platform.

new workloads with $\lambda = 5$, sampled uniformly from the three applications. Poisson distribution is a natural choice for a bag-of-tasks workload model, common in edge environments [Mao et al., 2016, Basu et al., 2019].

We emulate mobility of the edge nodes using the **NetLimiter** tool that varies the communication latency of each worker node with the broker using the mobility model described by Gilly et al. [Gilly et al., 2020]. Specifically, using the mobility traces from the Simulation of Urban Mobility (SUMO) tool [Krajzewicz et al., 2012], we set the latency and bandwidth parameters of each edge host. SUMO simulates mobile vehicles in a smart-city like environment. SUMO provides us with values of the `ping` time and network bandwidth for each node in the environment that we set using **NetLimiter**.

7.4.2 Evaluation Metrics

We now provide details on the evaluation metrics for supervised and unsupervised fault-tolerance.

7.4.2.1 PreGAN and PreGAN+

For fault detection, we consider detection accuracy, precision, recall and F1 scores. For a test datapoint $\{W_t, S_t, \hat{y}_t\}$ with the PreGAN or PreGAN+ outputs D and N , the predicted and ground-truth labels are obtained as

$$\bigwedge_{i=1}^m \mathbb{1}(D[1] > D[0]) \text{ and } \bigwedge_{i=1}^m \mathbb{1}(\hat{y}_{t,i} > 0)$$

respectively. This ground-truth label is 1 when any of the edge hosts has a fault and 0 otherwise. All data for training is generated using the co-simulator.

For fault diagnosis, we consider two commonly used metrics [Zhao et al., 2020]: (1) HitRate@100% is the measure of how many ground truth dimensions have been included in the top candidates predicted by the model [Su et al., 2019], (2) Normalized Discounted Cumulative Gain (NDCG@100%) [Järvelin and Kekäläinen, 2002]. For fault classification, we consider the classification accuracy. For test time, we also consider an “improvement ratio”, which for an execution of T intervals is calculated as:

$$\text{Improvement Ratio} = \frac{1}{T} \sum_{t=1}^T \mathbb{1}(D[1] > D[0]). \quad (7.38)$$

This metric denotes the ratio of the times the model is able to predict a better scheduling decision than the

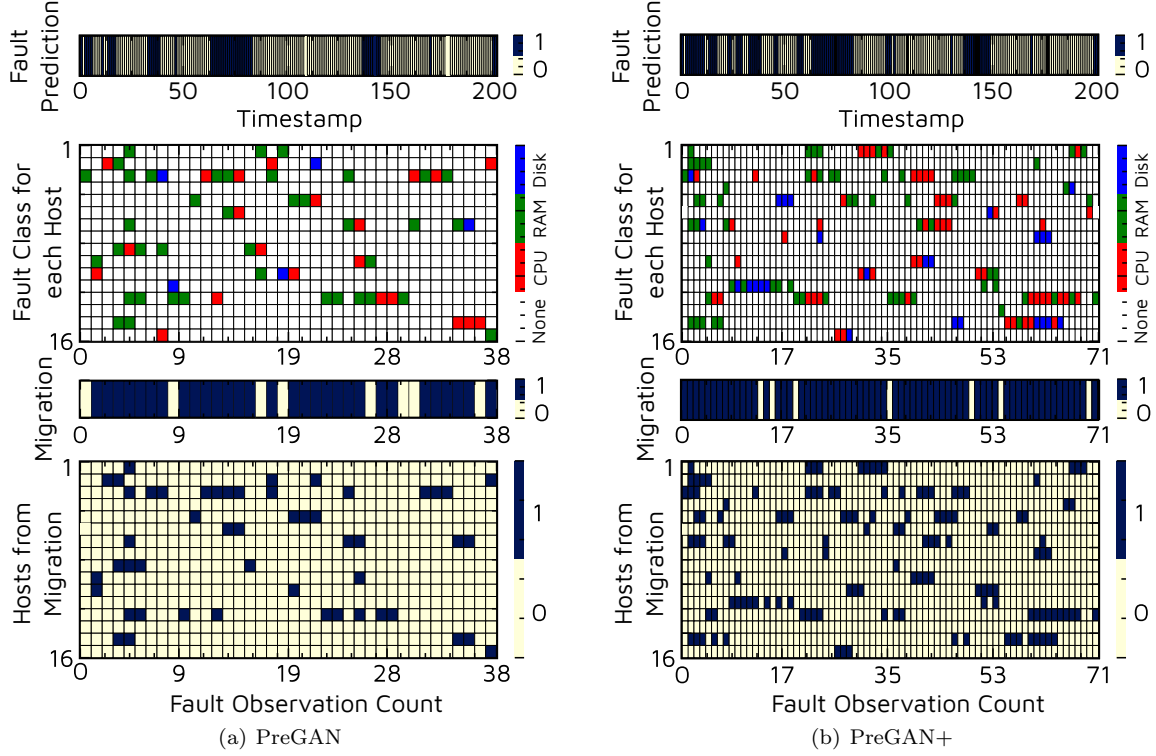


Figure 7.6: Visualization of fault prediction, classification and migration decisions with execution intervals.

original. This gives us an indication of the how well the preemptive migration prediction performs compared to the original scheduling decision. Together with the detection accuracy this gives us a complete picture on how well the model performs compared to the case where there is no preemptive migration. To explain these metrics with a simple visualization example on 200 intervals, consider Figure 7.6. The top heatmap denotes the intervals at which the model predicted a fault (fault denoted with black and yellow otherwise), specifically 38 intervals out of 200 for PreGAN and 71 for PreGAN+. The second heatmap shows the fault class prediction for each host (if any). The third heatmap shows the intervals in which the likelihood score of N was higher than S (30 out of 38 for PreGAN and 64 out of 71 for PreGAN+) and the fourth heatmap shows the hosts from which there was a task migrated. Here, the improvement ratio for PreGAN is $30/38 = 0.7895$ and for PreGAN+ is $64/71 = 0.9014$. Also, we observe a strong correlation between the fault class predictions and the host selected for task migration, thanks to the factored prediction in PreGAN and PreGAN+.

We also consider standard QoS metrics including energy consumption, response time, fraction of SLO violations, migration count and resource utilization. Further, to compare the overheads of the models, we also measure the ratio of the time to update input scheduling decision to the total decision making time and call this the “overhead ratio”.

7.4.2.2 DeepFT

To collect the training data for the surrogate model, we use a random scheduler. To test the fault detection and diagnosis performance, we utilize historical log data collected by running the GOBI approach (introduced in Chapter 3) on our testbed. To generate the ground-truth fault labels, we consider the Anomaly Detection Engine for Linux Logs (ADE) tool [Open Mainframe Project]. For every state of the dataset, we use the fault

flag from the ADE tool for the subsequent state as our ground-truth label. This fault flag is raised when one or more of the metrics like CPU, RAM, disk and network utilization is/are greater than dynamically set thresholds. These may be caused due to memory leaks, buffer overloads and resource throttling.

To evaluate fault-detection efficacy, we utilize commonly considered metrics including accuracy, precision, recall and F1 score. For fault-diagnosis (*i.e.*, the prediction of the specific hosts that have faults), the factored style prediction of fault scores and labels enables us to output labels for each host independently. We use two popular metrics for comparison: (1) HitRate@100% is the measure of how many ground truth dimensions have been included in the top candidates predicted by the model [Su et al., 2019], (2) Normalized Discounted Cumulative Gain (NDCG@100%) [Järvelin and Kekäläinen, 2002]. To also compare how well the model is performing compared to our reference scheduler GOBI that is used to collect our test dataset, we present a more insightful metric called the “Improvement Ratio”. For any model X , we calculate this using the co-simulator to obtain the QoS estimate (denoted as $\text{QoS}(\cdot)$) of the scheduling decision of X (S_t^X) and that of GOBI (S_t^{GOBI}). Thus,

$$\text{Impr. Ratio} = \frac{1}{T} \sum_{t=1}^T \mathbf{1}(\text{QoS}(S_t^X) > \text{QoS}(S_t^{\text{GOBI}})). \quad (7.39)$$

This denotes the ratio of the times the model X can predict a better scheduling decision than the reference. As GOBI is oblivious to future faults, this metric should always be > 0.5 . The QoS estimate is calculated using a combination of metrics like energy consumption and response time, common in edge computing deployments [Basu et al., 2019]. Thus,

$$\text{QoS}(S_t) = 1 - \alpha \cdot \text{ART}_t - \beta \cdot \text{AEC}_t. \quad (7.40)$$

Here, ART_t is the average response time of the tasks leaving the system at the end of I_t and AEC_t is the average energy consumption of the system in I_t . Both are obtained by executing S_t on a co-simulator of a scheduling interval. Also, α, β are convex combination weights that can be set as per user requirements. For our experiments, we consider $\alpha = \beta = 0.5$. Note that a higher QoS score is better.

To compare the scheduling times, we normalize them with respect to the scheduling time of the reference GOBI scheduler and call these overhead ratios. We also consider standard QoS metrics, including resource utilization, fraction of service level objective (SLO) violations, fairness and average migration time. We consider the relative definition of SLO (as in Chapter 3) where the deadline is the 90th percentile response time for the same application (Yolo/Pocketsphinx/Aeneas) on the state-of-the-art baseline TopoMAD. For fairness, we use the Jain’s fairness index. Migration time is the time spent to live-migrate the running container instances.

7.4.3 Model Training and Assumptions

7.4.3.1 PreGAN and PreGAN+

To conduct our tests, we use the COSCO framework introduced in Chapter 3. For a fair comparison, we use the GOBI scheduler as the underlying scheduling policy.

To generate the ground-truth fault labels and classes, we use the Anomaly Detection Engine for Linux Logs (ADE) tool [Open Mainframe Project]. The output anomaly classes for the tool are: CPU over-utilization (CPUO), Abnormal disk utilization (ADU), Memory leak (MEL), Abnormal memory allocation (AMA) and Network overload (NOL). CPUO occurs when the CPU utilization exceed 90%. When the disk controller of a

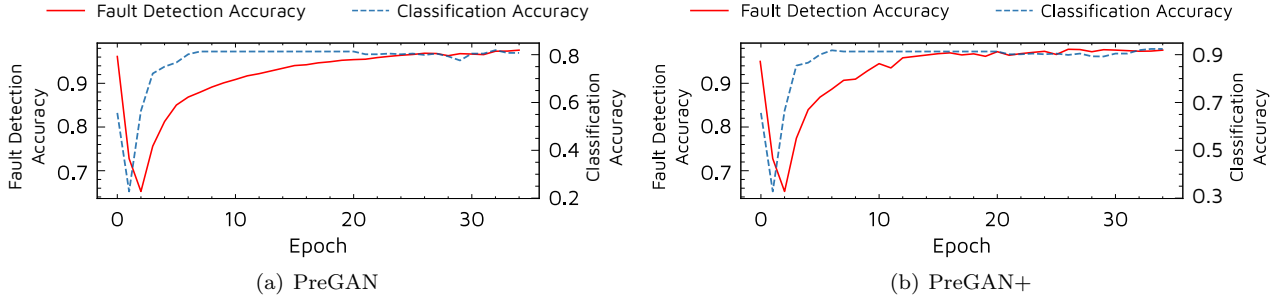


Figure 7.7: Training curves for the Fault Prototype Encoder.

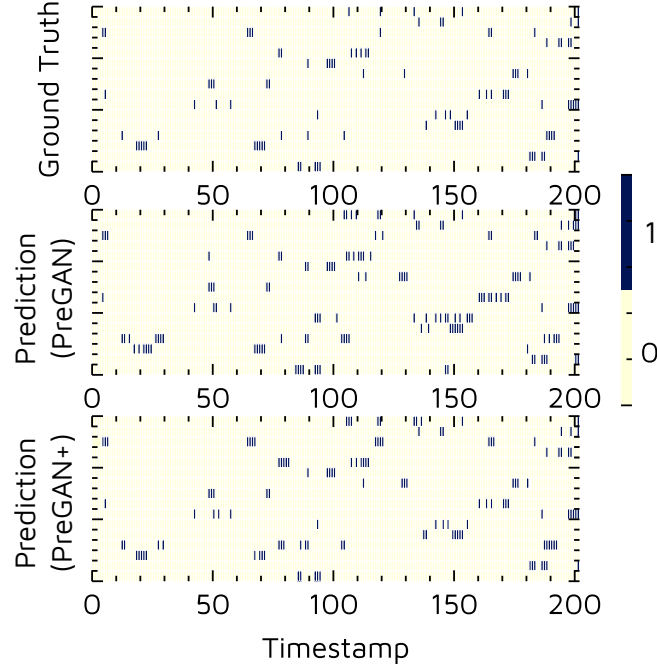


Figure 7.8: Fault Prediction using PreGAN and PreGAN+ models with Ground Truth labels.

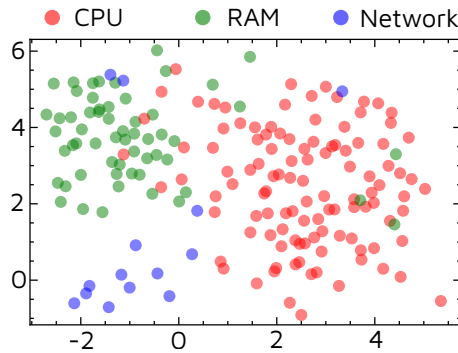


Figure 7.9: t-SNE plot of prototypes generated for the test set by the PreGAN+ model.

system throttles read/write operations, an ADU fault occurs. MEL occurs when there is incremental allocation of 1 MB memory every 3 seconds. When RAM I/O utilization exceed 90%, it causes an AMA fault. NOL fault occurs when network buffer overloads and temporary disk space needs to be allocated for file transfers. All faults events are raised when the above conditions hold for at least 60 seconds. For simplicity, the CPUO anomaly is classified as a CPU fault, NOL/ADU are clubbed together as a Network fault (as network buffer overloads almost always result in disk faults) and MEL/AMA are clubbed together as a RAM fault (due to

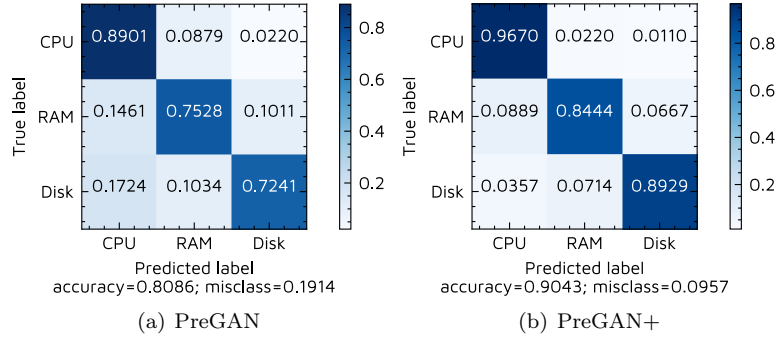


Figure 7.10: Confusion matrices for fault classification on the test set.

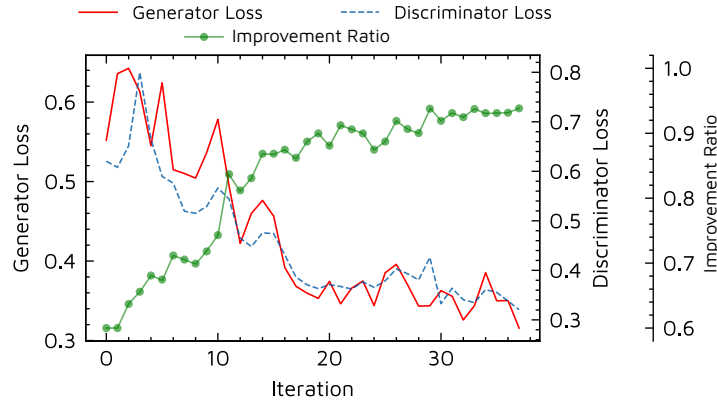


Figure 7.11: Training of the GAN network.

high correlation between these two). Thus, we consider 3 fault types/classes.

To train the FPE and GAN models, we use the AdamW optimizer [Saleh and Mashaly, 2019]. We train the FPE encoder and GAN using a traces of lengths 1000 and 1200 scheduling intervals respectively. These numbers were obtained using the early-stopping convergence criterion. The QoS score obtained from the co-simulations was a convex combination of the normalized energy consumption and SLO violations. The hyperparameter values used in our experiments for PreGAN were obtained using grid search. We use the prototype embedding size of 8, windows size of 5, initial value of α as 0.9 and decay rate ϵ as 0.05.

Figure 7.7 shows how the FPE encoder converges in offline training using 80% of the dataset collected from traces of 1000 intervals (rest as test set). As the model is trained, its fault detection and classification accuracies improve. We also observe that the Transformer model allows the FPE in PreGAN+ to give improved detection and classification accuracies due to the historic context being given as input for more informed decision making. Figure 7.8 shows the predicted and ground-truth fault labels on the test set, with hosts on the y-axis and intervals on the x-axis. We observe PreGAN+ performing better for fault diagnosis compared to PreGAN. Figure 7.9 shows a t-SNE plot of the class prototypes predicted on the test set using the FPE of PreGAN. Clearly, the prototype embeddings of the same class are clustered together, demonstrating that the model is able to distinguish among the different fault types. Classification confusion matrix on the test set can be seen in Figure 7.10. The prototype embeddings generation allows the model to generalize and accurately classify even previously unseen time-series inputs into one of the three classes. After training the FPE, we train the GAN. Figure 7.11 shows the discriminator (L_D) and generator (L_G) loss values (for PreGAN) with the Improvement Ratio for the first 40 iterations, *i.e.*, the intervals with faults in the GAN training dataset.

7.4.3.2 DeepFT

The resource metrics are collected every 10 seconds and the corresponding time-series windows are sent to the broker asynchronously at the start of each scheduling interval. The preemptive migration decisions are extracted as the difference between the input and the output schedules of the DeepFT method. Only those migrations are performed that can be accommodated by the target hosts.

To train and fine-tune the surrogate model of the proposed approach, we use the AdamW optimizer [Loshchilov and Hutter, 2018]. We apply a learning rate of 10^{-4} , number of fault classes $j = 3$ and windows size of $k = 5$. We also use a weight regularization parameter of 10^{-4} . The hyperparameters were obtained using grid-search. Figure 7.15 shows the convergence plots for the DeepFT’s surrogate model. We use the early stopping criterion to train the model. In 21 epochs, the reconstruction and triplet losses converge and we reach detection and classification accuracy of 0.9422 and 0.9592. As this classification does not correspond to any semantic fault class labels, we do not report these results for other models.

7.4.4 Baselines

7.4.4.1 PreGAN and PreGAN+

We compare the proposed methods, PreGAN and PreGAN+ with the state-of-the-art baselines: DFTM [Sivagami and Easwarakumar, 2019], ECLB [Sharif et al., 2020], PCFT [Liu et al., 2016] and CMODLB [Negi et al., 2021] (more details in Chapter 2). Other heuristic based approaches have been omitted for brevity as these baselines outperform those in all the metrics we consider in our experiments. We use hyperparameters of the baseline models as presented in their respective papers. We train all deep learning models using the PyTorch-1.8.0 [Paszke et al., 2019] library¹.

7.4.4.2 DeepFT

We compare the DeepFT method against the baselines DFTM [Sivagami and Easwarakumar, 2019], ECLB [Sharif et al., 2020], PCFT [Liu et al., 2016], AWGG [Hu et al., 2021] and TopoMAD [He et al., 2020] (more details in Chapter 2). As TopoMAD and AWGG are only fault-detection methods, we supplement them with the PSO based preemptive migration scheme for fault-recovery from the next best baseline, *i.e.*, PCFT. We train all deep learning models using the PyTorch-1.8.0 [Paszke et al., 2019] library.

7.4.5 Results for PreGAN and PreGAN+

We now describe the results corresponding to 100 interval runs on the testbed using the DeFog workloads. The ground-truth labels are obtained offline, after the complete execution to compare the detection and diagnosis performance of the models. As per prior work, we give equal weights to energy consumption and average response time in Eq. 7.23 by having $\beta = 0.5$. The trends discussed next follow also for other values of β . We also keep $\xi = 0.4$ as per the sensitivity analysis given in Section 7.4.5.2. Table 7.1 shows the detection and diagnosis metrics with the improvement ratios. It also shows the overhead ratio in terms of the time it takes to run the preemptive migration model relative to the scheduling model. The overhead ratio is highest for

¹All model training and experiments were performed on a system with configuration: Intel i7-10700K CPU, 64GB RAM, Nvidia RTX 3080 and Windows 10 Pro OS. This was also the broker node in our setup.

Table 7.1: Performance scores of PreGAN and PreGAN+ and the baseline methods with standard deviation. The best scores are shown in bold.

Method	Detection				Diagnosis		Overhead Ratio	Improvement Ratio
	Accuracy	Precision	Recall	F1 Score	HR@100	NDCG@100		
DFTM	0.8731 $\pm$ 0.0234	0.7713 $\pm$ 0.0823	0.8427 $\pm$ 0.0199	0.8054 $\pm$ 0.0872	0.5129 $\pm$ 0.0212	0.4673 $\pm$ 0.0019	0.0413 $\pm$ 0.0021	0.3783 $\pm$ 0.1001
ECLB	0.9413 $\pm$ 0.0172	0.7812 $\pm$ 0.0711	0.8918 $\pm$ 0.0203	0.8329 $\pm$ 0.0901	0.4913 $\pm$ 0.0010	0.5239 $\pm$ 0.0024	0.1028 $\pm$ 0.0009	0.5912 $\pm$ 0.0341
PCFT	0.8913 $\pm$ 0.0108	0.8029 $\pm$ 0.0692	0.9018 $\pm$ 0.0165	0.8495 $\pm$ 0.0312	0.5982 $\pm$ 0.0094	0.5671 $\pm$ 0.0020	0.0913 $\pm$ 0.0014	0.6824 $\pm$ 0.0473
CMODLB	0.9128 $\pm$ 0.0112	0.8158 $\pm$ 0.0343	0.9013 $\pm$ 0.0091	0.8605 $\pm$ 0.0284	0.6309 $\pm$ 0.0025	0.5432 $\pm$ 0.0031	0.2123 $\pm$ 0.0003	0.7283 $\pm$ 0.0065
PreGAN+ (SS)	0.9638 $\pm$ 0.0132	0.8744 $\pm$ 0.0123	0.9023 $\pm$ 0.0082	0.8881 $\pm$ 0.0193	0.6366 $\pm$ 0.0109	0.5901 $\pm$ 0.0031	0.1922 $\pm$ 0.0004	0.7622 $\pm$ 0.0058
PreGAN+ (GT)	0.9728 $\pm$ 0.0291	0.8819 $\pm$ 0.0193	0.9281 $\pm$ 0.0079	0.9044 $\pm$ 0.0192	0.6710 $\pm$ 0.0012	0.6099 $\pm$ 0.0029	0.2001 $\pm$ 0.0002	0.8092 $\pm$ 0.0102
PreGAN+ (T)	0.9789 $\pm$ 0.0099	0.8792 $\pm$ 0.0727	0.9298 $\pm$ 0.1018	0.9038 $\pm$ 0.0193	0.6715 $\pm$ 0.0061	0.6092 $\pm$ 0.0010	0.2101 $\pm$ 0.0010	0.7828 $\pm$ 0.0070
PreGAN	0.9635 $\pm$ 0.0092	0.8723 $\pm$ 0.0221	0.9018 $\pm$ 0.0121	0.8868 $\pm$ 0.0629	0.6232 $\pm$ 0.0069	0.5898 $\pm$ 0.0080	0.1617 $\pm$ 0.0017	0.7605 $\pm$ 0.0060
PreGAN+	0.9813 $\pm$ 0.0087	0.8827 $\pm$ 0.0059	0.9312 $\pm$ 0.0076	0.9063 $\pm$ 0.0384	0.6728 $\pm$ 0.0028	0.6109 $\pm$ 0.0041	0.2092 $\pm$ 0.0001	0.9014 $\pm$ 0.0026

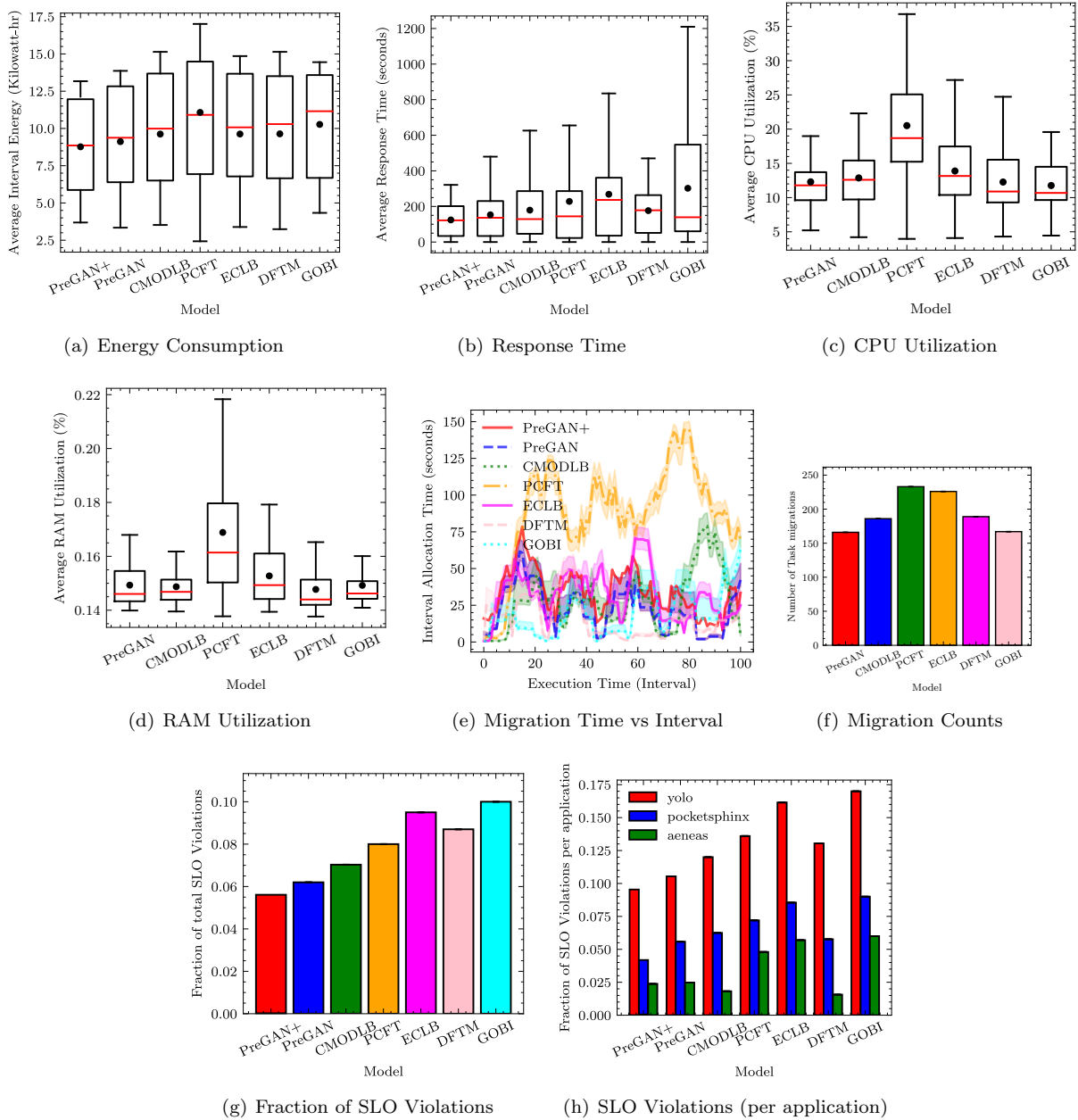


Figure 7.12: Comparison of QoS parameters of PreGAN against baselines.

CMODLB due to the several sequential steps in its model including k-means clustering, ANN inference, and PSO. DFTM has the lowest overhead, but still performs poorly as it uses a basic sinusoidal thermal trend

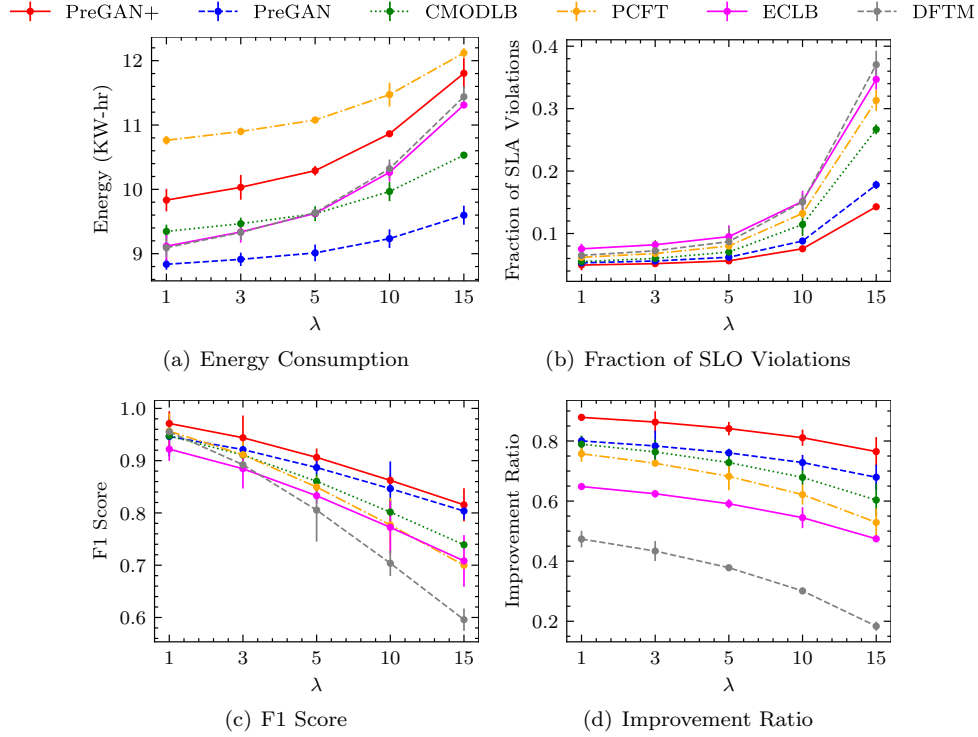


Figure 7.13: Sensitivity Analysis for all models with λ (parameter of the discrete Poisson distribution).

for all models and do not consider host heterogeneity. On average, the accuracy, precision, recall and the F1 score of the PreGAN and PreGAN+ models are higher than the baselines. This is due to the late-fusion of the embeddings obtained by the feature (GAT) and temporal trend extraction using GRU or Transformers. This allows the proposed models to not only exploit the correlations across host devices, but also detect sudden deviations from the expected resource utilization characteristics across time. Overall, scores are highest for the PreGAN+ model, thanks to the historic context given as input to the Transformer model as well as its ability to adapt in non-stationary settings by periodic fine-tuning. CMODLB uses k-means clustering to identify a common utilization threshold, irrespective of the running workloads or the host capacities. This gives higher fault detection scores for the PreGAN and PreGAN+ models. For fault-diagnosis, the PreGAN+ method has the highest HitRate (0.6728) with CMODLB being second (0.6309). The NDCG score of the PreGAN+ model is the highest being 0.6109. This is due to the factored fault prediction in the PreGAN+ model. Finally, the improvement ratio of the PreGAN+ model is the highest, *i.e.*, 0.9014 improving from 0.7283 of the CMODLB model by 23.76%. In terms of overhead, PreGAN has lower overhead compared to PreGAN+ as PreGAN+ additionally runs the fine-tuning steps and a more sophisticated Transformer based neural network.

Figure 7.12 compares the QoS scores of all models, including the vanilla GOBI approach without any fault-tolerant preemptive migrations. The PreGAN+ model has the lowest average energy consumption of 8.8721 KW-hr, with PreGAN being next at 9.0121 KW-hr. This is due to the relatively lower average CPU and RAM utilizations (Figures 7.12(c) and 7.12(d)) and the energy based QoS score when training the GAN model using co-simulations (see Eq. 7.23). Figure 7.12(b) shows the average response times with response time being defined as the time between the creation of a task from an IoT sensor and the gateway receiving the response. Among the baselines, the DFTM approach has the lowest response time as it uses minimum-migration-time heuristic [Beloglazov and Buyya, 2012] for task migrations (Fig. 7.12(e)). PreGAN and PreGAN+ avoid unnecessary migrations to prevent avoidable use of network resources, improving overall system reliability

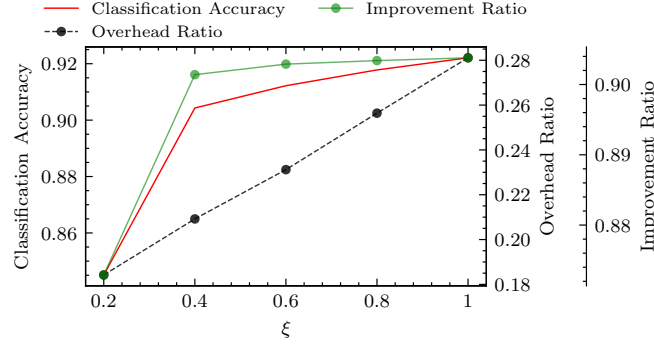


Figure 7.14: Sensitivity Analysis for PreGAN+ with ξ (fraction of labeled data).

(Fig. 7.12(f)). Even with higher response times, CMODLB achieves lowest SLO violation rates (0.0717) among the baselines as its migration decisions are SLO aware. PreGAN and PreGAN+ have SLO violation rates of 0.0621 and 0.0568, respectively, which are 13.4% and 20.8% lower than CMODLB. Figure 7.12(h) shows the SLO violation rates for each application. For all models, Yolo has the highest violation rate due to its compute heavy utilization characteristics. The migration decisions in PreGAN and PreGAN+ use the fault class labels to decide the target hosts in the migration decisions. The fine-grained classification in the proposed models allows them to migrate CPU intensive tasks from a compute constrained node to a node with low CPU utilization, even if the target node is running at capacity on RAM and Disk. Compared to the binary classification in CMODLB, this gives PreGAN and PreGAN+ more choices for the target host, allowing lower overheads in migration and more balanced resource utilization.

7.4.5.1 Ablation Analysis

To study the relative importance of each component of the PreGAN+ model, we exclude every major one and observe how it affects performance. First, we consider the PreGAN+ model without the semi-supervised fine-tuning of the FPE encoder and denote this as *PreGAN+ (SS)* model. We also disable the fine-tuning of the GAN model and denote this as *PreGAN+ (GT)* model. Finally, we replace the Transformer based encoder with GRU and refer to this as *PreGAN+ (T)* model. Results are presented in Table 7.1. We observe that without tuning the FPE encoder or GAN the improvement ratio drops significantly, which means that periodic fine-tuning of the neural network allows the model to adapt to dynamic scenarios and adjust to changes in the environment. Further, replacing the Transformer based encoder with a GRU, as in PreGAN, makes the training very slow as GRU requires iterative training whereas a Transformer can be given the whole sequence (C in Fig. 7.2) at once. Thus, periodic fine-tuning supports the adaptability of PreGAN+ and Transformer based FPE encoder makes tuning at each interval time-efficient.

7.4.5.2 Sensitivity Analysis

Sensitivity with λ . Figure 7.13 shows the variation of the energy consumption, SLO violations, F1 score and improvement ratio with the λ parameter in our Poisson distribution used to model the workloads. We vary λ from 1 to 15 ($\lambda = 15$ constantly gives $> 90\%$ CPU utilization for all hosts). Under a higher λ more tasks are produced, making the fault prediction harder. This is apparent from the drop in the F1 scores, leading to higher SLO violations. Even the energy consumption increases due to the increase in the average CPU utilization of

Table 7.2: Comparison of the load on broker node.

Model	Overhead Ratio	CPU (%)	RAM (GB)
PreGAN	0.1617	28%	4.3 GB
PreGAN+	0.2902	56%	16.2 GB

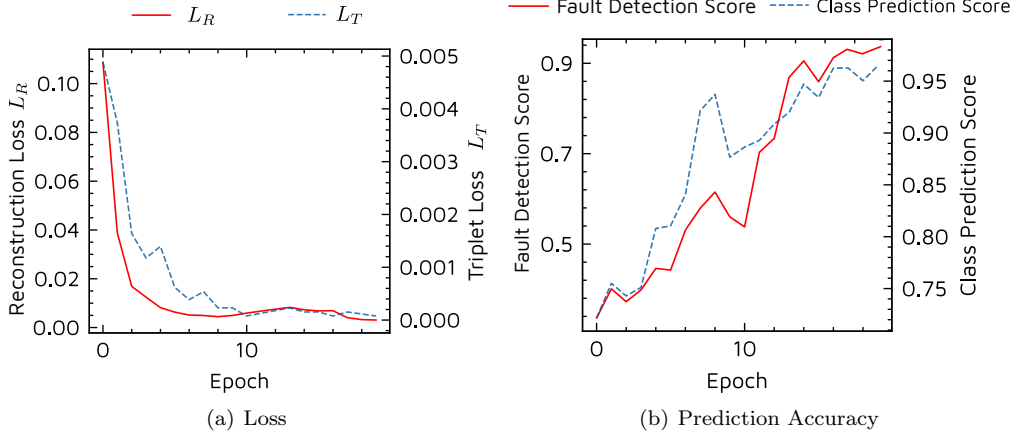


Figure 7.15: Convergence plots for the DeepFT model.

the system. Overall, PreGAN shows the least relative drop in F1 scores and improvement ratio as we increase λ giving the least SLO violations even in workload heavy executions.

Sensitivity with ξ . Figure 7.14 demonstrates the variation of the F1 score, overhead ratio and improvement ratio with the fraction of data that is labeled by ADE when periodically fine-tuning the FPE encoder of PreGAN+. Naturally, as we increase ξ , the amount of data to label increases, which consequently increases the overhead ratio. However, with more labeled data, the model is able to classify better and hence the classification score improves as ξ increases. Thus, there is a tradeoff between the decision making time and the performance of the model. As the classification accuracy saturates for $\xi \geq 0.4$, labelling more data does not improve significantly improve performance, but linearly increases the overhead. This is shown by the improvement ratio, which shows negligible improvement for $\xi > 0.4$. Thus, we use $\xi = 0.4$ in our experiments.

7.4.5.3 Broker Load Comparison

Table 7.2 compares the CPU and RAM utilizations as well as the overhead ratios of the PreGAN and PreGAN+ models. Even though PreGAN+ gives better QoS scores, it imposes high overhead on the broker node, which makes it unsuitable for settings with resource constrained broker. In such settings (broker with RAM up to 5 GB), we could use PreGAN for fault-tolerant computing that too outperforms baselines.

7.4.6 Results for DeepFT

Visualization of the DeepFT method. Figure 7.16 shows how the DeepFT method works starting from a sample scheduling decision. The plot on the left shows the optimization loss L_O with iteration count for the Adam optimizer. The blips in the optimization curve are due to the warm restarts in cosine annealing. In 20 iterations, the optimization loss drops from 3.03 to 2.43. On the right is a t-SNE plot of the NAP (no-fault) class and the three fault classes. The scheduling decision belongs to the c_2 fault class in the beginning, but

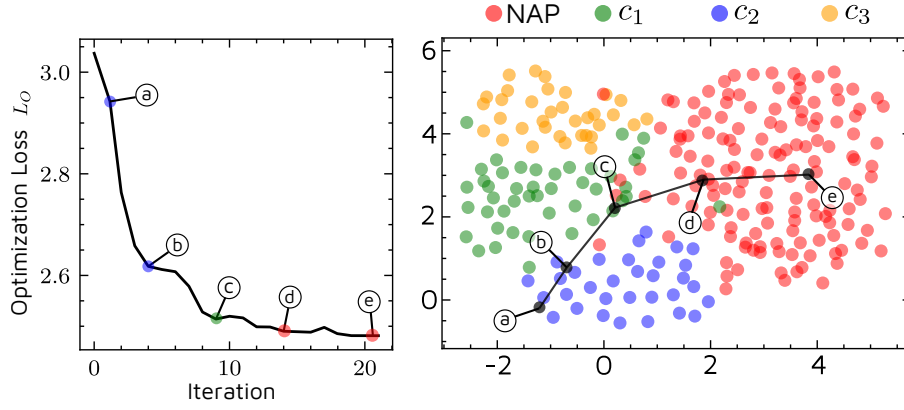


Figure 7.16: Visualization of the optimization loop for a sample scheduling decision.

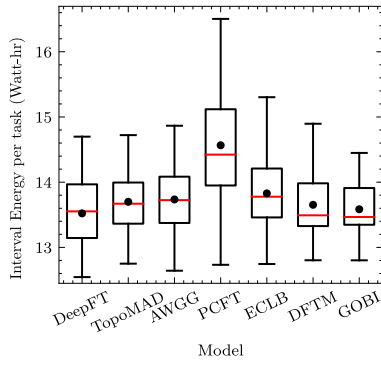
Table 7.3: Performance scores of DeepFT and the baseline methods with standard deviation. The best scores are shown in bold.

Method	Detection				Diagnosis		Overhead Ratio	Improvement Ratio
	Accuracy	Precision	Recall	F1 Score	HR@100	NDCG@100		
DFTM [332]	0.8731 $\pm$ 0.0234	0.7713 $\pm$ 0.0823	0.8427 $\pm$ 0.0199	0.8054 $\pm$ 0.0872	0.5129 $\pm$ 0.0212	0.4673 $\pm$ 0.0019	1.0413 $\pm$ 0.0021	0.3783 $\pm$ 0.1001
ECLB [322]	0.9213 $\pm$ 0.0172	0.7812 $\pm$ 0.0711	0.8918 $\pm$ 0.0203	0.8329 $\pm$ 0.0901	0.4913 $\pm$ 0.0010	0.5239 $\pm$ 0.0024	1.1028 $\pm$ 0.0009	0.5912 $\pm$ 0.0341
PCFT [220]	0.8913 $\pm$ 0.0108	0.8029 $\pm$ 0.0692	0.9018 $\pm$ 0.0165	0.8495 $\pm$ 0.0312	0.5982 $\pm$ 0.0094	0.5671 $\pm$ 0.0020	1.0913 $\pm$ 0.0014	0.6824 $\pm$ 0.0473
AWGG [144]	0.9194 $\pm$ 0.0081	0.8237 $\pm$ 0.0124	0.9012 $\pm$ 0.0081	0.8607 $\pm$ 0.0135	0.6284 $\pm$ 0.0010	0.5564 $\pm$ 0.0007	1.2130 $\pm$ 0.0001	0.7209 $\pm$ 0.0027
TopoMAD [134]	0.9229 $\pm$ 0.0028	0.8562 $\pm$ 0.0038	0.8927 $\pm$ 0.0015	0.8741 $\pm$ 0.0101	0.6098 $\pm$ 0.0023	0.5330 $\pm$ 0.0030	1.2892 $\pm$ 0.0007	0.7313 $\pm$ 0.0016
DeepFT	0.9422 $\pm$ 0.00783	0.8635 $\pm$ 0.0011	0.9001 $\pm$ 0.0092	0.8814 $\pm$ 0.0271	0.6193 $\pm$ 0.0012	0.5682 $\pm$ 0.0072	1.2092 $\pm$ 0.0009	0.7556 $\pm$ 0.0047

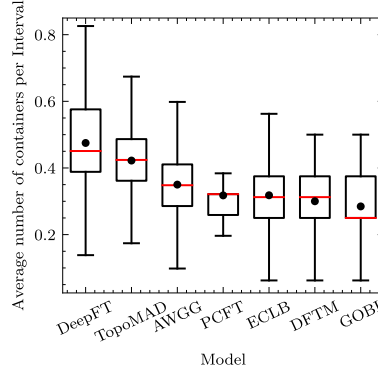
moves towards the NAP class with iterations. Finally, the converged scheduling decision belongs to the no-fault class shown in red.

Comparison with baselines. Table 7.3 shows the detection and diagnosis metrics with the improvement and overhead ratios. Figure 7.17 show the QoS metrics of all models for $Poisson(\lambda = 5)$. We also present QoS results for the GOBI scheduler. The fault-detection accuracy of the DeepFT model is the highest (0.9422), improving the best among baseline (0.9229) of the TopoMAD model. Similarly, the DeepFT model also outperforms the baselines in terms of F1 score. The diagnosis scores of the proposed method are close to the state-of-the-art values. The scores are close to the *supervised* PreGAN method without the need for any supervised labels. This demonstrates the efficacy of the DeepFT approach and its capacity to generalize to settings where supervised methods, such as PreGAN, would not be able to perform well due to the lack of ground-truth data. The improvement ratios of all models except DFTM are higher than 0.5. This shows that the DFTM model performs slightly worse than the GOBI scheduler in terms of the QoS metrics of response time and energy consumption. However, the SLO violation rates of the DFTM model are lower than GOBI (Fig. 7.17(g)). This is partly because the DFTM approach performs migrations very aggressively (Fig. 7.17(j)).

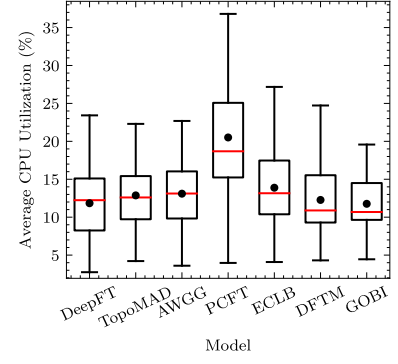
In terms of the improvement ratio, TopoMAD has the highest score of 0.7313 among the baselines. The DeepFT gives a 3.32% higher score of 0.7556. Even with the periodic fine-tuning of the surrogate model, the overhead ratio of the DeepFT model is 6.20% lower than the best baseline, TopoMAD. This is due to the relatively time consuming PSO optimization strategy in baselines like TopoMAD and AWGG. DeepFT, on the other hand, uses a directed gradient-based update strategy to optimize the scheduling decision, which has been shown to converge quickly compared to gradient-free methods (see Chapter 3). In terms of the diagnosis results, the AWGG method has the highest HitRate (0.6309), with the DeepFT being very close (0.6193). The NDCG score of the DeepFT model is the highest (0.5682). This is due to the factored fault prediction in the DeepFT model.



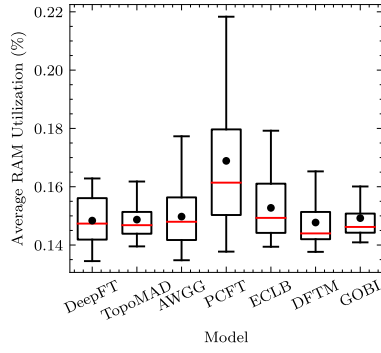
(a) Energy Consumption per task



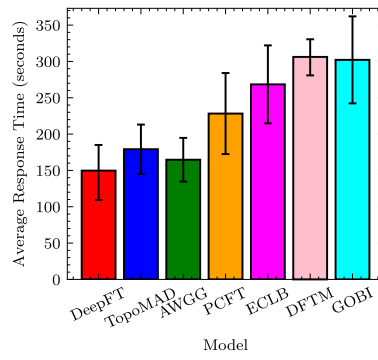
(b) Number of active containers



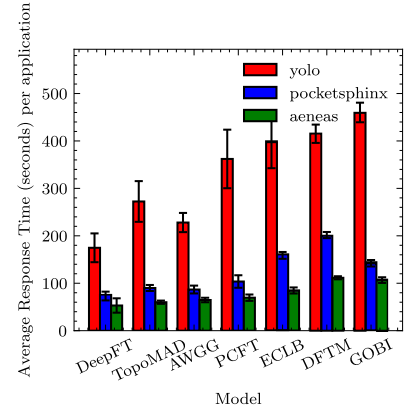
(c) CPU Utilization



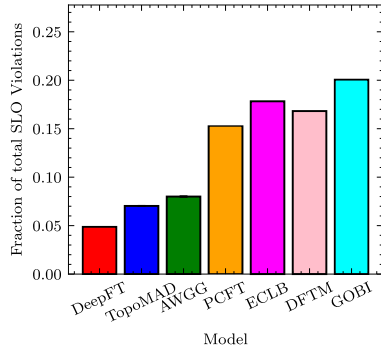
(d) RAM Utilization



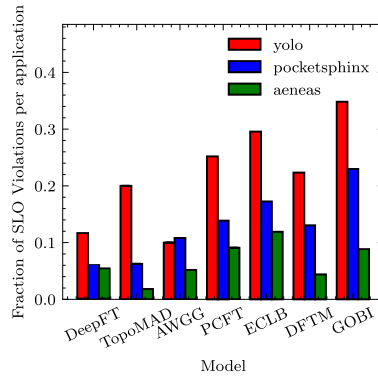
(e) Average Response Time



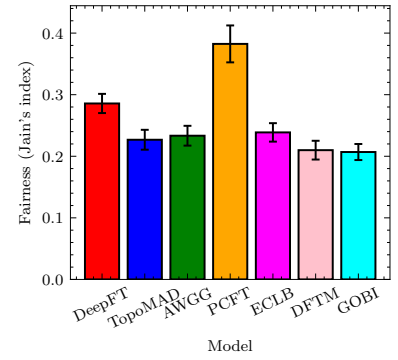
(f) Average Response Time (per application)



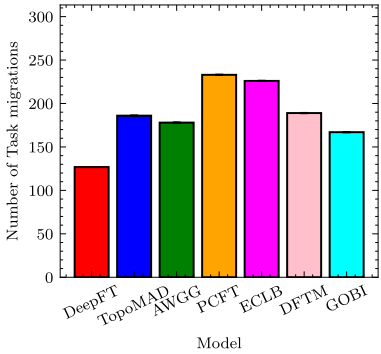
(g) Fraction of SLO Violations



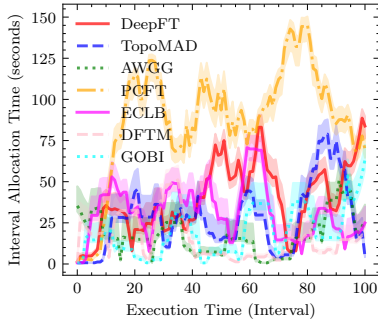
(h) SLO Violations (per application)



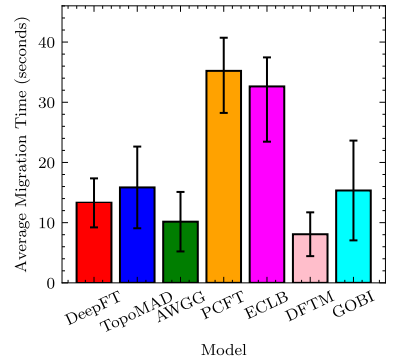
(i) Fairness



(j) Migration Count



(k) Migration Time vs Interval



(l) Migration Time

Figure 7.17: Comparison of QoS parameters of DeepFT against baselines.

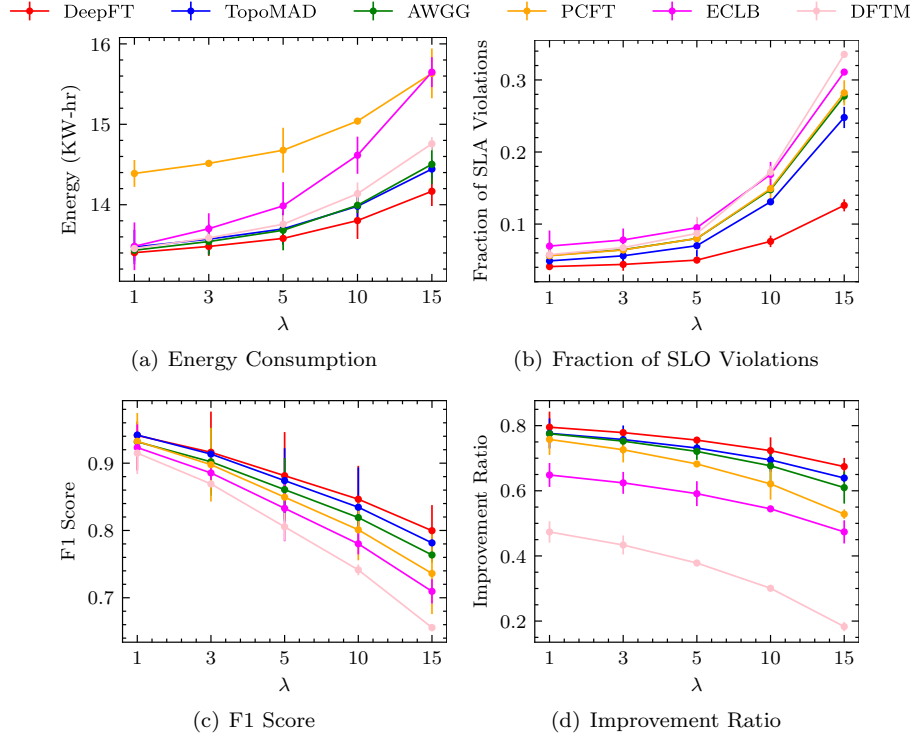


Figure 7.18: Sensitivity Analysis for all models with λ (parameter of the Poisson distribution).

In terms of QoS scores, DeepFT gives the lowest energy consumption per completed task 13.55 KW-hr, with TopoMAD being next with 13.74 KW-hr. This is due to the relatively low average CPU and RAM utilization in DeepFT (Figs. 7.17(c)-7.17(d)) and the high number of active containers (Fig. 7.17(b)). Moreover, DeepFT gives the lowest average response time of 149.2 seconds, with AWGG giving the next best of 164.7 seconds (Fig. 7.17(e)). DeepFT also gives a significant reduction of 37.21% on the SLO violation rates (Fig. 7.17(g)). This is because DeepFT avoids unnecessary migrations to prevent avoidable use of network resources, improving overall system reliability (Fig. 7.17(j)) leading to low overall migration times (Fig. 7.17(l)). Figures 7.17(f) and 7.17(h) show the average response time and SLO violations for each application. The Yolo application has the highest response time and SLO violation rates due to its computationally heavy requirements. The PCFT method gives the highest fairness index of 0.39, with DeepFT coming second with 0.28 as shown in Fig. 7.17(i). **Summary.** The results demonstrate that all method for fault-tolerant computing lead to performance improvement, in terms of critical metrics such as SLO violation rates and response times, compared to the case without any fault-tolerance (GOBI). Across all methods, the DeepFT approach gives the best QoS scores, thanks to its high fault prediction accuracy and low overheads.

7.4.6.1 Sensitivity Analysis

Figure 7.18 shows the variation of the energy consumption, SLO violations, F1 score and improvement ratio with the λ parameter in our Poisson distribution used to model the workloads. We vary λ from 1 to 15 ($\lambda = 15$ constantly gives $> 90\%$ CPU utilization for all hosts). Under a higher λ more tasks are produced, making the fault prediction harder. This is apparent from the drop in the F1 scores, leading to higher SLO violations. Even the energy consumption increases due to the increase in the average CPU utilization of the system. Overall, DeepFT shows the lowest relative drop in F1 scores and improvement ratio as we increase λ giving the lowest SLO violations even in workload heavy executions.

7.5 Summary

We have presented two preemptive migration prediction models (PreGAN and PreGAN+) that can detect, diagnose and classify faults in edge computing environments. PreGAN uses GAT and GRU for feature extraction with a Multi-Head-Attention and Prototype prediction decoder to detect and classify faults. It also leverages a generator model to utilize the anomaly class prototypes to output a delta scheduling decision (migrations) to rectify the faults and improve QoS. The discriminator model with co-simulations allow us to decide between the original and modified scheduling decisions. Moreover, the discriminator aids generator training using adversarial loss and does not require it to run co-simulations at test time. PreGAN is a static model which we extend to PreGAN+ that uses a Transformer based encoder and semi-supervised model fine-tuning to dynamically adapt in non-stationary mobile edge environments. Experiments demonstrate that PreGAN and PreGAN+ outperform baselines in terms of energy consumption, response times and SLO violations respectively. They also able to correctly identify faults, giving an average F1 score of 0.9063, higher than the state-of-the-art models. The current model assumes a master-slave design. As part of future work, we plan to explore extensions to enable deploying PreGAN in serverless platforms with streaming tasks [Casale et al., 2020].

We have also presented a deep surrogate model-based fault-tolerance approach (DeepFT) for reliable edge computing. DeepFT can detect and diagnose faulty system conditions without the need for extensive labeled data. Instead, DeepFT utilizes a deep surrogate model that analyzes the scheduling decision and state information to predict a reconstruction of the next state window and fault prototype. Using reconstruction and few-shot based triplet loss, the surrogate model is trained offline to predict fault scores based on the past data trends. The model represents a self-supervised training model, where it applies co-simulations to generate true state windows. Generating its own ground-truth labels (by utilizing a co-simulator) allows the model to be fine-tuned on-the-fly to adapt it in volatile settings. These advances allow DeepFT to have high fault detection and diagnosis scores that facilitate efficient fault-aware scheduling for optimal QoS. Specifically, DeepFT achieves an improvement of 2% and 3% for fault-detection accuracy and improvement ratio. DeepFT achieves this with 6.2% lower overhead than the best baseline TopoMAD. DeepFT is also able to support 1.38%, 9.41% and 37.21% lower energy consumption, response times and SLO violations, respectively compared to the state-of-the-art models. As a future work, we plan to explore whether the DeepFT approach can be used for efficient resource and application management of SLO-based serverless (FaaS) computing at the edge.

Both models allow us to create a fault-tolerance fog environment, achieving objective 3(a) as described in Chapter 1. We build on the ideas of co-simulation developed in Chapter 3 and integrate with generative and classification models to build state-of-the-art fault-tolerance models.

This page intentionally left blank.

Chapter 8

AI Augmented Service Resilience

This chapter describes novel resilience approaches, namely DRAGON and CAROL and their extensions, to achieve network resilience (objective 3(b)) as described in Chapter 1. In Chapters 4-7, we discussed how to optimally provision resources, deploy and place tasks in a fault-tolerant fog environment. However, the probability of node failure is still non-zero and we need to ensure the broker-worker setup is resilient to such failures, especially in the case of the common federated architectures with several broker-worker clusters. This chapter develops service resilience models, extending the ideas of generative modelling, to ensure the broker-worker architecture does not break in case of node failures.

Background. Federated computing is a paradigm that brings together the software, infrastructure and platform services from disparate and distributed computing environments. This prevents resource under-utilization and facilitates the accommodation of sudden spikes in demand [Rochwerger et al., 2009]. In the past few years, this paradigm has been extended to cloud computing, giving rise to cloud federations [Gouasmi et al., 2018]. However, bringing the computation closer the users, *i.e.*, the edge of the network, has many benefits including lower latency, efficient communication and reduced costs [Gill et al., 2019]. Bolstered by affordable computation, recently proposed federated edge architectures enable large-scale deployment by leveraging multiple standalone edge computing providers [Cao et al., 2020]. The cost benefits of this seamless interconnect of several edge devices in a federated setup have pushed the industry from relying on cloud-only and hybrid edge-cloud deployments to edge-only ones [Zeydan et al., 2016]. Due to the large number of devices at the edge, this has led to an explosion in the amount of operational log data being generated. As modern applications shift to resource-intensive AI based workloads, mission critical tasks make it crucial to detect and recover from faults arising from malicious attacks, intrusion events or unexpected device breakdowns by mining this log data [Sharma et al., 2019].

Challenges. The key challenge addressed in this chapter is dealing with the diverse effects of node failures, such as increased task response time and violations of Service Level Objectives (SLOs), requiring different remediation steps to maintain system performance and reduce service downtime. We consider resilience in both worker and broker layers of fog environments. This is motivated by the excessive load that AI-based IoT applications put on the limited computational capacity and working memory (up to 8GB) of edge devices [Hao et al., 2018]. Higher computational load translates to increased task processing times and SLO violation rates. Excessive memory load is typically dealt with using storage mapped virtual memory, which in most settings is a network-attached disk [Shao and Zhang, 2020]. This also leads to higher response times due to time-consuming data transfers over

congested backhaul edge networks [Bilal et al., 2019]. All these issues lead to persistent resource contention and faulty behavior that adversely affects system QoS. Furthermore, without resource intensive cloud backends to rely on in edge federations, it becomes challenging to deploy modern deep learning based management solutions in resource constrained edge devices. This makes it crucial to have resilience in the worker layer of a fog environment where the task computation takes place.

Node Failures. However, typical broker-worker federations, if a worker node fails, a broker could act as a worker and complete the task or allocate the same task to another worker [Sajjad et al., 2016]. If a broker fails, all active tasks within the LEI and all incoming tasks that are sent to the broker are impacted. This makes broker resilience crucial in large-scale deployments. However, the problem of broker resilience is hard as maintaining hot-redundancy by replicating running task instances makes edge nodes more susceptible to failures [Li and Lan, 2020]. The difficulty primarily arises from the critical need for result delivery with low latency and high accuracy [Ahuja et al., 2012]. A vital parameter that controls this is the number of brokers in the system. For a fixed number of devices in the system, a high broker count translates into having fewer workers, reducing the average throughput of the system. Low broker count can cause bottlenecks and contentions, increasing fault frequency. So we need to consider both the increase or decrease of the number of brokers in the system. Furthermore, the statistical moments and correlations of the workload characteristics are non-stationary and vary over time, requiring continuous steps to adapt management models. This imposes additional overheads and impacts QoS [Ristov et al., 2020].

Existing Solutions. Several fault-tolerance approaches have been proposed for both cloud and edge systems that use heuristics or unsupervised learning strategies [Costa et al., 2016, Rawat et al., 2021, Ray et al., 2020, Gan et al., 2020, He et al., 2020]. Compared to supervised learning methods that rely on labeled data, unsupervised methods enable fault detection without human annotations or time-consuming log-trace analyses. In this domain, the best performing solutions are contemporary models that leverage memory-intensive deep generative models such as autoencoders (AE), recurrent neural networks and generative adversarial networks (GANs) [Khanna et al., 2020, Li et al., 2019, He et al., 2020, Girish and Rao, 2021, Chouliaras and Sotiriadis, 2021, Gan et al., 2020, Feng et al., 2021]. State-of-the-art fault-tolerance approaches in the cloud use deep learning due to the ability of such models to capture patterns and trends in large amounts of data. On the other hand, the approaches designed for edge systems are dominated by either heuristic based methods [Grover and Garimella, 2018, Mudassar et al., 2020] or techniques that use powerful cloud devices as management nodes [Javed et al., 2018, Wang and Mu, 2021]. This is particularly due to the memory limitations at the edge. The models that have the best detection scores (AEs and GANs) often have a very high memory footprint, 4-6 GB for training and inference in deep learning based methods. Due to the large amount of time-series data generated by edge systems [Audibert et al., 2020], the AEs and GANs used in such methods have extremely high parameter size. Some prior work aims at reducing the on-device parameter size of such models using techniques like model compression [Polino et al., 2018], neural network slimming [Wang et al., 2020] or model distribution [Li et al., 2021]. However, such approaches usually have accuracy penalties associated with them. This creates a research gap of achieving fault-detection performance comparable with that of state-of-the-art deep generative models, within memory footprint constraints.

Developing a Worker Resilience Model. To develop a memory-efficient generative model, we propose a novel model called generative optimization networks (GON). We take motivation from GANs that utilize two

neural networks, generator and discriminator, where the former generates new data and the latter acts as a critic to train the former [Goodfellow et al., 2014]. The key insight of our work is that a sufficiently trained discriminator should not only indicate whether an input belongs to a data distribution, but also how to modify the input to make it resemble the target distribution more closely. This can be done using the gradient of the discriminator output with respect to its input (as introduced in Chapter 3) and executing gradient optimization of the output score. Having only a single network should give significant memory footprint gains, at the cost of higher training times. This is acceptable in many applications scenarios where initial model training is a one-time activity [Goli et al., 2020, Huang et al., 2020]. The GON model is agnostic to the specific use case and can be utilized in any application where GANs are used. We present Decentralized Fault-Tolerance using Generative Optimization Networks (DRAGON) that extends GONs using graph neural networks and attention operations with also utilizing a simulator to estimate future system states. The decentralized fashion of fault-detection and remediation in DRAGON has a two-fold benefit. Firstly, there is no single point of failure in the system as the detection and remediation steps are run in each edge provider’s local edge infrastructure (LEI). Secondly, it allows the distribution of fault-tolerance related management workload across multiple computing devices, facilitating scalability of the model. Specifically, each LEI runs GON in its broker to detect faults within its edge devices. DRAGON extends the GON model to **also consider the system topology** and leverages the updated model in a decentralized fashion. It additionally uses a simulator, akin to a cyber-cyber digital-twin, to converge to fault-remediation decisions. All brokers altruistically transfer and resume tasks from faulty nodes to a different device for load-balancing, commonly referred to as *preemptive migration* [Engelmann et al., 2009]. The GON model is trained using log traces on DeFog benchmarks [McChesney et al., 2019] and executed on AIoT benchmarks [Luo et al., 2018] to test efficacy in unseen settings. DRAGON is the first system that uses a GON in edge brokers to predict faults in edge devices of each LEI and proactively runs load-balancing preemptive migration steps to remediate faults. Extensive experiments on a Raspberry-Pi based federated edge cluster show that DRAGON performs best in terms of fault-detection and QoS metrics. To test the generalization of the model, we test on three different federated configurations. Specifically, DRAGON reduces energy consumption, response time and service deadline violations by up to 74%, 63% and 82% compared to state-of-the-art baselines.

Developing a Broker Resilience Model. In the proposed work, an offline trained GON model with labelled data generated from a system with similar behavior as that of the training dataset would typically give high confidence scores, whereas as the system behavior changes, the confidence score declines. Dynamic thresholding techniques allow us to decide confidence thresholds below which we fine-tune the network parameters. Thus, we only run fine-tuning measures when required, significantly reducing overheads and improving system QoS. In this chapter we present CAROL: Confidence Aware Resilience Model for edge federations. CAROL is the first system that uses a GON in edge brokers to reactively run topology optimization to optimize system QoS. The GON model is trained using log traces on DeFog benchmarks [McChesney et al., 2019]. Extensive experiments on a Raspberry-Pi based federated edge cluster show that CAROL performs best in terms of QoS metrics. To test the generalization of the model, we test on different benchmarks as workloads, namely AIoT [Luo et al., 2018]. Specifically, CAROL reduces energy consumption, SLO violation rates and resilience overheads by up to 16, 17 and 36 percent, respectively, compared to state-of-the-art baselines.

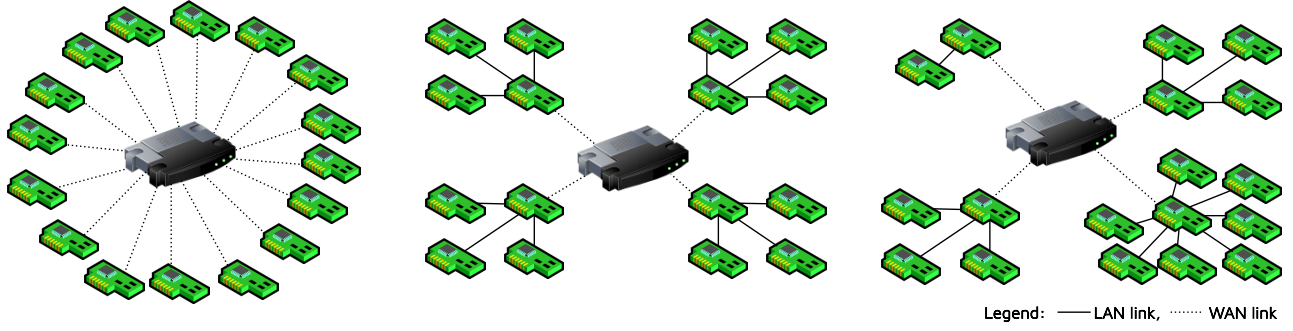


Figure 8.1: Edge Federation Model with 16 edge nodes.

8.1 System Model and Problem Formulation

8.1.1 System Model

As is commonplace in federated edge computing environments, we assume a system with heterogeneous nodes configured in a broker-worker fashion [Weissman and Grimshaw, 1996]. The assignment of edge nodes as brokers or workers and the allocation of all workers to one of a broker defines the topology of the system. We denote the number of edge nodes by H . As is common in edge federations, we assume that the broker nodes of LEIs are interconnected and we allow data sharing among brokers to facilitate transferring management tasks of workers across LEIs. We also assume a bounded timeline of execution of tasks within the federated environment, which we divide into fixed-sized scheduling intervals, where I_t denotes the t -th interval (t ranging from 0 to T). We also consider that the edge broker can sample the resource utilization metrics of all hosts within its LEI group at any time including CPU, RAM, disk/network bandwidth, some additional fault-related metrics including consumption of the swap space, disk buffers, network buffers, disk and network I/O waits. Although our methods are generalizable to any federated setup with H hosts, in the rest of the discussion we consider three federated configurations with 16 edge nodes (see Figure 8.1):

- *Configuration 1:* 16 LEI groups, each with a single edge device duplicates as a broker and edge worker. All broker nodes are connected to a Wide Area Network (WAN).
- *Configuration 2:* 4 LEI groups, each with its own broker node and 3 worker nodes. All nodes in an LEI are connected with each other in a Local Area Network (LAN).
- *Configuration 3:* 4 LEI groups with 2, 4, 4 and 8 nodes. Each LEI has its own broker node.

These configurations have been chosen to cover a wide range of heterogeneous federated setups. Configuration 1 and 2 are balanced, whereas configurations 3 is unbalanced in terms of number of nodes in an LEI.

Fault Model. As per prior work [Ye et al., 2018, Sivagami and Easwarakumar, 2019], we consider a resource-contention failure model for the edge nodes in our setup [Driscoll et al., 2003]. We assume that all failures are recoverable, *i.e.*, the machines can be rebooted to resume execution from their last working state, by frequently updating snapshots of the active hosts in the system. As in prior work, edge hosts in the same LEI are connected to the same power supply and unrecoverable faults like outages are ignored [Sivagami and Easwarakumar, 2019]. Failures can occur for a variety of reasons, including software or hardware defects, such as an infinite loop or long-running uninterruptible computation, resource exhaustion (thrashing), under-performing hardware (throttling),

external events such as a slow computer network, misconfiguration, and compatibility problems. We realize this using an existing fault injection module [Ye et al., 2018] to create different fault types like CPU overload, RAM contention, Disk attack and DDOS attack. We specifically restrict our attention to faults that manifest in the form of resource over-utilization; for instance, a DDOS attack could lead to contention at the network interface. A broker or worker node may become unresponsive due to this resource over-utilization [Vasilakos et al., 2020]. This work aims to find the best system topology in terms of estimated QoS in case one or more broker nodes fail. This is crucial as broker failures lead to service downtimes from all nodes in the LEI. In case of worker failures, we simply rerun tasks on the worker with the least resource utilization in the LEI. As this work takes a step in the complementary direction of traditional load-balancing and autoscaling methods, existing fault-tolerance techniques may be leveraged for worker management [Sharif et al., 2020, Wang et al., 2020].

Workload Model. All the tasks in our system are generated by users and transferred to the edge federation using gateway devices. We assume that gateway devices send tasks to the closest broker in terms of network latency, breaking ties uniformly at random. We assume a bag-of-tasks workload model, where a set of independent tasks enter each LEI at the start of each scheduling interval. Each task has an associated (soft) SLO deadline.

8.1.2 Problem Formulation

8.1.2.1 Worker Resilience

We assume that the edge brokers run scheduling algorithms that place each incoming task onto one of the node in the LEI. Apart from this, the brokers also decide tasks that need to be migrated to another LEI as part of fault management decisions. Formally, at the start of the interval I_t , the edge broker makes a decision denoted S_t , that is a mapping of new tasks in the interval I_t as well as the active tasks from interval I_{t-1} to its edge hosts. The edge broker also makes a decision M_t , that migrates active tasks to another LEI. All tasks being migrated to an LEI are considered as incoming tasks in that interval. Unlike new tasks, migratory tasks may resume from a saved state. We also encode the undirected topology graph of the federated edge environment as G_t .

At the beginning of interval I_t , for an input time-series of LEI system states $\{x_0, \dots, x_t\}$, each broker needs to predict the next system state, *i.e.*, x_{t+1} . Here, the system state (x_t) consists of values for an arbitrary set of resource utilization metrics for all active tasks and hosts at the start of I_t . Instead of directly using the system states, we use a sliding window of length k to capture the temporal contextual trends:

$$W_t = \{x_{t-k+1}, \dots, x_t\}. \quad (8.1)$$

We use replication padding [Liu et al., 2018] for the first k intervals, where we replicate the first system state to ensure the time-series window is always of the same size. Also, to ensure robust predictions in the DRAGON model, we normalize these time-series windows using min-max scaling.

Using the input window W_t , graph topology G_t and an input scheduling decision S_t , the model needs to predict whether there is likely to be a fault in the next system state, *i.e.*, x_{t+1} . For S_t we assume that there is an underlying scheduler in the system independent from the proposed fault-tolerance solution. In lieu of directly predicting the fault label (denoted as y_t), we predict a fault score f_t using a predicted reconstruction of the

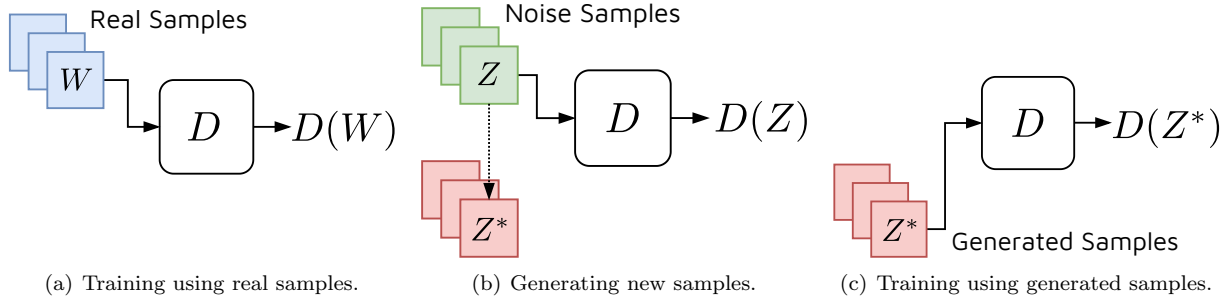


Figure 8.2: The GON framework.

next window W_{t+1} , denoted as $\hat{W}_{t+1}$. The fault score f_t is obtained by calculating the deviation between the true window W_{t+1} and its reconstruction $\hat{W}_{t+1}$. Without loss in generality, whenever unambiguous, we drop the subscripts for the sake of simplicity. Hence, we only refer to the inputs and outputs as G , W , S , $\hat{W}$, and f .

8.1.2.2 Broker Resilience

We assume that the edge brokers run software solutions to manage their worker nodes. This includes making scheduling decisions for all incoming and active tasks in the system. Such solutions may include various fault tolerance and resilience models. We denote the set of brokers by B and workers by W . In this work, in each scheduling interval, we check which broker or worker nodes are unresponsive (inactive) and update the system topology to optimize QoS. We encode the undirected topology graph of the federated edge environment at the start of the interval I_t as G_{t-1} . Formally, at the start of the interval I_t , all active nodes in the system are utilized to generate the topology G_t to execute tasks in I_t . The new graph G_t may be the same as G_{t-1} in case the active node set is unchanged.

To generate the graph G_t , we utilize the performance metrics of the previous interval, denoted by M_{t-1} , and the scheduling decision by S_t . The performance metrics include resource utilization and QoS metrics such as energy consumption and response time. For S_t we assume an underlying scheduler in the system independent from the proposed fault-tolerance solution. Using the input graph topology G_t , performance metrics M_{t-1} and an input scheduling decision S_t , the model needs to predict the performance metrics for the current interval, *i.e.*, M_t and a confidence score C_t . Using this model, we optimize over the topology space to find the optimal G_t for interval I_t . Without loss in generality, whenever unambiguous, we drop the subscripts for the sake of simplicity. Hence, we only refer to the inputs and outputs as G , M , S and C .

8.2 Generative Optimization Networks

We now describe the GON framework for generic data generation and later elucidate its function in fault-detection. In GON, we have a single discriminator model $D(\cdot; \theta)$ that is a differentiable multilayer perceptron with parameters θ . For any input set of G , W and S , the output $D(G, W, S; \theta)$ is a scalar. We denote a generated distribution with p_g , and with $D(G, W, S; \theta)$ the probability that W belongs to the data p_{data} instead of p_g . We train D such that it correctly labels samples from data and p_g . A working representation is shown in Fig. 8.2. The key idea is divided into three parts.

Training using real samples. For any sample W from the data, to make sure the output of D is maximized,

we train it to minimize the cross-entropy loss by descending the stochastic gradient

$$-\nabla_{\theta} \log(D(G, W, S; \theta)). \quad (8.2)$$

Generating new samples. Starting from a random noise sample Z , we aim to increase its probability of coming from the data. To do this, we use D as a surrogate model and maximize the objective $\log(D(G, Z, S))$. To do this, starting from a random noise sample, a stochastic gradient ascent based solution executes the following until convergence

$$Z \leftarrow Z + \gamma \nabla_Z \log(D(G, Z, S; \theta)). \quad (8.3)$$

Here, γ is a step parameter. The optimization loop converges to Z^* that should have a high $D(G, Z^*, S; \theta)$ probability and represents a reconstruction of the next window. The working intuition is that the surrogate surface is highly non-convex and optimization over diverse noise samples would lead to distinct local optima. In principle, every optimum corresponds to a unique element from the data distribution, allowing the GON model to generate a diverse set of samples. Alternatively, we could stop after each step with probability $D(G, Z, S; \theta)$. A caveat of this approach is that sample generation could be slow as we run optimization in the input space. For some applications this additional overhead might be negligible or acceptable [Goli et al., 2020, Huang et al., 2020].

Training using generated samples. We consider the new samples as self-supervised fake ones [Chen et al., 2020] not belonging to the data and train D using the cross-entropy loss by descending the gradient

$$-\nabla_{\theta} \log(1 - D(G, Z^*, S; \theta)). \quad (8.4)$$

Informally, as D becomes a better discriminator, minimizing (8.2) and (8.4), the generated samples should also be close to the data.

8.3 DRAGON Based Worker Resilience

8.3.1 Optimizing GONs for fault-detection

The above described GON framework is for generic data generation. However, for the specific case of anomaly/fault detection, we leverage some of its properties to optimize GONs. Consider a GON model that takes inputs the graph topology G , time-series window W and scheduling decision S for each LEI. The output of such a GON needs to be a reconstruction of the next window

$$\hat{W} = D(G, W, S; \theta).$$

Window Initialization. Firstly, exploiting the temporal trend of time-series data, instead of starting from a random noise sample Z we can start from the window W and optimize $D(G, Z, S; \theta)$. This is because of the high correlation in the time-series windows of the current and next scheduling intervals in typical edge environments. Thus, we initialize

$$Z = W.$$

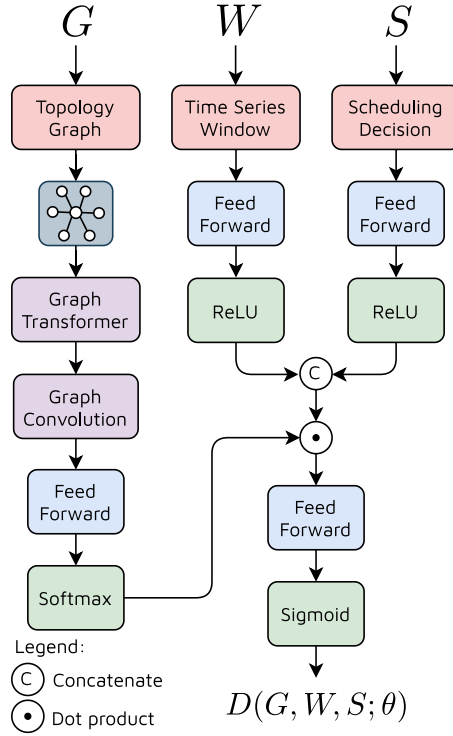


Figure 8.3: Neural network used in DRAGON.

Second-Order Optimization. Secondly, second-order optimization methods have been shown to be much faster for diverse optimization setups [Yao et al., 2021]. So, instead of using a first-order gradient optimization in (8.3), we can use higher-order variants. Thus, we could use

$$Z \leftarrow Z + \gamma [\nabla_Z^2 \log(D(G, Z, S; \theta))]^{-1} \nabla_Z \log(D(G, Z, S; \theta)). \quad (8.5)$$

However, computing the inverse of the second-order gradient (Hessian) is expensive [Kochenderfer and Wheeler, 2019, § 6.3]. This is alleviated by adapting the Hutchinson’s method and using the moving average of the approximate Hessian diagonal to speed up convergence. This is done by using the **AdaHessian** optimizer [Yao et al., 2021]. We also use warm restarts using cosine annealing [Loshchilov and Hutter, 2018] to speed up convergence.

GON Network. It is crucial that the D network is able to capture the temporal trends in the time-series data with the scheduling decision to effectively predict a reconstruction of the next time-series window. The model used in our approach is a composite neural network shown in Fig. 8.3. Considering we have p tasks running as a sum of new and active tasks, the scheduling decisions of these are encoded as one-hot vectors of size m (number of hosts in the LEI). We thus get a matrix of scheduling decisions (S) of size $[p \times m]$. Also, the n features of m hosts and p tasks in the form of resource utilization metrics, each state window (W) is encoded as a $[(m + p) \times n \times k]$ tensor. All operations are performed in a factored fashion, *i.e.*, the neural network operates on S as a batch of p vectors, each having dimension m , and on W as a batch of size $m + p$ tensors of size $n \times k$. To encode inputs W and S , we use feed-forward layers to bring down the dimension size of the input and ReLU

activation:

$$\begin{aligned} E^W &= \text{ReLU}(\text{FeedForward}(W)), \\ E^S &= \text{ReLU}(\text{FeedForward}(S)). \end{aligned} \quad (8.6)$$

The topology of the edge federation is represented as a graph with nodes as edge hosts and edges corresponding to edge groups. All edge workers are connected via undirected edges to their respective broker, and all brokers are connected to each other broker. The resource utilization characteristics of each edge host are then used to populate the feature vectors of the nodes in the graph. We denote the feature vector of host h_i as e_i . We encode this graph G using a graph transformer network [Yun et al., 2019] for scalable computation over the input graph. The motivation behind using a graph transformer network is to allow computation across cross-LEI paths that are defined as paths from a worker in one LEI to another worker in a different LEI. This is done by modifying the input graph by graph-transformer operations that generate a transformed graph with cross-LEI paths as edges. As we run preemptive migrations across LEIs, this facilitates prediction of effects on QoS and resource utilization metrics in the next interval when migrating task along the cross-LEI paths. Thus, we transform the input graph G as

$$G_1 = \text{GraphTransformer}(G). \quad (8.7)$$

We then pass the graph through a graph convolution network [Zhang et al., 2019] to capture the inter host dependencies by convolving feature vectors across the cross-LEI paths in G , *i.e.*, the edges in G_1 . Here, the features for host h_i are aggregated over one-step connected neighbors $n(i)$ in the graph over r convolutions, resulting in an embedding e_i^r for each host node in the graph. Here, r denotes the number of convolution operations and is typically kept less than the number of LEIs. Specifically, this results in *graph-to-graph* updates as:

$$\begin{aligned} e_i^0 &= \tanh(W e_i + b), \\ e_i^q &= \sum_{j \in n(i)} W^q e_j^{q-1}, \end{aligned} \quad (8.8)$$

where W^q are the parameters of the graph convolution block in Figure 8.3. The stacked representation for all hosts is represented as E^H . Now, we pass this representation through a feed-forward layer with softmax activation that allows us to use this as attention weights

$$E_1^H = \text{Softmax}(\text{FeedForward}(E^H)). \quad (8.9)$$

To generate the final window reconstruction, we use dot-product attention [Chorowski et al., 2015] as

$$D(G, W, S; \theta) = \text{Sigmoid}(\text{FeedForward}(E_1^H \cdot [E^W, E^S])). \quad (8.10)$$

Here, the sigmoid function allows us to bring the output in the range $[0, 1]$, the same as that of the normalized true window.

Algorithm 8.1 Minibatch stochastic gradient based training of generative optimization networks. Input is dataset Λ and hyperparameters m and γ . In experiments we use **AdaHessian** optimizer for line 4 and **Adam** for line 5.

Require: Dataset $\Lambda = \{W_0, \dots, W_T\}$

- 1: **for** number of training iterations **do**
- 2: Sample minibatch of size m window samples $\{W^{(1)}, \dots, W^{(m)}\}$ from Λ .
- 3: Sample minibatch of size m noise samples $\{Z^{(1)}, \dots, Z^{(m)}\}$ such that $Z^{(i)} = W^{(i)}$.
- 4: Generate new samples $\{Z^{*(1)}, \dots, Z^{*(m)}\}$ by running the following till convergence

$$Z \leftarrow Z + \gamma [\nabla_Z^2 \log (D(G, Z, S; \theta))]^{-1} \cdot \nabla_Z \log (D(G, Z, S; \theta)).$$

- 5: Update the discriminator $D(\cdot; \theta)$ by ascending the stochastic gradient.

$$\nabla_{\theta} \frac{1}{m} \sum_{i=1}^m [\log (D(G, W^{(i)}, S; \theta)) + \log (1 - D(G, Z^{*(i)}, S; \theta))].$$

- 6: **end for**
-

8.3.2 Offline Model Training

A more pragmatic implementation of GON training is summarized in Alg. 8.1, where we combine the first and last steps in Section 8.2 when training D using minibatches. The training procedure runs offline, *i.e.*, before executing any workloads on the federated edge setup, training the neural network in a GPU equipped machine. The input to the training framework is a dataset of time-series windows $\Lambda = \{W_0, \dots, W_T\}$. We first randomly sample m window samples from Λ (Z^* s in line 2). We also initialize noise samples as the window samples (line 3). We then perform **AdaHessian** optimization on the noise samples to obtain generated windows (line 4). Finally, we train the model using a combined cross-entropy loss of generated ($Z^{*(i)}$) and data samples ($W^{(i)}$), as described in Section 8.2 (line 5):

$$L = \log (D(G, W^{(i)}, S; \theta)) + \log (1 - D(G, Z^{*(i)}, S; \theta)). \quad (8.11)$$

8.3.3 Decentralized Fault Tolerance using GON and Co-Simulations

The working of the DRAGON framework is summarized in Alg. 8.2. The algorithm runs in each broker node of the federated edge setup where the neural network is periodically fine-tuned to adapt to changing environment and workload characteristics. We now elucidate the different steps in detail.

Fault Detection. For unsupervised fault detection using the GON framework, for each input datum from a multivariate time-series, we create a reconstruction using a neural network D . Now, for an input window W_t , we create a reconstruction of the next window $\hat{W}_{t+1}$, using a trained model D and the **AdaHessian** based version of (8.5) (line 5). However, to generate an approximation of the next window, we leverage co-simulation to quickly run discrete-event simulations alongside and synchronously with the running computer system. Similar to digital-twins, this allows us to continuously capture the system state and get an estimate of the utilization metrics of a future state of the system, further allowing proactive fault remediation. Having the scheduling decision S_t (line 2) and the current window W_t , we denote the discrete-event simulation as (line 4)

$$W_{t+1} \leftarrow \text{Sim}(W_t, S_t). \quad (8.12)$$

Algorithm 8.2 The DRAGON fault-tolerance algorithm**Require:**

Trained GON Model $D(\cdot; \theta)$;
Set of hosts Ω

- 1: **for** $t = 0$ to T **do**
- 2: **for each** broker **do**
- 3: S_t from underlying scheduler
- 4: $W_{t+1} \leftarrow \text{Sim}(W_t, S_t)$
- 5: $\hat{W}_{t+1} \leftarrow \text{AdaHessian}(D(G_t, W_t, S_t; \theta))$
- 6: $f = \|\text{ReLU}(W_{t+1} - \hat{W}_{t+1})\|$
- 7: $l = \mathbb{1}(f \geq \text{POT}(W_t))$
- 8: Compile time-series windows of all brokers as W_t^M
- 9: Faulty hosts $\omega \subseteq \Omega$ that have $l = 1$
- 10: Randomly initialize migrations from ω as M_t
- 11: $M_t^* \leftarrow \text{SimAnnealing}(M_t, \text{Fitness} = (8.18))$
- 12: **for each** allocation (t, h) in M_t^* **do**
- 13: **if** t in LEI and h not in LEI **do**
- 14: Migrate t to LEI corresponding to h
- 15: $L = \log D(G, W_{t+1}, S; \theta) + \log(1 - D(G, \hat{W}_{t+1}, S))$
- 16: Fine-tune D using L and Adam

To generate the fault score we only consider upward spikes of the true window from the reconstructed window. This is due to the nature of our state data, *i.e.*, resource utilization metrics leading to faults only when there is a sudden increase in CPU/RAM/disk/network consumption. The ReLU activation is apt for this as it gives a zero fault score when the true utilization metrics are lower than the predicted ones. Thus, as in line 6

$$f = \|\text{ReLU}(W_{t+1} - \hat{W}_{t+1})\|. \quad (8.13)$$

Once we have the fault scores, we generate fault labels using the Peak Over Threshold (POT) method [Siffer et al., 2017]. POT chooses the threshold automatically and dynamically, above which an fault score is classified as a true class label. Thus, our fault label (line 7) is calculated as

$$l = \mathbb{1}(f \geq \text{POT}(W_t)). \quad (8.14)$$

Comparing the resource utilization metrics of each host separately, the edge broker generates a fault-label for each edge device in its LEI.

Co-Simulated Fictitious Play. Now that we predict the edge hosts that could possible have fault in the next interval, we now need to decide the tasks that should be preemptively migrated from these nodes. To do this, we again use the co-simulation technique. A co-simulator can also give us the QoS score of the next state for resource utilization metrics of all nodes instead of the LEI, denoted by W_t^M (line 8), and a preemptive migration decision M_t . We denote the co-simulated QoS score and the next window by

$$QoS_{t+1} \leftarrow \text{Sim}^{\text{QoS}}(W_t^M, M_t), \quad (8.15)$$

$$W_{t+1}^M \leftarrow \text{Sim}(W_t^M, M_t). \quad (8.16)$$

This score could be a convex combination of multiple metrics like SLO violation rate, energy consumption, response time, etc. However, for this work, we only use response time and energy consumption as these are

the most important metrics in edge federations [Gouasmi et al., 2018, Cao et al., 2020, Shao and Zhang, 2020]. Thus,

$$\text{QoS}_{t+1} = \alpha \cdot \text{ART}_{t+1} - \beta \cdot \text{AEC}_{t+1}. \quad (8.17)$$

Here, ART_{t+1} is the average response time for all tasks leaving the system in interval I_{t+1} normalized by the maximum response time until the current interval. AEC_{t+1} is the average energy consumption of the infrastructure (which includes all edge hosts) in interval I_{t+1} normalized by the maximum power of the hosts. Similar to Chapter 3, α and β are hyper-parameters set by the industry adopters or the cloud service providers based on the relative importance of the two optimization objectives (energy consumption and response time) for the running services. These values may be obtained using grid-search before deploying the services. We assume that the same hyperparameters work even for fine-tuning. We performed grid-search online as well and did not find any difference in the final values obtained. Also, using W_{t+1}^M we detect the number of anomalies in I_{t+1} . We denote this by N_{t+1} and use the normalized value N_{t+1}/H , where H is the total number of hosts in the federated environment (in our case $H = 16$). To keep into account the costly migration overhead, we also include the migration overhead score O_t that is the migration time of all tasks in M_t normalized by the maximum migration time until the current interval. We define a fitness score as

$$\text{Fitness}_{t+1}(W_t^M, M_t) = \text{QoS}_{t+1} + \delta_1 \cdot \frac{N_{t+1}}{H} + \delta_2 \cdot O_t, \quad (8.18)$$

where δ_1 and δ_2 are hyperparameters set based on user requirements. The motivation behind this fitness score is that when we optimize a preemptive migration decision, we need to make sure that we consider the immediate reward in terms of the QoS score, but also consider the migration overheads and number of faults in the next interval. This entails the minimization of the fitness score given in (8.18).

Thus, at the start of every interval, each broker node gets the time-series window of other LEIs, combines them to form W_t^M and runs a simulated execution to get the next window and QoS scores. This kind of distributed execution in a multi-agent setup to estimate objective function of a global environment is commonly referred to as “fictitious play” [Eksin and Ribeiro, 2017].

Preemptive Migration Decision. The co-simulated fictitious-play described earlier is a very fast and lightweight operation compared to an actual execution of a preemptive migration decision. In fact, running a simulation for W_t^M and M_T is so fast that testing for all possible M_t takes ~ 0.5 seconds second for 16 nodes. This increases linearly, by around 5% for each new node in the system. Still, for scalability, we run simulated annealing on M_t to optimize the fitness score (line 13). A neighbor of a decision M_t is defined by either migrating a task not in M_t to another LEI or migrating a task in M_t to a different LEI than in M_t . Thus, optimizing fitness score from a preemptive migration decision, which randomly selects tasks to be migrated from faulty hosts detected by (8.14) (line 4), converges to the decision M_t^* . This is the output of the DRAGON model. Note that each broker generates such a decision and migrates the set of tasks in M_t^* that are active in its LEI to the target LEI group (lines 21-23). All brokers altruistically accept incoming tasks from other LEIs and consider them new tasks in their group. The fault labels are also combined using logical or, for each interval from t to $t + N - 1$ to generate the fault label for DRAGON.

Extending to DRAGON+. As DRAGON just optimizes the fitness score for the next interval, it is myopic in its optimization capacity. To tackle this, we run multiple steps of DRAGON for a long-term optimization

Algorithm 8.3 The DRAGON+ fault-tolerance algorithm**Require:**

Trained GON Model $D(\cdot; \theta)$;
Set of hosts Ω ; Look-ahead intervals N ;

```

1: for  $t = 0$  to  $T$  do
2:   for each broker do
3:      $S_t$  from underlying scheduler
4:      $W_{t+1} \leftarrow \text{Sim}(W_t, S_t)$ 
5:      $\hat{W}_{t+1} \leftarrow \text{AdaHessian}(D(G_t, W_t, S_t; \theta))$ 
6:      $f = \|\text{ReLU}(W_{t+1} - \hat{W}_{t+1})\|$ 
7:      $l = \mathbb{1}(f \geq \text{POT}(W_t))$ 
8:     Compile time-series windows of all brokers as  $W_t^M$ 
9:     Faulty hosts  $\omega \subseteq \Omega$  that have  $l = 1$ 
10:    Initialize empty migration decision as  $M_t^+$ 
11:    for  $k = 0$  to  $N$  do
12:      Randomly initialize migrations from  $\omega$  as  $M_{t+k}$ 
13:       $M_{t+k}^* \leftarrow \text{SimAnnealing}(M_{t+k}, \text{Fitn.} = (8.18))$ 
14:       $QoS_{t+k+1} \leftarrow \text{Sim}^{\text{QoS}}(W_{t+k}^M, M_{t+k})$ 
15:       $W_{t+k+1}^M \leftarrow \text{Sim}(W_{t+k}^M, M_{t+k})$ 
16:      for each allocation  $(t, h)$  in  $M_{t+k}^*$  do
17:        if  $\exists h^+ | (t, h^+) \in M_t^+$  do
18:          Replace  $(t, h^+)$  with  $(t, h)$ 
19:        else do
20:          Add  $(t, h)$  to  $M_t^+$ 
21:      for each allocation  $(t, h)$  in  $M_t^*$  do
22:        if  $t$  in LEI and  $h$  not in LEI do
23:          Migrate  $t$  to LEI corresponding to  $h$ 
24:       $L = \log D(G, W_{t+1}, S; \theta) + \log(1 - D(G, \hat{W}_{t+1}, S))$ 
25:      Fine-tune  $D$  using  $L$  and Adam

```

strategy and call this DRAGON+ (summarized in Alg. 8.3). Assume, for example, that in the first iteration DRAGON generated a decision M_t^* . Now, having W_{t+1}^M , we can generate S_{t+1} , and subsequently QoS_{t+1} from the underlying scheduler and run DRAGON again to generate M_{t+1}^* . We continue this for N intervals and combine $\{M_t^*, \dots, M_{t+N-1}^*\}$. To do this, if task t is allocated to h_i in M_p^* and h_j in M_q^* , then we select $h_{\max(p,q)}$. This is done because a migration to h_i then later to h_j has higher overhead than direct migration to h_j . This is achieved by maintaining a migration decision M_t^+ and updating the decision as we iterate from t to $t + N$ (see lines 16-20).

As the DRAGON+ approach performs optimization using a long-term QoS estimation, it can make more informed decisions. Further, DRAGON+ avoid unnecessary task migrations by looking at future migrations by DRAGON. However, the level of performance uplift depends on the volatility of the workloads and the look-ahead interval count N (see Section 8.5). Note that DRAGON is a special case of DRAGON+ where $N = 1$. Thus, for N higher than 1, the decision time and subsequently the scheduling time would increase, possibly affecting the overall performance of the system. A more empirical comparison is shown in Section 8.5.6.4. Overall, for computationally powerful brokers DRAGON+ is an appropriate choice whereas DRAGON is suitable in case of resource constrained brokers.

Model Fine-tuning. Now that we have a true window W_{t+1} and a generated window $\hat{W}_t$, we can fine-tune the model with this new datapoint using the loss

$$L = \log(D(G, W_{t+1}, S; \theta)) + \log(1 - D(G, \hat{W}_{t+1}, S)). \quad (8.19)$$

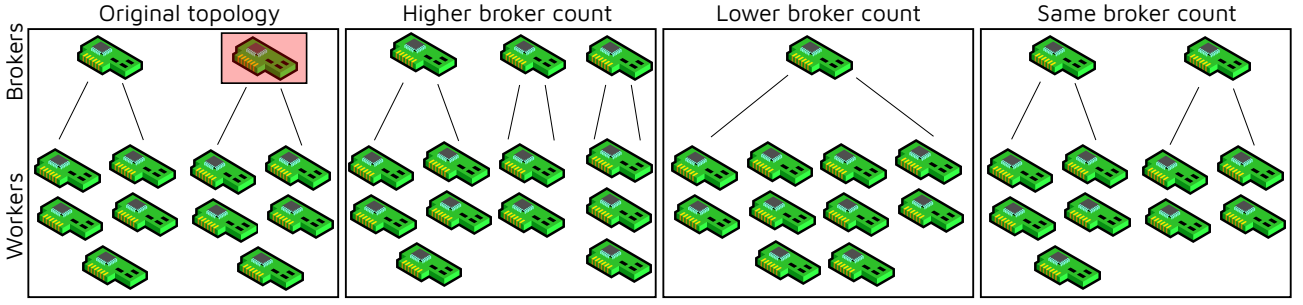


Figure 8.4: Possible node-shifts in CAROL. Nodes marked in red are the high consumption brokers that break-down.

This allows the model to dynamically adapt in non-stationary environments (lines 14-15).

To summarize, the novel contribution of this chapter is two-fold. First, we develop a decentralized fault-detection model based on the GON framework that uses significantly lower number of parameters than previous generative models to detect faults in each LEI independently. Second, we develop a synchronization strategy using co-simulated fictitious play to jointly decide of preemptive migrations that aim at avoiding faults arising in the system. Thus, we run migrations proactively, *i.e.*, predict faults using GONs and simulations, to avoid faulty states. For crash-like states, we restart tasks.

8.4 CAROL Based Broker Resilience

We now provide details on how the CAROL operations decides among the different types of node-shift operations that we consider in this work.

8.4.1 Node-Shifts

We assume that the network has tens of broker and worker nodes. In our work, the number of brokers and the system topology is known to all nodes in the federation. Whenever a broker node fails, we consider the worker nodes of the corresponding broker as being “orphaned”. In such an event, either one of the worker nodes is shifted to the broker layer, or another node in the broker layer manages the orphaned nodes. This shift of nodes from worker layer to broker layer gives rise to the name “node-shift” and is similar to the fault-tolerance schemes used in software defined networks (SDNs) [Samarji and Salamah, 2021]. Considering an input topology G , there are multiple types of node shifts that may increase, decrease or even keep the broker count static. Three types of *worker-to-broker* node-shift mechanisms considered in this work are described below with a visualization presented in Figure 8.4 instead. We similarly consider the respective counterparts as the *broker-to-worker* node-shifts.

- *Type 1*: If a broker node fails, two of the orphaned nodes may be shifted to the broker layer and the remaining orphaned nodes may be evenly distributed among these two new brokers. This increases the broker count by one, also increasing the management capacity of the system for higher throughput at the broker layer.
- *Type 2*: The orphaned nodes could be assigned to another active broker in the system. This decreases the broker count than before failure and increases the computational throughput of the worker layer.

- *Type 3*: One of the orphaned worker nodes may be assigned to be the broker for the other worker nodes. This keeps the same number of broker nodes as before the failure. The three node-shift types may be utilized to trade off the throughputs of the broker and worker layers.

However, in heterogeneous edge federations, where nodes with disparate resource capacities may be present, the choice of the worker node to shift to the broker layer becomes vital. Moreover, with heterogeneous broker nodes, the choice of the broker nodes to which the orphaned nodes are assigned affects the QoS of the system. Another crucial factor is the overhead corresponding to the node-shift operations. This is due to the initialization of the broker management systems and synchronization of the system topology with other brokers in the federation.

Another critical aspect in finding the best graph topology is the workload heterogeneity across broker nodes. As shown in Figure 8.4, the node operations can lead to a different number of worker nodes and subsequently computational throughput of disparate LEIs. A single node-shift step might not be sufficient to manage resources in case of different computational loads across LEIs. Thus, a sequence of node-shifts may need to be tested to discover the topology with optimal QoS. However, generating QoS scores for a large number of node-shift sequences may not be feasible in latency-critical settings where generating QoS scores by execution or simulation might be a time-consuming activity. This is because a node-shift entails transfer of broker level data to the worker node and initializing management software containers, both of which give rise to a higher latency.

8.4.2 Confidence-Aware Model

To eschew the costly execution of multiple node-shift sequences and observe their effects on QoS, we need a lightweight model that mimics the behavior of a computationally expensive simulation or execution [Kochenderfer and Wheeler, 2019]. Such models are referred to as “surrogate models” in the literature [Kochenderfer and Wheeler, 2019]. Using a surrogate model enables such methods to run optimization in the input space and generate decisions, such as node-shift sequences that give G_t , giving high estimated QoS. Surrogate optimization techniques, such as GOBI, use gradient-based optimization compared to evolutionary search strategies that facilitate quick convergence. However, in discrete search spaces, such as graph topologies, such techniques typically select the discrete point close to the converged point in the continuous space. This may often lead to non-optimal solutions [Kochenderfer and Wheeler, 2019, §19.1]. Another drawback of such methods is that the parameters of the neural networks need to be periodically fine-tuned, leading to high overheads [Tuli et al., 2022]. To reduce such overheads, it is crucial to fine-tune models only when the system or workload configurations change. To do this, we take motivation from confidence-aware deep learning [Moon et al., 2020] and GONs, to integrate in CAROL a memory efficient QoS surrogate. This is used to predict a confidence score, such that for low confidence the model can be fine-tuned with online generated data.

GON based Neural Network. Consider a discriminator network D with parameters θ that takes as input graph topology G , performance metrics M and scheduling decision S . The output of D , i.e., $D(M, S, G; \theta)$ is a likelihood score of G , M and S belonging to a distribution. When trained for a dataset generated from the normal execution of an edge federation, the GON model predicts a high score for an input tuple (M, S, G) in the distribution of the dataset and a low score for an unseen tuple. This allows us to translate the likelihood score output as a measure of the confidence of the network. Essentially, for dynamic systems, when the confidence

score drops below a threshold, it indicates us to fine-tune the model. This enables parsimonious model fine-tuning, giving us an effective compromise between prediction performance and training overheads.

To generate the performance scores for an input graph topology G , we can randomly initialize M and maximize the discriminator output by ascending the stochastic gradient

$$M \leftarrow M + \gamma \nabla_M \log (D(M, S, G; \theta)), \quad (8.20)$$

where γ denotes the step size in the optimization loop. We use log-likelihood instead of likelihood scores for training stability [Goodfellow et al., 2014]. Thus, starting from an input (S, G) pair, the above optimization loop converges us to give performance metrics M^* and a confidence score of $D(M^*, S, G)$.

Offline Model Training. To train the GON model D , we first collect an execution trace $\Lambda = \{M_t, S_t, G_t\}_{t=0}^T$. A summary of the training process is presented in Algorithm 8.1. We leverage an adversarial training process where we use the generated data (say Z^*) as fake samples and datapoints from Λ (say M) as real samples and train the model using the binary cross-entropy loss

$$L = \log (D(M, S, G; \theta)) + \log (1 - D(Z^*, S, G)), \quad (8.21)$$

where $(M, S, G) \in \Lambda$. Thus, the model is trained to optimize the likelihood scores for fake and real samples in an adversarial fashion using the same loss function. This adversarial style training has a two-fold goal. First, the model learns to generate new samples (Z^*) such that the tuple (Z^*, S, G) belongs to the data distribution. Thus, with sufficient training, converged Z^* are close estimates of the performance metrics for inputs G and S . Second, for previously seen datapoints, *i.e.*, (M, S, G) tuples, the predicted confidence scores are high. If the (S, G) input is unseen during training, then the converged confidence score $D(Z^*, S, G)$ would also be low. In practice, for the scheduling interval I_t , instead of starting from a random noise sample Z , we initialize M as M_{t-1} and then converge $D(M, S_t, G_t)$ to M_t . This exploits the temporal correlation between subsequent system states, facilitating quick convergence of (8.20).

8.4.3 Finding optimal edge topology

Having a surrogate model, at each scheduling interval I_t , we can now run optimization of the QoS by taking a convex combination of the QoS metrics in M_t . We denote this objective function as $O(M_t)$. Now, if a node fails in I_{t-1} , then we can create the graph topology G_t by a node-shift operation on G_{t-1} (line 4 in Alg. 8.4). However, to decide the optimal node-shift operation, we can run a local search in the topology space. We select the tabu search algorithm due to its deterministic nature and empirically faster convergence for the specific optimization problem we consider [Connor and Shea, 2000]. Thus, in case of a node failure, we start from graph G_{t-1} , apply a random node shift to generate G and use tabu search to generate the graph G_t such that the QoS score is optimized (line 8 in Alg. 8.4). We perform this iteratively for each failed broker node to achieve the final graph topology. If no broker failed in I_{t-1} , then we add the datapoint to a running dataset Γ (line 10 in Alg. 8.4).

The local neighbors of a graph G for a failed node, say b , is obtained by performing all possible node-shift operations. We denote this neighbourhood set of G by $N(G, b)$ (line 7 in Alg. 8.4). The QoS score is obtained

Algorithm 8.4 The CAROL resilience model.**Require:**Trained GON Model D ; Running dataset Γ Broker set B ; Worker set W

```

1: for  $t = 0$  to  $T$  do
2:    $S_t$  from underlying scheduler
3:    $M_t$  from edge system
4:    $G_t \leftarrow \text{NodeShift}(G_{t-1})$  ▷ Initialize topology
5:   for each broker  $b \in B$  do
6:     if  $b$  fails do
7:        $G \in N(G_t, b)$  ▷ Random node-shift
8:        $G_t \leftarrow \text{TabuSearch}(G, \Omega)$  ▷ Tabu Search
9:   if no broker in  $B$  fails do
10:    Add datapoint  $(M_t, S_t, G_t)$  to  $\Gamma$  ▷ Save datapoint
11:     $C \leftarrow D(M_t, S_t, G_t; \theta)$  ▷ Confidence score
12:     $POT \leftarrow \text{PeakOverThreshold}(C)$  ▷ POT calculation
13:    if  $C < POT$  do
14:       $L = \log(D(M, S, G; \theta)) + \log(1 - D(Z^*, S, G))$ 
15:      Fine-tune  $D$  using  $L$  and Adam ▷ Fine-tune
16:      Clear  $\Gamma$ 
17:      Configure topology as  $G_t$ 
18:      Schedule tasks using  $S_t$ 

```

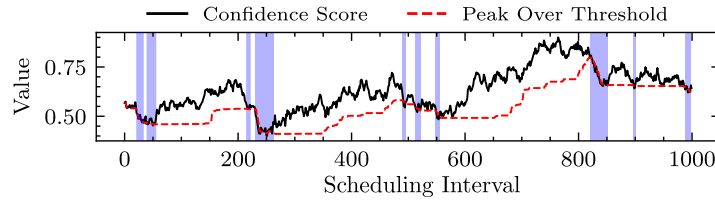


Figure 8.5: Visualizing the confidence scores and POT threshold values for 1000 scheduling intervals in CAROL.

using the GON model described previously, where we evaluate M_t using (8.20) over $D(M, S_t, G)$ and use the objective function $O(M_t)$ for our local search. For notational convenience, we define the objective score of a graph G , parameterized by the GON model D , scheduling decision S_t and objective function O as $\Omega(G; D, S_t, O)$. As the network size increases, the number of possible node-shift operations also increases. Thus, we use a tabu list of a fixed size L in our search scheme.

8.4.4 Confidence Aware Fine-Tuning

After performing optimization in the topology space and obtaining G_t for interval I_t , we also evaluate the confidence score $D(M_t, S_t, G_t; \theta)$ (line 11 in Alg. 8.4). We then use the Peak Over Threshold (POT) method [Siffer et al., 2017] that dynamically chooses threshold values below which we fine-tune our model (line 12). POT uses extreme value theory to generate a threshold value from a continuous record of past confidence scores. It does this by checking the peak values reached for any period during which values fall below a certain threshold. This threshold is dynamically updated based on incoming data to ensure that the model adapts to non-stationary settings. In any interval, if the confidence score falls below the POT threshold, CAROL fine-tunes the GON network with the latest collected dataset Γ (line 15). A visualization of the training process is shown for a thousand scheduling intervals in Figure 8.5. The figure clearly demonstrates that the model trains the GON network only when there are dips in the confidence scores, allowing our technique to have much lower fine-tuning overheads compared to other methods that tune their neural networks at each scheduling interval.

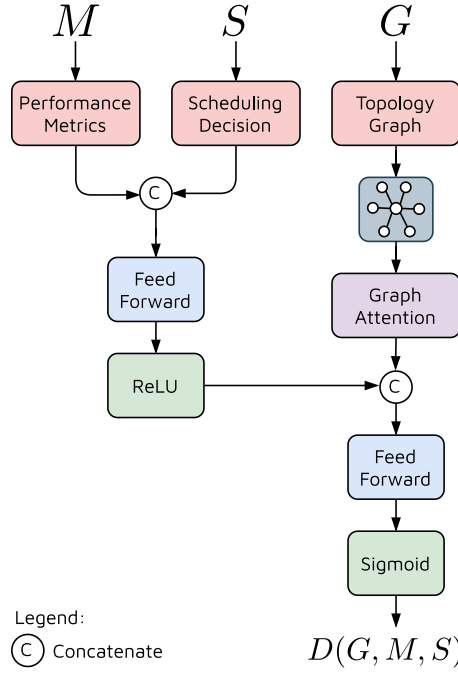


Figure 8.6: Neural network used in CAROL.

8.4.5 GON Network

It is crucial that the D network is able to capture the correlation between system topology and scheduling decisions with the performance metrics to effectively predict QoS and confidence scores. The model used in our approach is a composite neural network shown in Figure 8.6 that infers the correlations between performance metrics, scheduling decision and graph topology to generate the discriminator output. Considering we have p tasks running as a sum of new and active tasks, the scheduling decisions of these are encoded as one-hot vectors of size $|H|$ (number of hosts in the system). We thus get a matrix of scheduling decisions (S) of size $[p \times |H|]$. Also, we collect performance metrics of each host $i \in H$ denoted by M_i that includes resource utilization metrics CPU, RAM, Disk, Bandwidth consumption with QoS metrics such as energy consumption and SLO violation rates. We denote the resource utilization metrics of host i as u_i and QoS metrics as q_i . We also include the task resource utilization consumption with SLO deadlines, denoted as t_i . Thus, $M_i = [u_i, q_i, t_i]$. The M_i vectors for all hosts are stacked together to form the matrix M . To encode inputs M and S , we use feed-forward layers to bring down the dimension size of the input and ReLU activation:

$$E^{\{M, S\}} = \text{ReLU}(\text{FeedForward}([W, S])). \quad (8.22)$$

The topology of the edge federation is represented as a graph with nodes as edge hosts and edges corresponding to edge groups. All edge workers are connected via undirected edges to their respective broker, and all brokers are connected to every other broker. The resource utilization characteristics of each edge host are then used to populate the feature vectors of the nodes in the graph. We denote the feature vector of host i as e_i that are computed from u_i . We encode this graph G using a graph attention network [Veličković et al., 2018] for scalable computation over the input graph. The motivation behind using a graph attention network is to allow computation over the graph to be agnostic to the number of nodes in the system topology. Graph attention operation performs convolution operation for each node over its neighbors and uses dot product self-attention

to aggregate feature vectors. Thus, we transform the input graph G using *graph-to-graph* updates as:

$$e_i = \sigma \left(\sum_{j \in n(i)} W^q \cdot \tanh(W u_i + b) \right), \quad (8.23)$$

where W^q are the attention weights obtained by the weight matrix W [Veličković et al., 2018] and $n(i)$ are the neighbors of host i in graph G . The stacked representation for all hosts (e_i) is represented as E^G . Now, we pass this representation through a feed-forward layer with sigmoid activation as

$$D(M, S, G; \theta) = \text{Sigmoid}(\text{FeedForward}([E^{\{M, S\}}, E^G])). \quad (8.24)$$

Here, the sigmoid function allows us to bring the output in the range $[0, 1]$, the same as that of the true normalized window.

8.4.6 Objective function

For generating the objective function $O(M_t)$ as part of Ω in Alg. 8.4, we use a convex combination of the energy consumption and SLO violation rates of the system as is commonly used in prior work to estimate system performance. For the energy consumption (q_i^{energy}) and SLO violation rates (q_i^{slo}) of each host i , we generate scores for the complete system

$$\begin{aligned} q^{\text{energy}} &= \sum_{i \in H} q_i^{\text{energy}}, \\ q^{\text{slo}} &= \sum_{i \in H} q_i^{\text{slo}}. \end{aligned} \quad (8.25)$$

We then compute the QoS score as

$$O(M) = \alpha \cdot q^{\text{energy}} + \beta \cdot q^{\text{slo}}. \quad (8.26)$$

8.5 Performance Evaluation

We now provide details on how we evaluate the CAROL and DRAGON models. For our evaluation, we use the same setup as described in Chapter 7.

8.5.1 Workloads

Compared to the DeFog workloads, we now shift to resource heavier workloads for more frequent node failures, which uses *AIoTBench* applications [Luo et al., 2018]. AIoTBench is a AI based edge computing benchmark suite that consists of various real-world computer vision application instances. The seven specific application types correspond to the neural networks they utilize. These include three typical heavy-weight networks: ResNet18, ResNet34, ResNext32x4d, as well as four light-weight networks: SqueezeNet, GoogleNet, MobileNetV2, MnasNet.¹ This benchmark has been used in our experiments due to its volatile utilization characteristics and heterogeneous resource requirements. The benchmark consists of 50,000 images from the COCO dataset as

¹AIoTBench: <https://www.benchcouncil.org/aibench/aiotbench/index.html>. Accessed: 5 October 2021.

workloads [Lin et al., 2014]. To evaluate the proposed method in a controlled environment, we abstract out the users and IoT layers described in Chapter 2 and use a discrete probability distribution to realize tasks as container instances. Thus, at the start of each scheduling interval, we create new tasks from a Poisson distribution with rate $\lambda = 1.2$, sampled uniformly from the seven applications. The Poisson distribution is a natural choice for a bag-of-tasks workload model, common in edge environments [Mao et al., 2016, Basu et al., 2019]. Our tasks are executed using Docker containers. We run all experiments for 100 scheduling intervals, with each interval being 300 seconds long, giving a total experiment time of 8 hours 20 minutes. We average over five runs and use diverse workload types to ensure statistical significance of our experiments.

8.5.2 Fault Injection Module

To generate broker failures at test time, we use an existing fault-injection module [Ye et al., 2018]. We create faults of the type: CPU overload, RAM contention, Disk attack and DDOS attack. In a CPU overload attack, a simple CPU hogging application is executed that creates contention of the compute resources. In RAM contention attack, a program is run that performs continuous memory read/write operations. In disk attacks, we run the IOZone benchmark that consumes a large portion of the disk bandwidth. In a DDOS attack, we perform several invalid HTTP server connection requests causing network bandwidth contention. More details are given in [Ye et al., 2018]. We generate faults using a Poisson distribution with the rate $\lambda_f = 0.5$, sampled uniformly at random from the attack set. These attacks were performed to cause the failure of broker nodes.

8.5.3 Broker Failure Detection and Recovery

To evaluate the CAROL model, all broker nodes periodically (every 30 seconds) `ping` each other to test which hosts are active in the system. Five Internet Control Message Protocol (ICMP) packets are sent using the `ping` utility to every other broker and the node waits for a response with a timeout counter set to 10 seconds. If the broker responds, we run the audit checking procedure that verifies the signed log entries since the previous audit [Haeberlen et al., 2006]. If a broker is unresponsive or the audit check fails, an entry is made in a shared PostgreSQL database. For a broker $b \in B$, if all other brokers report it as unresponsive, then b is assumed to be compromised. A ‘reboot’ command is run on this broker in a non-blocking fashion using the `ssh` utility.

As described in Section 8.1, we assume temporary failures in our setup. Thus, after a broker node fails, it takes 1-5 minutes for this node to restore to the previous state and resume computation/management. To ensure smooth execution of the system, we use the Virtual Router Redundancy Protocol (VRRP) to assign a set of virtual IPs to the broker nodes in the system. We implement this using the `keepalived` high-availability toolkit.² As soon as a failed node comes back online, we add it to the graph topology and assign it as a worker in the closest active broker as per network latency. This is performed at the time of graph topology initialization at the start of each interval (line 4 of Alg. 8.4).

8.5.4 Fog Time Series Anomaly Detection (FTSAD) Dataset

The FTSAD is a suite of datasets collected on a Raspberry Pi 4B compute cluster [Tuli et al., 2021]. The cluster consists of five 4GB versions and five 8GB versions. We ran the DeFog benchmark applications [McChesney

²Keepalived. <https://github.com/acassen/keepalived>. Accessed 7 November 2021.

Table 8.1: Dataset Statistics

Dataset	Train	Test	Dimensions	Anomalies (%)
FTSAD-1	600	5000	1	12.88
FTSAD-25	574	1700	25	32.23
FTSAD-55	2158	2264	55	13.69

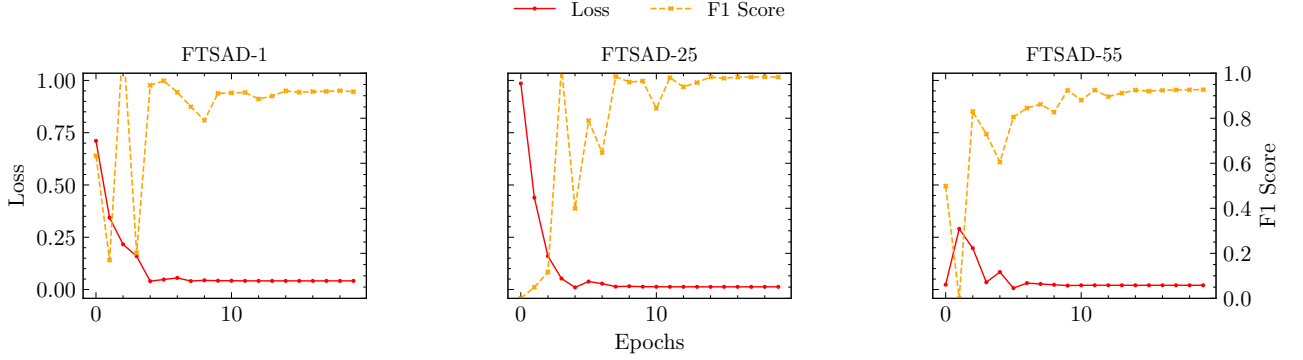


Figure 8.7: Training the GON model.

et al., 2019], specifically the Yolo, PocketSphinx and Aeneas workloads. We use a random scheduler and execute these tasks as Docker containers. We create faults of the type: CPU overload, RAM contention, Disk attack and DDOS attack [Ye et al., 2018]. In a CPU overload attack, a simple CPU hogging application is executed that create contention of the compute resources. In RAM contention attack, a program is run that performs continuous memory read/write operations. In disk attack, we run the IOZone benchmark that consumes a large portion of the disk bandwidth. In DDOS attack, we perform several invalid HTTP server connection requests causing network bandwidth contention. We generate faults using a Poisson distribution with rate $\lambda_f = 5$, sampled uniformly at random from the attack set. More details are given in [Ye et al., 2018].

For the 10-node cluster, we collect traces of resource utilizations including CPU, RAM, disk and network. Each datapoint is collected at the interval of 10 seconds. We truncate the dataset to have only 1, 25 or 55 dimensions and call these FTSAD-1/25/55 datasets. For FTSAD-1, we use the CPU utilization of one of the cluster nodes. For FTSAD-25, we use the CPU and network read bandwidth utilization traces for all 10 nodes and RAM utilization for the five 4GB nodes (as 8GB nodes rarely have memory contentions). For FTSAD-55, we use the CPU utilization, disk read, disk write, network read, network write bandwidths for all nodes and RAM utilization of the five 4GB nodes. The overall traces are collected for 737 minutes, giving 4422 datapoints. For the FTSAD-55 dataset, we split them at the 2158-th interval at which we started saving the anomaly labels as well, giving a test set size of 2264 datapoints. The FTSAD-1/25 datasets were manually curated to have low anomaly frequency in FTSAD-1 and high frequency in FTSAD-25. This allowed us to create a collection of datasets with diverse fault characteristics and dataset sizes. We summarize the characteristics of the three datasets in Table 8.1

8.5.5 Implementation and Training

We now provide details on the implementation of the CAROL and DRAGON models as extensions of the COSCO framework, and training of the GON network.

8.5.5.1 Implementation

We extend the COSCO framework to add support for decentralized decision making and worker/broker failures and recovery. COSCO uses the Checkpoint/Restore In Userspace (CRIU) [Venkatesh et al., 2019] tool for container migration. All sharing of resource utilization characteristics across brokers uses the `rsync` utility. We extend the `Framework` class to allow decentralized decision making. For synchronization of outputs and execution of workloads, we utilize the HTTP Notification API.

8.5.5.2 GON Training

We train the GON network using Alg. 8.1 on the three FTSAD datasets. All model training and experiments were performed on a system with configuration: Intel i7-10700K CPU, 64GB RAM, Nvidia RTX 3080 and Windows 10 Pro OS. The training data is collected offline and sent to this machine for model training. As the model is agnostic to the dimension size of the multivariate time series, our model can be trained for any of these datasets without changing the network size. For our experiments, we use the model trained on FTSAD-55 as it covers the highest number of parameters. The time-series windows and the scheduling decisions were taken directly from the datasets, whereas the graph topology was taken from the three configurations. For each configuration, we train a separate model. For training we randomly split the dataset into 80% training and 20% testing data. We use learning rate of 10^{-4} with weight decay of 10^{-5} . We use Adam optimizer for optimizing the loss function. The training curves showing loss and F1 scores on the test set are shown in Fig. 8.7. We use early stopping criterion for convergence. For hyperparameter selection, please refer to [Tuli et al., 2022] and [Tuli et al., 2022].

8.5.6 Results for DRAGON and DRAGON+

8.5.6.1 Comparison of Fault-Detection Performance

The detection and diagnosis scores for the FTSAD datasets are presented in Table 8.2. We do not present results on AIOtBench as it does not have fault labels. We use the *point-adjust* technique to determine whether a fault prediction is true or false. Specifically, faulty observations usually occur in the form of contiguous anomaly segments; thus, if at least one observation of an faulty time-series window is correctly detected, all the other observations of the segment are also considered as correct. The approach was first presented in [Audibert et al., 2020]. Compared to the baselines, the DRAGON and DRAGON+ models improve upon the accuracy, precision and recall by up to 17.28%-11.87%-14.14% and 18.32%-12.50%-14.34% respectively. Among the baselines, the StepGAN baseline has the highest average F1 score of 0.9040. The DRAGON and DRAGON+ models gives a higher average F1 scores of 0.9418 and 0.9453, *i.e.*, 4.18% and 4.56% higher than StepGAN respectively. In terms of AUROC, the TopoMAD baseline has the highest value of 0.9416. The proposed models outperform this score by 3.56%-4.08%, getting 0.9751 and 0.9800 scores. When comparing F1/GB, the deep learning methods have the lowest scores. Among the baselines, the TBAFT model has the highest average F1/GB score of 0.8252. DRAGON and DRAGON+ improve upon this by 17.94%-21.34%. Compared to the deep learning baselines, DRAGON and DRAGON+ improve the F1/GB by up to 305.03% and 316.68% respectively. In terms of diagnosis scores, StepGAN is the best baseline with HR of 0.7160 and NDCG of 0.7772. Compared to these scores, DRAGON and DRAGON give 5.01%-6.54% and 5.69%-8.95% improvement for HR and NDCG

Table 8.2: Comparison of detection and diagnosis scores with standard deviations. Results reported on the test set with 80:20 train-test split.

Method	Detection					Diagnosis		
	Accuracy	Precision	Recall	AUROC	F1	F1/GB	HR@100	NDCG@100
FTSAD-1								
PBFM	0.8526±0.0039	0.7362±0.0015	0.8412±0.0075	0.8713±0.0070	0.7852±0.0031	0.7325±0.0062	0.5461±0.0048	0.4716±0.0031
IoTEF	0.8713±0.0014	0.7814±0.0049	0.8721±0.0032	0.8827±0.0048	0.8243±0.0010	0.7418±0.0001	0.4958±0.0049	0.5125±0.0019
TBAFT	0.8967±0.0088	0.8012±0.0068	0.8516±0.0037	0.8561±0.0054	0.8257±0.0000	0.7737±0.0055	0.5112±0.0022	0.6019±0.0042
AWGG	0.7981±0.0039	0.7613±0.0039	0.8913±0.0055	0.8824±0.0056	0.8212±0.0024	0.4599±0.0006	0.5827±0.0044	0.6110±0.0010
TopoMAD	0.8924±0.0083	0.8385±0.0072	0.7123±0.0055	0.9012±0.0033	0.7703±0.0036	0.2095±0.0020	0.5918±0.0028	0.6512±0.0044
StepGAN	0.8987±0.0058	0.8622±0.0045	0.8345±0.0011	0.9066±0.0062	0.8481±0.0052	0.3420±0.0017	0.5946±0.0037	0.6622±0.0000
DRAGON	0.9101±0.0023	0.9524±0.0020	0.8872±0.0019	0.9418±0.0066	0.9186±0.0063	0.9494±0.0056	0.6124±0.0060	0.7013±0.0060
DRAGON+	0.9201±0.0020	0.9601±0.0037	0.8901±0.0003	0.9539±0.0056	0.9238±0.0061	0.9786±0.0070	0.6200±0.0016	0.7221±0.0025
FTSAD-25								
PBFM	0.6712±0.0060	0.9173±0.0008	0.7235±0.0054	0.5571±0.0022	0.8090±0.0048	0.7546±0.0057	0.6110±0.0054	0.7701±0.0048
IoTEF	0.9513±0.0012	0.9413±0.0009	0.9828±0.0026	0.8712±0.0040	0.9616±0.0008	0.8654±0.0040	0.6918±0.0039	0.8371±0.0032
TBAFT	0.9351±0.0061	0.9547±0.0046	0.9561±0.0064	0.8517±0.0033	0.9554±0.0058	0.8952±0.0041	0.6713±0.0004	0.8127±0.0070
AWGG	0.9013±0.0089	0.9326±0.0092	0.8913±0.0022	0.8091±0.0079	0.9114±0.0013	0.5104±0.0017	0.6542±0.0026	0.7817±0.0062
TopoMAD	0.9123±0.0019	0.9497±0.0075	0.9781±0.0002	0.9882±0.0065	0.9637±0.0050	0.2622±0.0009	0.7474±0.0011	0.8720±0.0057
StepGAN	0.9233±0.0022	0.9378±0.0056	0.9669±0.0028	0.9866±0.0022	0.9521±0.0018	0.3839±0.0023	0.7512±0.0041	0.8654±0.0010
DRAGON	0.9781±0.0040	0.9665±0.0075	0.9978±0.0082	0.9918±0.0048	0.9819±0.0008	1.0148±0.0014	0.7981±0.0034	0.9017±0.0053
DRAGON+	0.9812±0.0058	0.9665±0.0010	0.9998±0.0051	0.9934±0.0005	0.9829±0.0007	1.0412±0.0062	0.8013±0.0055	0.9172±0.0003
FTSAD-55								
PBFM	0.8912±0.0066	0.8721±0.0048	0.9163±0.0020	0.6029±0.0006	0.8937±0.0061	0.8336±0.0053	0.7612±0.0052	0.7410±0.0051
IoTEF	0.8888±0.0011	0.9038±0.0068	0.9424±0.0060	0.9011±0.0078	0.9227±0.0044	0.8304±0.0076	0.7812±0.0025	0.7717±0.0032
TBAFT	0.7812±0.0034	0.7673±0.0062	0.9801±0.0065	0.9635±0.0033	0.8607±0.0035	0.8065±0.0080	0.7911±0.0064	0.6526±0.0063
AWGG	0.8964±0.0029	0.9054±0.0076	0.9121±0.0069	0.9251±0.0019	0.9087±0.0046	0.5089±0.0046	0.8013±0.0026	0.7915±0.0070
TopoMAD	0.9350±0.0024	0.9105±0.0001	0.9212±0.0009	0.9355±0.0001	0.9158±0.0031	0.2491±0.0002	0.8129±0.0051	0.7998±0.0001
StepGAN	0.9361±0.0010	0.9128±0.0017	0.9107±0.0031	0.9078±0.0007	0.9117±0.0085	0.3676±0.0011	0.8022±0.0052	0.8041±0.0060
DRAGON	0.9440±0.0019	0.9038±0.0005	0.9469±0.0047	0.9916±0.0048	0.9248±0.0046	0.9558±0.0039	0.8452±0.0082	0.8612±0.0042
DRAGON+	0.9561±0.0094	0.9120±0.0068	0.9469±0.0081	0.9927±0.0077	0.9291±0.0080	0.9842±0.0012	0.8671±0.0040	0.9013±0.0079

respectively. These results show that having only a single discriminator network allows us to significantly reduce the network’s number of parameters and give a performance boost compared to having two such networks in StepGAN.

8.5.6.2 Comparison of QoS Metrics

We now present the QoS scores of all models for the three federated configurations in Fig. 8.8. Fig. 8.9 presents the average response time and SLO violation rates for each application in the AIOTBench workloads. To test DRAGON and DRAGON+, we use a GON network trained on the FTSAD-55 dataset and the scheduling decisions from the GOBI scheduler (see Chapter 3). Training time of the GON network on this dataset is 46.23 seconds. Among the baselines, StepGAN has the lowest energy consumption of 10.15 KW-hr averaged over the three configurations. The DRAGON+ model has the lowest energy consumption of 8.48 KW-hr, 16.40% lower than StepGAN. Second lowest is DRAGON with 8.79 KW-hr, 13.39% lower than StepGAN (Fig. 8.8(a)). We see similar trends for response time. The AWGG model struggles in the third configuration with heterogeneous resource capacities across LEIs due to the drop in the performance of reconstructive clustering. DRAGON and DRAGON+ have average response times comparable to the best baseline for each configuration. Comparing the energy consumption and response times across configurations, we observe that they are lower for the second configuration compared to the first. This is primarily because the second configuration has only 4 running instances of the fault-tolerance model due to four brokers. The third configuration has the highest energy consumption and response time due to the increasing difficulty in fault-tolerance in heterogeneous federated setups (Figs. 8.8(a) and 8.8(b)). The SLO violation rates remain mostly consistent across configurations,

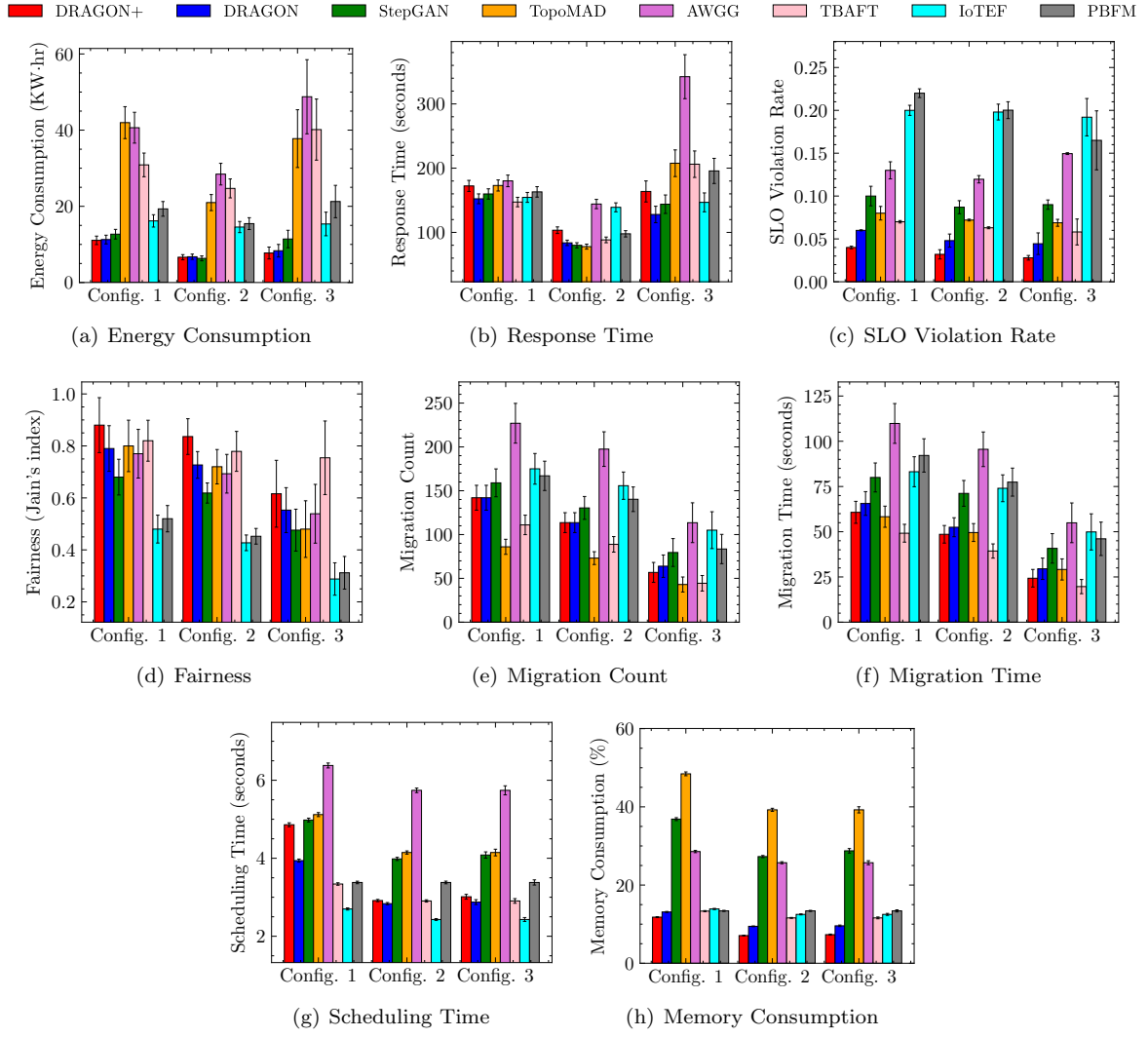


Figure 8.8: Comparison of QoS parameters of DRAGON against baselines. DRAGON and DRAGON+ trained on complete FTSAD-55 data and tested on AIoTBench based workloads.

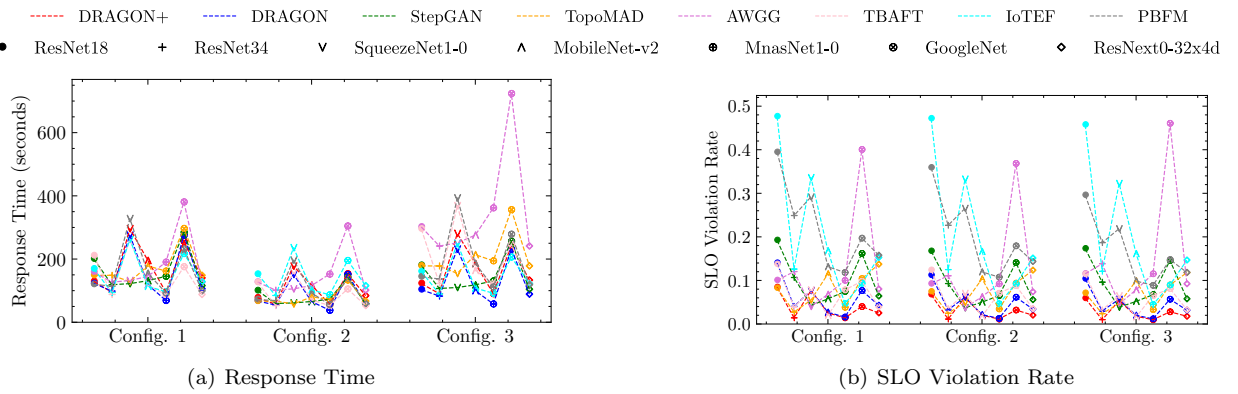


Figure 8.9: Comparison of QoS parameters of DRAGON against baselines for each application type in AIoTBench workloads.

except for AWGG that has a higher violation rate for the third configuration due to higher response times (Figs. 8.8(c) and 8.9(b)). Among all baselines, TBAFT has the lowest average SLO violation rate of 6.37%, where DRAGON and DRAGON+ improve upon this by 20.25% and 47.72%, giving violation rates of 5.08% and 3.33% respectively (Fig. 8.8(c)). In terms of fairness, DRAGON+ has the highest average fairness index of 0.78

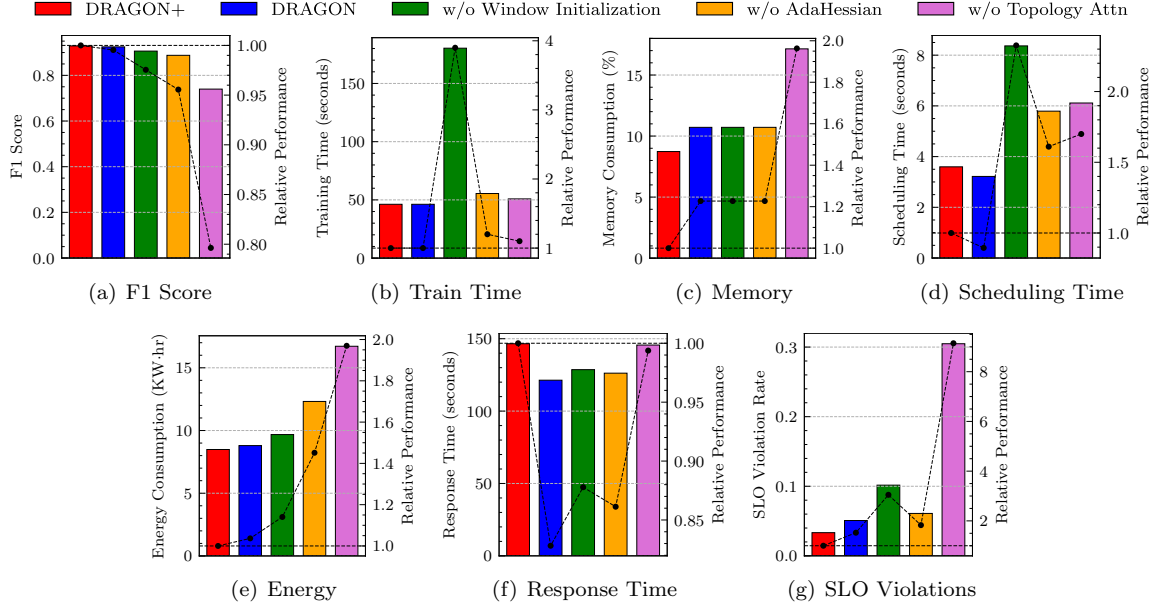


Figure 8.10: Ablation Analysis of different model components of DRAGON.

(Fig. 8.8(d)). The fairness drops from the first to third configurations due to more disparate LEI resources. In terms of migration count and migration time, AWGG is the most aggressive in terms of migrations (Figs. 8.8(e) and 8.8(f)). On the other hand, TopoMAD is the most parsimonious. DRAGON and DRAGON+ trade off between the fault-tolerance of migrations and their overheads. In terms of scheduling time, the heuristic solutions have the lowest times with IoTEF having only 2.52 seconds on an average. The deep learning methods have much higher scheduling times. However, for the second and third configurations, deep learning baselines have scheduling times consistent with the first configuration. On the other hand, DRAGON and DRAGON+ have lower scheduling times than all deep learning baselines. This is due to the dot-product attention in the GON network and the higher-order optimization, allowing the model to scale with the dimension size of the time-series window and the number of edge devices in an LEI (Fig. 8.8(g)). Memory consumption of the DRAGON and DRAGON+ models is the lowest, being only 10.71% and 8.72%. It is lower for DRAGON+ due to the lower parameter size GON network compared to DRAGON (see Fig. 8.8(h)).

The DRAGON+ method is not meant to substitute DRAGON, but rather offer a different tradeoff of latency and detection accuracy. Comparing DRAGON+ and DRAGON, we observe that the former has a higher average response time compared to the latter. This is due to the multi-step lookahead and optimization of the migration decision. However, the more informed decision choice in DRAGON+ leads to improvements in other metrics such as energy consumption, SLO violation rates and memory consumption. Although both methods beat state-of-the-art baselines, the above demonstrates that DRAGON is more suitable for latency critical decision making or when the broker nodes are resource constrained. However, for resource abundant brokers or when energy or SLO violation rates have higher importance, DRAGON+ is more appropriate.

8.5.6.3 Ablation Study

To study the relative importance of each component of the model, we exclude every major component one at a time and observe how it affects the performance of the scheduler. An overview of this ablation analysis is given in Fig 8.10. First, we consider the DRAGON+ and DRAGON models. We then consider DRAGON with

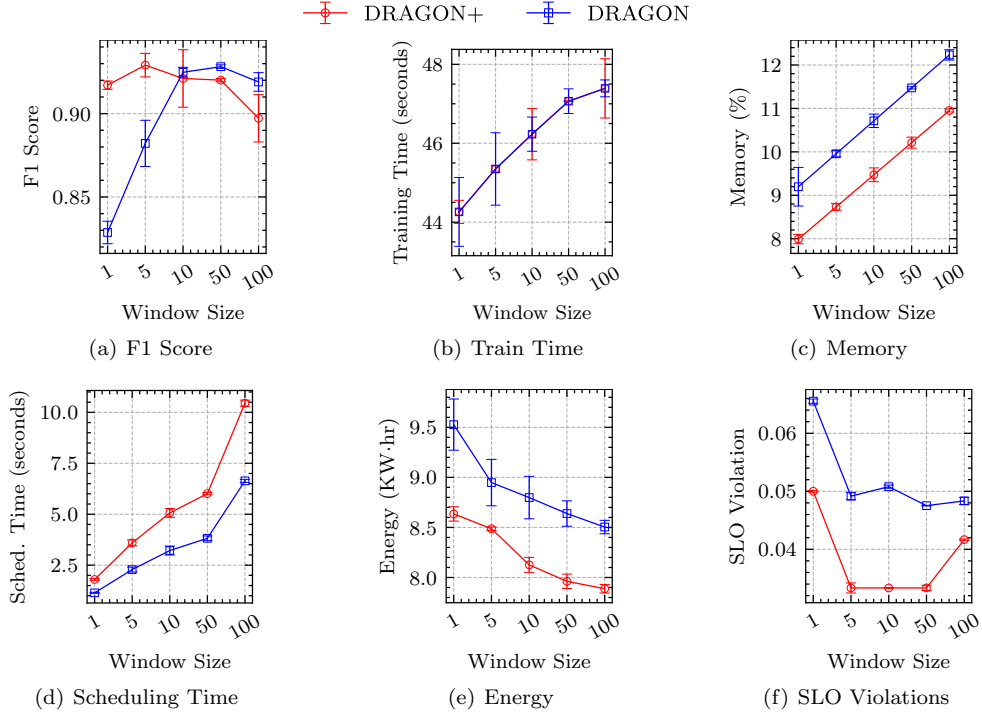


Figure 8.11: Sensitivity Analysis of performance of DRAGON and DRAGON+ with window size.

random noise initialization of Z instead of W (*w/o Window Initialization* model). Next, we consider a model with the **Adam** optimizer instead of **AdaHessian** (*w/o AdaHessian* model). Lastly, we consider the DRAGON model without the topology attention, using only W and S as inputs (*w/o Topology Attn* model). We report the following findings:

- Without the multi-step calculation of the fitness score in DRAGON+, the DRAGON model performs a myopic optimization and suffers an increase in the SLO violation rate, energy consumption and memory footprint.
- Without the window initialization, the generation stage significantly more time, increasing the scheduling and training time of the model.
- Without second-order optimization, the training and scheduling times are much higher.
- Without the topology attention, the SLO violation rate drastically increases as a model in a broker does not have any contextual information of the rest of the federated setup.

We observe that window initialization and topology attention have the maximum impact to the performance.

8.5.6.4 Sensitivity Analysis

Figs. 8.11 and 8.12 provide a sensitivity analysis of the performance of DRAGON and DRAGON+ models with window size and **AdaHessian** learning rate respectively. This sensitivity analysis is done to highlight the trade-off between the model performance and training time as we increase either of these parameters. As window size increases, the memory consumption, training time and scheduling time jointly increase due to larger input size of the GON network giving rise to higher time to generate next time-series window. However, higher window

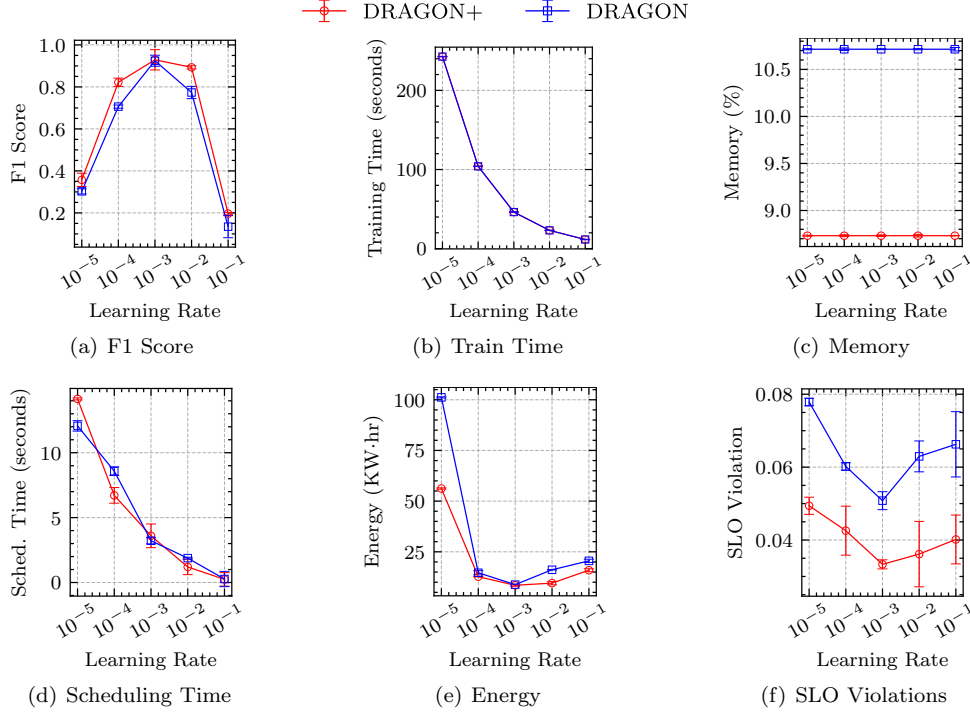


Figure 8.12: Sensitivity Analysis of performance of DRAGON and DRAGON+ with AdaHessian learning rate.

size also allows the model to get a broader context of the environment, reducing energy consumption and SLO violation rates. For DRAGON, the F1 score increases monotonically until a window size of 10, after which it remains mostly consistent. However, for DRAGON+, increasing the window size deteriorates the F1 score after 5. Intuitively, this is because DRAGON+ is able to get a broader temporal context due to multi-step iterations, and larger windows make model training unstable. Thus, we choose window size of 10 for DRAGON and 5 for DRAGON+ in our experiments.

The variation of performance metrics with the **AdaHessian** learning rate is more straightforward. Memory consumption is independent of the learning rate of the optimizer. However, training time and scheduling time of DRAGON and DRAGON+ decrease as we increase the learning rate. This is due to larger jumps in the optimization steps. However, for a high learning rate of $\geq 10^{-2}$, the model is unable to converge to the optima and hence we see the increase in energy consumption and SLO violation rates for these values. In terms of F1 score, it is highest for learning rate of 10^{-3} and hence has been used in our experiments.

8.5.7 Results for CAROL

We now present results comparing CAROL with baselines. For our experiments, we use $\alpha = \beta = 0.5$ in (8.26) as per prior work. Figure 8.13(a) compares the total energy consumption of all models and shows that CAROL is able to execute tasks with minimum energy consumption in the federated edge environment. The presence of the average energy consumption metric of all the edge nodes within the QoS score calculation as shown in (8.26) forces the model to take energy-efficient node-shift decisions. It results in the minimum number of active hosts with the remaining hosts in standby mode to conserve energy. CAROL reduces energy consumption by 16.45% compared to StepGAN, the model having minimum energy consumption across all baseline methods.

Figure 8.13(b) shows the average response time per task in seconds. The response time is measured as

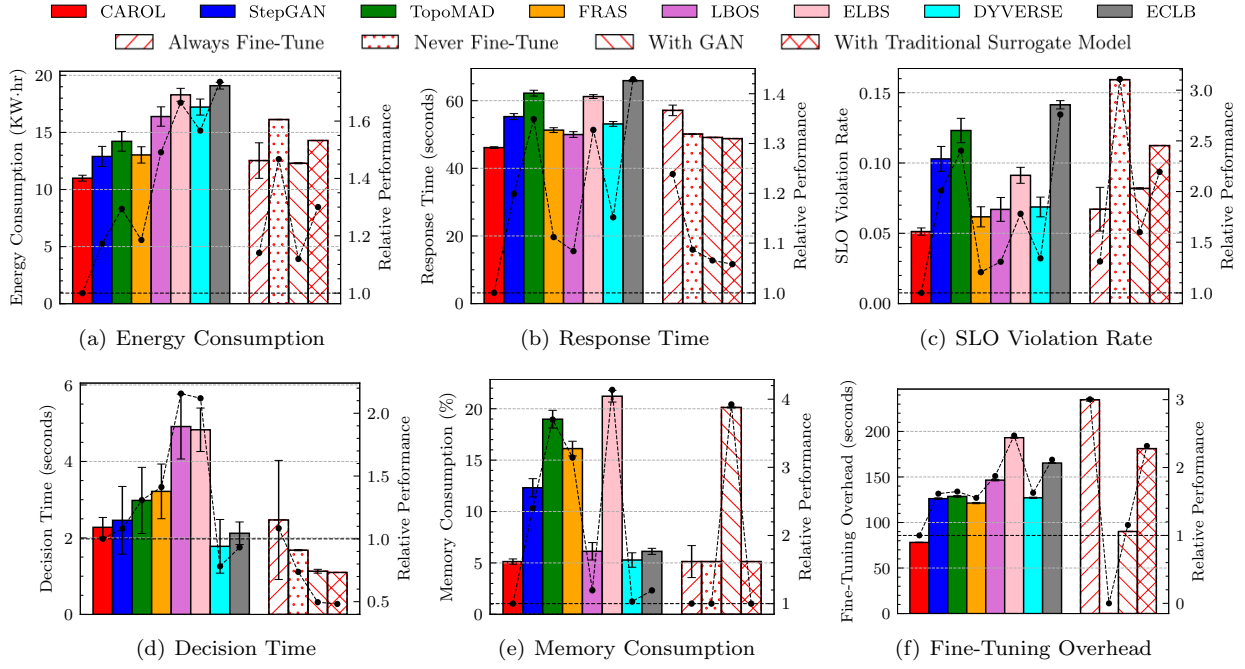


Figure 8.13: Comparison of CAROL with baseline methods and ablated models.

the time difference between the timestamps of task creation and result generation and is a crucial metric to compare the service throughput offered by the edge federation. The figure demonstrates that CAROL reduces this metric by 8.04% compared to the best baseline FRAS.

Figure 8.13(c) shows the fraction of SLA violations out of the total completed tasks in the 100 scheduling intervals. The figure shows that CAROL has the lowest SLO violation rate of only 5.12%, 17.01% lower than the least violation rates among the baselines, *i.e.*, 6.17% of the FRAS method. Allowing node-shift at each interval with the SLO violation rate in the QoS score calculation ensures that the topology minimizes response time and subsequently the SLO violation rates on an average. Moreover, CAROL is given the SLO deadline for each task that it utilizes to decide the number of worker nodes to allocate per task to minimize the SLA violation metric in the QoS score as shown in (8.26).

Figure 8.13(d) shows the comparison of average time to decide the fault-tolerance steps for each model. The figure shows that CAROL has a lower decision time compared to all AI-based methods. LBOS and ELBS have the highest decision time due to the time-consuming weighted round-robin and match-making algorithms used in these methods. The least decision time is of the DYVERSE heuristic method. However, CAROL has only 6.77% higher decision time than DYVERSE.

Figure 8.13(e) shows the average percentage memory utilization of the fault-tolerance models. The surrogate and generative methods have high memory consumption. ELBS has the highest memory consumption due to the resource intensive fuzzy neural networks in this approach. The RL based method, LBOS, has the lowest memory footprint among the AI baselines due to a lightweight Q-table used in the Q learning model in LBOS. Compared to the heuristic methods, CAROL has memory consumption ($\sim 5\%$), thanks to the memory-efficient GON model.

Figure 8.13(f) shows the overheads for fine-tuning the AI models and dynamically updating the priority scores in the heuristic models. The overhead is measured for the complete experiment, *i.e.*, over 100 scheduling intervals. The figure shows that CAROL has the lowest overhead of 78.12 seconds, only 0.78 seconds on an

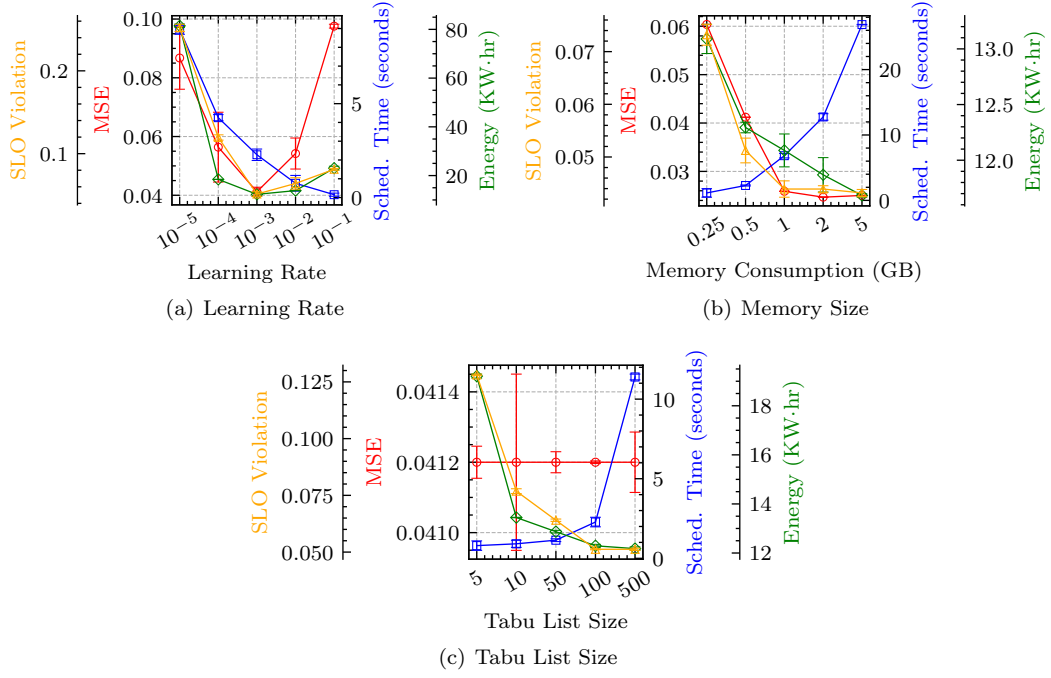


Figure 8.14: Sensitivity Analysis.

average per five-minute interval. This is 35.62% lower than the baseline with the lowest overhead, *i.e.*, FRAS that takes 121.35 seconds to update its model periodically. The significantly lower overhead of CAROL is due to the economic confidence-based model fine-tuning.

Overall, we observe that the ability to approximate the QoS scores for a given graph of the system topology and scheduling decision enables the GON-based generative model to predict the confidence scores and accurately predict QoS scores for any given state of the system. This, with the node-shifting techniques in CAROL, allows us to optimize the QoS of the federated environment.

8.5.7.1 Ablation Analysis

To study the relative importance of each component of the model, we exclude every major component one at a time and observe how it affects the performance of the scheduler. An overview of this ablation analysis is given in Fig 8.13 as hatched bars. First, we consider the CAROL model without the confidence scores, fine-tuning the GON network at every scheduling interval (*Always Fine-Tune* model). We also consider CAROL without fine-tuning the GON network in any interval (*Never Fine-Tune* model). Next, we consider a model with a traditional GAN network instead of the GON based confidence and QoS prediction (*With GAN* model). Lastly, we consider the CAROL model with a traditional feed-forward surrogate network instead of GON (*With Traditional Surrogate* model). We report the following findings:

- Without the confidence-aware model fine-tuning, the overheads significantly increase, causing a higher average decision time and response time of applications.
- Without ever fine-tuning the model, it does not adapt in non-stationary settings giving rise to poor QoS scores.
- Without the GON model, and using a GAN instead, the decision time is reduced as we do not need to

optimize in the topology space. However, the memory consumption increases from 5% to 30%, which impacts the running applications and broker management. This causes the SLO violation rate to increase by 6%.

- Without the GON model, and using a traditional feed-forward model, the decision time is reduced again, but at the cost of higher fine-tuning overheads.

We observe that the confidence-aware model fine-tuning and using the GON network have the maximum impact on performance values, accounting to nearly 70% of the total impact.

8.6 Summary

We have presented a decentralized fault-tolerance model, DRAGON, that utilizes fault predictions from GON and runs co-simulated fictitious play to converge to preemptive migration decisions. DRAGON leverages a novel memory-efficient generative model called generative optimization networks (GONs). Unlike previously proposed GAN models, GON uses a single neural network to both discriminate and generate samples. This comes with the advantage of significant savings in terms of memory footprint. Unlike prior work aiming at reducing the parameter size of popular generative models such as GANs, this framework is suitable for memory-constrained systems like edge devices. We extend DRAGON to DRAGON+ where we perform a multi-step optimization of a fitness score. The proposed models give up to 4.56% higher F1 scores compared to the best baseline StepGAN. In a federated Raspberry Pi based edge setup with real-world AIoTBench workloads, DRAGON and DRAGON+ are able to outperform baselines by giving up to 74%, 63% and 82% better energy consumption, response time and service deadline violations, respectively. These improvement are primarily due to advances like window initialization, second-order optimization and graph topology based attention.

We also present a novel method for resilient computing in edge federations. Our method uses a lightweight generative network as a surrogate model to efficiently and precisely map performance metrics like energy consumption and SLO violation rates for a given graph topology and scheduling decision. In the event of a broker failure, CAROL reactively optimizes the graph topology to accommodate the orphaned worker nodes. The topology is chosen by running a tabu search to optimize the QoS scores predicted by the surrogate model. CAROL uses the discriminator output as a confidence score that allows us to run model fine-tuning steps only when confidence scores drop below running thresholds. This parsimonious fine-tuning gives rise to 35.6% lower overheads compared to the current state-of-the-art. Performance evaluation of a real edge test-bed with AI and IoT based benchmark applications show that the low overheads in CAROL can improve energy consumption and SLO violation rates by 16.5% and 17.0% compared to the state-of-the-art.

Both models allow us to create a fog environment that is resilient to faults, achieving objective 3(b) as described in Chapter 1. We build on the ideas of co-simulation developed in Chapter 3 and develop novel generative neural models to build state-of-the-art service resilience.

Chapter 9

AI Augmented Concurrent Resource Management

In the previous chapters we discuss various ways in which we can leverage AI and Co-Simulation to augment resource management in fog environments. As part of this chapter, we explore the possibility of concurrent design of resource management decisions. Specifically, we focus on the aspects of provisioning, deployment and placement to synergistically leverage the dependencies across these decisions.

Background. Concurrent design, also known as co-design in literature, refers to the simultaneous optimization of system level design decisions [Werner and Tai, 2021, Hao et al., 2021, Gu et al., 2020]. Thus, a co-design based decision making process in the context of cloud computing needs to optimize the provisioning, deployment and placement decisions in tandem to achieve high QoS. Co-design is crucial to exploit the synergism of cloud hardware with AI software and build high-performance cloud systems [Ji et al., 2021]. Most prior work solves the QoS optimization problem for each decision type independently by using either forecasting models such as ARIMA and LSTM to predict future system states [Singh et al., 2019, Ouhamme et al., 2021] or deep neural networks (DNNs) as surrogate models for the system QoS and decision optimization (see Chpaters 3 and 6). These methods typically use several execution traces as datasets, with resource utilization and other system characteristics, to train AI models. Other co-design methods for cloud [Ji et al., 2021, Mladenov et al., 2020] do not explicitly capture the correlations across multiple data sources, like energy consumption and CPU utilization, adversely impacting performance (empirically shown in Section 9.3). However, this chapter proposes to perform co-design for resource management of cloud computing environments using a co-simulator [Ahlgren et al., 2020, Mladenov et al., 2020] as they enable us to explicitly capture the intrinsic dependencies of the system.

Challenges. Using a co-simulator to optimize QoS in typical cloud environments is challenging. Prior work that leverages a co-simulator [Mladenov et al., 2020], uses it only for augmenting the data being fed to the training process of the AI model and not for decision optimization. The main reason for this is that contemporary cloud services, like serverless, need quick decision making whereas executing several co-simulations to optimize decisions is a time-consuming activity. Recently, cloud service providers have shifted to the serverless execution model wherein cloud resources are provisioned and applications are executed on-demand, improving scalability and reducing execution costs [Javadi et al., 2020]. We focus on the use-case of co-design in serverless cloud environments throughout this chapter. In serverless, the applications use precisely the amount of resources

needed at any point in time and charge accordingly, making the costs proportional to the exact resource usage [Hendrickson et al., 2016]. For non-stationary workloads with dynamically changing characteristics, this entails frequently making decisions for provisioning, deployment and placement in a cloud environment, typically every few seconds. However, the size of the decision space scales with the number of workloads and cloud VMs in the system. Thus, for large-scale platforms, running a co-simulator for all possible decisions to estimate their impact on system QoS might take several minutes. Other strategies, such as evolutionary search [Aliyu et al., 2020] are unable to cover an ample search space due to time-constraints. For online decision making systems, this makes it challenging to adopt co-simulation for decision making mechanisms.

Research Contributions. To tackle the above mentioned issue of utilizing co-simulators for decision optimization, we propose a novel imitation learning model that aims to imitate an offline co-simulated oracle. We term this model as Serverless Co-design Network (SciNet). Our imitation learner is a composite neural network that predicts optimal provisioning, deployment and placement decisions by imitating an oracle, which for a given system state, estimates the impact on QoS scores using a co-simulated trace for each decision and outputs the optimal ones. As all cloud platforms have a finite number of workloads and VMs, such an oracle is able to generate decisions that are close to optimal by brute force, albeit rather slowly. An imitation learner that perfectly imitates such an oracle would also give us optimal decisions without the excessive overheads of executing several simulations at test time. The low inference time of neural networks enables SciNet to perform online decision making that efficiently scales with the size of the cloud infrastructure while harnessing the benefits of co-simulators. A preliminary version of the imitation learning model was presented in Chapter 4. This chapter presents a substantially detailed exposition of that work and includes additional supporting background material. Moreover, it also extends the previous approach to concurrently optimize provisioning, deployment and placement decisions, instead of just provisioning, as in the original chapter. This chapter also presents the tradeoffs dealt with each decision type and their impact on each other, to build the SciNet model and establish its superior performance against baselines. To test the efficacy of SciNet, we execute real-world serverless application workloads on a large-scale (50 VMs) public cloud infrastructure. Experiments show that SciNet outperforms the state-of-the-art baseline approaches by improving QoS metrics such as execution cost, inference accuracy, energy consumption and response times by up to 48%, 79%, 71% and 56% respectively.

9.1 System Model and Problem Formulation

We use a system model akin to that used in Chapter 6.

Workload Model. We consider a bag-of-tasks workload model, where a set of new workloads enter the system at the start of each interval. The set of new workloads generated at the start of the interval I_t is denoted by N_t . Each new workload $n_t^j \in N_t$ belongs to an application class, denoted by $A_t^j \in \mathcal{A}$, where $\mathcal{A}$ denotes the set of applications supported by the cloud computing environment, where A_t^j is represented as a one-hot vector of size $|\mathcal{A}|$. The application set $\mathcal{A}$ can range from object detection in images, natural language processing for a text or speech input, and other such deep learning based services. In the form of a DNN model, each application can be deployed in multiple ways, such as layer splitting, semantic splitting, and model compression. The set of these three deployment choices is denoted by $\mathcal{D}$. As described in Chapter 5, layer and semantic splitting techniques split a DNN model into multiple fragments that can be processed in a distributed fashion. On

the other hand, model compression primarily uses techniques to prune the parameters of a DNN to reduce its resource utilization without drastically affecting its inference performance. The deployment decision for each incoming workload $n_t^j \in N_t$ is denoted by D_t^j . This decision deploys the DNN model corresponding to A_t^j as per the splitting or compression strategy, *i.e.*, D_t^j and leads to a set of tasks denoted by $\{n_t^j\}_{D_t^j}$. Thus, the new tasks created from the incoming workloads N_t are denoted by $\bigcup_k \{n_t^j\}_{D_t^j}$. Tasks in our setup are realized as serverless function calls. We denote the set of all active tasks in the interval I_t , which also includes incomplete tasks from I_{t-1} by W_t , where W_0 is $\emptyset$. Each task $w_t^k \in \{n_t^j\}_{D_t^j} \subseteq W_t$ is characterized by the application class A_t^j and deployment decision D_t^j of its workload, with a one-hot vector denoted by O_t^k corresponding to the index of w_t^k in $\{n_t^j\}_{D_t^j}$. This one-hot vector corresponds to, for instance, the layer-group index in a layer splitting deployment decision. The feature vector of task w_t^k includes the workload information with CPU utilization in terms of the number of instructions per second (IPS), denoted by c_t^k ; RAM utilization in GBs, denoted by r_t^k ; and disk storage utilization in GBs, denoted by s_t^k . Here, $k \in \{1, \dots, |W_t|\}$. Thus, the feature vector for task w_t^k of workload n_t^j is denoted by $W_t^k = [c_t^k, r_t^k, s_t^k]$. The collection of feature vectors of all active tasks in I_t is denoted by W_t .

Edge/Cloud Hosts. At I_t , the set of all VM hosts that can be provisioned is denoted by $\mathcal{H}_t$. In I_t , we consider a placement decision for each active task in W_t . The set of tasks placed onto host $h_t^i \in \mathcal{H}_t$ is denoted by $\mathcal{T}_t^i \subseteq W_t$, where $i \in \{1, \dots, |\mathcal{H}_t|\}$. Here, $\mathcal{T}_t^i = \emptyset$ for all deallocated hosts $h_t^i \in \mathcal{H}_t$. For each host $h_t^i \in \mathcal{H}_t$, we create a feature vector that includes cumulative utilization and maximum capacity of resources (CPU, RAM and disk). We characterize the CPU utilization capacity in terms of maximum IPS, denoted by $\bar{c}_t^i$; RAM capacity in GBs, denoted by $\bar{r}_t^i$; and disk storage capacity in GBs, denoted by $\bar{s}_t^i$. Here, $i \in \{1, \dots, |\mathcal{H}_t|\}$. We denoted the feature vector of host h_t^i by $H_t^i = [\sum_{w_t^j \in \mathcal{T}_t^i} W_t^j, \bar{c}_t^i, \bar{r}_t^i, \bar{s}_t^i]$, where $\sum$ denotes vector sum and $\bar{c}_t^i, \bar{r}_t^i, \bar{s}_t^i$ denote IPS, RAM and disk storage capacities of the host. As no tasks are placed on deallocated hosts, $H_t^i = [\vec{0}, \bar{c}_t^i, \bar{r}_t^i, \bar{s}_t^i]$ in such cases. Each host has a VM type that corresponds to a distinct set of utilization capacities and execution costs in a public cloud deployment. We consider a static set of VM types $\mathcal{V}$, where the type of host h_t^i is denoted by $V_t^i = [\mu^i, \xi^i]$. Here, μ^i is the cost per unit time and ξ^i is the distribution of the provisioning time of a VM instance for a VM type V_t^i . We assume μ^i and ξ^i to be stationary with time $\forall i$. As elements of V_t^i are independent of time, these symbols are not sub-scripted with t . V_t denotes VM types for all hosts in I_t , $\mathcal{H}_t$.

Provisioning Decisions. VM provisioning is performed at the start of each interval. The new set of provisioned hosts in I_t is the set of provisioned hosts in the previous interval union the newly provisioned hosts minus the deallocated hosts at the start of I_t . Deallocation of hosts in the system entails migrating running workloads in those hosts to other preexisting active or newly provisioned hosts in the system. Such tasks are referred to as *orphaned tasks* in the rest of the discussion. For each host $h \in \mathcal{H}_t$, at the start of each interval I_t , we decide whether it should be provisioned or not as a binary value (0 or 1) and denote the provisioning decision for h_t^i as L_t^i . The hosts with the decision as 1 form the provisioned hosts in the interval I_t . Each host h_t^i that is allocated in the interval I_t takes a time sampled from distribution ξ_t^i to come online. The collection of provisioning decisions at interval I_t for all hosts is denoted by L_t .

Deployment Decisions. Deployment decisions are made for each incoming workload N_t at the start of the interval I_t . These decisions belong to the set of deployment types $\mathcal{D}$ that includes layer splitting, semantic splitting and model compression. As mentioned previously, for each input workload $n_t^j \in N_t$, the deployment decision is denoted by D_t^j leading to the task set $\{n_t^j\}_{D_t^j}$. The deployment decision D_t^j is a categorical value

in the set $\{0, \dots, |\mathcal{D}| - 1\}$ where each value represents to the corresponding decision type in $\mathcal{D}$. The collection of all deployment decisions for incoming tasks at the start of the interval I_t is denoted by D_t . We use multiple deployment options as a more traditional deployment strategy of directly utilizing a DNN (similar to a layer splitting decision with all splits allocated to the same cloud host) limits the level of control over the inference characteristics such as result fidelity (*viz* accuracy of object detection like workloads) and response time. Multiple deployment options expose another control surface layer to optimize QoS parameters.

Placement Decisions. Task placement is performed for each new task created at the start of the interval as well as the tasks that are orphaned due to host deallocations. For each task $w_t^k \in W_t$, the placement decision is denoted by P_t^k that is a categorical value in the set $\{0, \dots, |\mathcal{H}_t| - 1\}$ representing the host on which the task w_t^k is to be placed. Thus, if the placement decision for w_t^k is i , then it is placed on host h_t^i . For such a decision to be valid, host h_t^i needs to be provisioned, *i.e.*, $L_t^i = 1$.

Imitation Learner. In our formulation, we use a neural network denoted by f_θ^{il} where we use θ to denote the parameters of such a network. Our neural network imitates a co-simulated oracle to predict the provisioning, deployment and placement decision at each interval I_t . To do this, we also use our neural model to predict the task utilization characteristics for I_t , *i.e.*, W_t . We denote this prediction by $\hat{W}_t$, for which the model may use historic data $\{W_x | 1 \leq x < t\}$. Our decision model, f_θ^{il} , uses the workload utilization metrics of the previous interval W_{t-1} to predict $\hat{W}_t$, provisioning decision L_t , deployment decisions D_t and placement decision P_t . For provisioning decisions L_t , the model utilizes the predicted task utilization characteristics $\hat{W}_t$, the VM types V_{t-1} and the placement decision in the previous interval P_{t-1} to estimate the feature vectors of the hosts in the next interval, denoted by $\hat{H}_t$. The model also utilizes the application types A_t^j for all incoming workloads $n_t^j \in N_t$ to predict the deployment decisions D_t . Based on the deployment decisions D_t , for each task $w_t^k \in W_t$, the model utilizes the deployment one-hot vectors O^k and application types A^k . Here with slight abuse of notation, we use A_t^k to denote the application type of the workload that w_t^k belongs to. This placement decision also includes all orphaned tasks at the start of I_t due to host deallocations. Thus, for interval I_t , we represent the inference of the imitation learner as

$$\hat{W}_t, L_t, D_t, P_t = f_\theta^{il}(W_{t-1}, V_{t-1}, P_{t-1}, \mathcal{H}_t, N_t, W_t). \quad (9.1)$$

Co-Simulated Digital Twin. We utilize a co-simulator to model the physical environment and estimate changes in the system behavior based on provisioning, deployment or placement decisions. Our co-simulator stores the time-series utilization characteristics W_t . It executes a simulation of the cloud platform for a given workload feature vectors W_{t-1} , VM types V_t , provisioning decision L_t , deployment decision D_t and placement decision P_t to generate a set QoS metrics for interval I_t that we denote by $\mathcal{Q}_t$. These metrics include energy consumption, response time of completed workloads, SLA violation rates, inference accuracy, execution cost and utilization ratio. It also provisions new VMs and deallocates existing ones as per L_t , deploys AI models as tasks as per D_t and places tasks to provisioned hosts as per P_t . We denote the co-simulator by f^{sim} and the set of QoS metrics as

$$\mathcal{Q}_t = f^{sim}(W_{t-1}, V_t, L_t, D_t, P_t, \mathcal{H}_t, N_t, W_t). \quad (9.2)$$

Problem Formulation. At the start of the interval I_t , given VM types V_t , set of input workloads N_t , active tasks W_t and history of the system states, we define our problem as to find a feasible provisioning decision

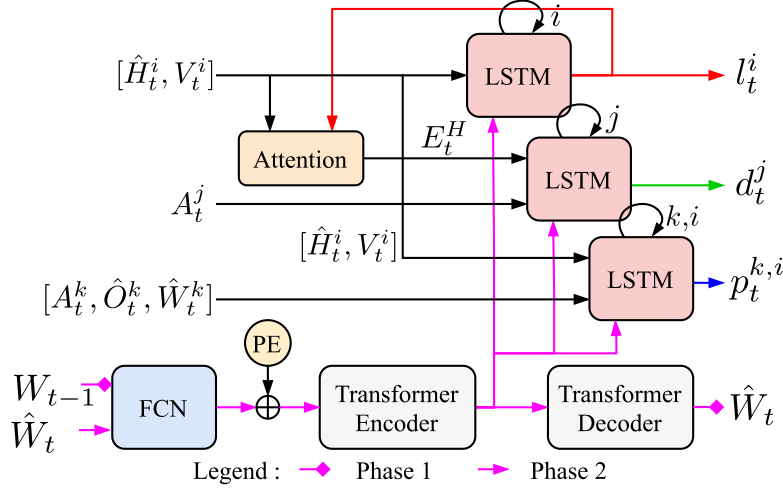


Figure 9.1: Neural network as imitation learning model in SciNet.

L_t , deployment decision D_t and placement decision P_t that maximize the QoS metrics of a cloud computing platform. Our optimization objective may be a convex combination of the QoS metrics of a hybrid cloud platform, including response time, energy consumption, etc., and we denote it as q_t for interval I_t . The formal optimization program is described as follows.

$$\begin{aligned}
& \max_{\theta} \quad \sum_{t=1}^T q_t \\
& \text{s.t.} \quad \forall t, \forall w_t^k \in W_t, \sum_{w_t^k \in \mathcal{T}_t^i} W_t^k \leq [\bar{c}^i, \bar{r}^i, \bar{s}^i] \\
& \quad \forall t, \forall w_t^k \in W_t, i = P_t^k, L_t^i = 1 \\
& \quad \forall t, \hat{W}_t, L_t, D_t, P_t = f_{\theta}^{il}(W_{t-1}, V_{t-1}, P_{t-1}, \mathcal{H}_t, N_t, W_t) \\
& \quad \forall t, Q_t = f^{sim}(W_{t-1}, V_t, P_t, D_t, I_t, \mathcal{H}_t, N_t, W_t)
\end{aligned} \tag{9.3}$$

for a given co-simulator f^{sim} and imitation learner f^{il} and $t = \{1, \dots, T\}$.

9.2 The SciNet Model

The SciNet model aims to learn to predict *a-priori* unknown workload characteristics of the next interval with optimal provisioning, deployment and placement decisions in cloud computing setup. It is realized as a neural network as an imitation learner f_{θ}^{il} .

9.2.1 Neural Model and Two-Phase Inference

For interval I_t , the inputs of the neural model are workload utilizations of the previous interval W_{t-1} , hosts $\mathcal{H}_t$, VM types V_t , placement decision for the previous interval P_{t-1} , new and active workloads N_t and W_t (see Fig. 9.1 for an overview). We run inference in two stages, where our model first estimates the resource utilization characteristics, *i.e.*, predicted feature vectors $\hat{W}_t$. In the second phase, the model uses $\hat{W}_t$ to predict the decisions L_t , D_t and P_t . To encode each hosts $h_t^i \in \mathcal{H}_t$ in the system for provisioning decision, we use the predicted workload demands $\hat{W}_t$ and use placement decision in the previous interval P_{t-1} to find the task

allocation set $\mathcal{T}_t^i$. Then, the estimates feature vectors of the hosts are $\mathcal{H}_t^i = [\sum_{w_t^j \in \mathcal{T}_t^i} \hat{W}_t^j, \bar{c}^i, \bar{r}^i, \bar{s}^i]$. Thus, the encoding of host h_t^i becomes the concatenation of the feature vector and VM type vector as $[\mathcal{H}_t^i, V_t^i]$. To encode the new workload $n_t^j \in N_t$ for deployment decisions, we use the application type one-hot vector A_t^j . For the placement decision, we encode the task $w_t^k \in W_t$ by concatenating the application type, deployment one-hot vector and predicted feature vector as $[A_t^k, O_t^k, \hat{W}_t^k]$ where $\hat{W}_t^k$ for tasks in new workloads N_t have $\hat{W}_t^k = \vec{0}$.

To predict $\hat{W}_t$, L_t , D_t and P_t , we use a composite neural model by inferring the utilization characteristics using a feed-forward network (also referred to as a fully connected network or FCN) and a Transformer to capture the temporal trends in the data. Unlike recurrent models, Transformers have lower inference times. We first pass the normalized input W_t of the previous interval through a feed-forward network with sigmoid activation such that

$$E_t^W = \sigma(\text{FeedForward}(W_t)). \quad (9.4)$$

We then pass the vector E_t^W through a Transformer encoder after adding positional encodings (PE) [Vaswani et al., 2017] using multi-head self-attention, giving an encoded representation,

$$E_t = \text{TransformerEncoder}(E_t^W). \quad (9.5)$$

Using E_t , we then predict an estimate of utilization characteristics of the next interval, *i.e.* $\hat{W}$, we use a Transformer decoder as

$$\hat{W} = \text{TransformerDecoder}(E_t). \quad (9.6)$$

We then use the predicted $\hat{W}_t$ in the second-phase of inference that predicts the decisions. Two-phase inference allows the second-phase to take a more informed decision in the sense of having predicted workload utilization of the coming interval I_t , instead of historic utilization characteristics. Thus, passing the predicted utilization $\hat{W}_t$ through the FCN and Transformer encoder, we get

$$\begin{aligned} \hat{E}_t^W &= \sigma(\text{FeedForward}(\hat{W}_t)), \\ \hat{E}_t &= \text{TransformerEncoder}(\hat{E}_t^W). \end{aligned} \quad (9.7)$$

To generate provisioning, deployment and placement decisions, we independently predict the likelihood score for each decision, which we refer to as the likelihood vector, using Long-Short-Term-Memory (LSTM) networks. Unlike other recurrent models, LSTMs can handle long-range dependencies and hence are more appropriate for large-scale setups. To allow the model to scale with $|\mathcal{H}_t|$, for each active host $h_t^i \in \mathcal{H}_t$, we use the encoding $[\hat{H}_t^i, V_t^i]$ to infer on which host should be deallocated or provisioned. The likelihood vector l_i is predicted of size 2, which denotes the likelihood scores of provisioning and deallocating host h_t^i in I_t . We recurrently perform this prediction using an LSTM, such that we can make the provisioning decision for a host conditioned on the decisions for other hosts. Thus,

$$l_t^i = \text{LSTM}_i(\hat{E}_t, \hat{H}_t^i, V_t^i), \quad (9.8)$$

where LSTM_i denotes that the LSTM is applied in recurrence with index i . This factored-style prediction of the likelihood score for each provisioning action enables our model to be agnostic to the number of hosts in the

setup. The final provisioning action, L_t^i , becomes the action corresponding to the highest likelihood score, *i.e.*,

$$L_t^i = \arg \max l_t^i, \quad (9.9)$$

where the $\arg \max$ operation is along the vector l_t^i .

To generate the likelihood vectors for deployment decisions, we want the likelihood scores to be conditioned on the provisioning decisions. To ensure that the encoding passed to the LSTM that generates deployment likelihood vectors is of a static size, we use Bahdanau style attention [Bahdanau et al., 2015] such that we generate an attention weight for each host and generate a convex combination. To do this we use the `gumbel_softmax` activation function [Jang et al., 2017], which is a variation of softmax that allows us to generate a one-hot vector while also allowing gradients to backpropagate (as $\arg \max$ is not differentiable). It uses a temperature parameter τ that allows to control how close the output is to one-hot versus the softmax output. Typically, we keep a positive value of τ at training time and gradually anneal it a value close to 0 to ensure a one-hot output. Thus, we form the embedding E_t^H

$$\begin{aligned} G_t^i &= \text{gumbel_softmax}(l_t^i, \tau), \\ \text{attn}^i &= \sigma(\text{FeedForward}(\hat{H}_t^i, V_t^i, G_t^i)), \\ E_t^H &= \sum_i \text{attn}^i \cdot [\hat{H}_t^i, V_t^i], \end{aligned} \quad (9.10)$$

Using this, we generate the deployment decision for each input workload $n_t^j \in N_t$ as d_t^j , which is a likelihood vector of size $\mathcal{D}$, and D_t^j is the decision corresponding to the one with highest likelihood score. Thus,

$$\begin{aligned} d_t^j &= \text{LSTM}_j(\hat{E}_t, E_t^H, A_t^j), \\ D_t^j &= \arg \max d_t^j. \end{aligned} \quad (9.11)$$

To generate likelihood vectors for the placement decision, we want the likelihood scores to be conditioned both on the host encodings and the deployment decisions. To do this, for each task $w_t^k \in W_t$ and host $h_t^i \in \mathcal{H}_t$, we predict a likelihood score of placing task w_t^k on host h_t^i . We also use the application type A_t^k , predicted demands $\hat{W}_t^k$ and one-hot vector corresponding to the deployment decision. Again, to ensure that we are able to propagate gradients to the upstream neural models, we use `gumbel_softmax` to generate $\hat{O}_t^k$ to approximate $\hat{O}_t^k$. Thus, our placement likelihood vectors and consequently the placement decisions are generated as

$$\begin{aligned} \hat{O}_t^k &= \text{gumbel_softmax}(d_t^i, \tau), \\ p_t^{k,i} &= \text{LSTM}_{k,i}(\hat{E}_t, \hat{H}_t^i, V_t^i, A_t^k, \hat{O}_t^k, \hat{W}_t^k), \\ P_t^k &= \arg \max_i p_t^{k,i}, \end{aligned} \quad (9.12)$$

where the LSTM is first applied in a recurrence for each host (loop across i) then for each task (loop across k) and $\arg \max$ is across i for scalars $p_t^{k,i}$. This auto-regressive style of predicting decision of provisioning, deployment and placement in this order is crucial for optimal decision making. Intuitively, the placement decisions depends on the provisioned hosts and deployment types used for incoming workloads. Thus, forwarding the decisions using `gumbel_softmax` allows these decisions to be utilized by the downstream predictors.

We thus have a neural model that takes in the inputs $W_{t-1}, V_{t-1}, P_{t-1}, \mathcal{H}_t, N_t$, and W_t to generate provisioning decisions for each host h_t^i as L_t^i , deployment decision for each incoming workload n_t^j as D_t^j and the placement decision for each task w_t^k as P_t^k . Collecting the decisions for each decision type gives us L_t , D_t and P_t for interval I_t . This operation is denoted by the neural model f_θ^{il} in eq. (9.1).

9.2.2 Co-Simulated Oracle

Now that we have an imitation learner f_θ^{il} that given the state history of a cloud system can generate provisioning, deployment and placement decisions, we need a teacher that f_θ^{il} can imitate. To develop such a decision model, we utilize a co-simulator. Our co-simulator consists of several expert-encoded relationships across of data sources [Mahmud et al., 2021, Calheiros et al., 2011]. For instance, our co-simulator consists of the functions that give the power consumption of a cloud VM given its CPU utilization. Examples include power models from the SPEC benchmark power profile repository that give the power consumption in watts of several computing systems given their CPU utilization percentage.¹ Using such information, with the provisioning, deployment and placement decisions, allows co-simulators to estimate power consumption in a future state of the system, subsequently giving the energy consumption in interval I_t at the start of I_t . In our formulation, we use a co-simulator to generate QoS metrics directly as shown in Eq. 9.2.

Considering, we have a method to combine the various estimates of QoS $\mathcal{Q}_t$ for interval I_t , say $\Gamma(\mathcal{Q}_t)$, we can generate the decisions that maximize this metric. Given host set $\mathcal{H}_t$, workload set N_t and task set W_t , we denote the set of all feasible provisioning, deployment and placement decisions by $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$. Thus, for a convex combination $\Gamma(\mathcal{Q}_t)$, we define oracle decisions as

$$L_t^*, D_t^*, P_t^* = \arg \max_{L_t \in \mathcal{X}, D_t \in \mathcal{Y}, P_t \in \mathcal{Z}} \Gamma(f^{sim}(W_{t-1}, V_t, L_t, D_t, P_t, I_t, \mathcal{H}_t, N_t, W_t)). \quad (9.13)$$

Thus, the decision tuple of L_t^*, D_t^*, P_t^* is then considered as the oracle decision set using the simulator f^{sim} .

9.2.3 Training Recipe

The above co-simulated oracle executes the co-simulator for $|\mathcal{X}| \times |\mathcal{Y}| \times |\mathcal{Z}| = 2^{|\mathcal{H}_t|} \cdot |\mathcal{D}|^{|\mathcal{N}_t|} \cdot |\mathcal{H}_t|^{|\mathcal{W}_t|}$ times to generate oracle decisions for interval I_t . To train the imitation learner f^{il} , we need to generate oracle decisions for several execution intervals. Generating oracle decisions in this fashion makes the training intractable for large-scale cloud systems with $|\mathcal{W}_t|$ and $|\mathcal{H}_t|$ as high as 500 and 100 respectively.

To resolve this issue, we generated a limited amount of data corresponding to the oracle decisions using the co-simulator, denoted by Λ^* . The majority of our data of oracle decisions are generated by optimizing each decision type independently, denoted by Λ . This gives us $|\mathcal{X}| + |\mathcal{Y}| + |\mathcal{Z}|$ simulator runs, albeit not giving the optimal decisions. However, optimizing each decision independently requires different Γ functions. For instance, we do not have deployment or placement decisions when optimizing provisioning. Thus, we utilize metrics agnostic to these decisions.

Optimizing Provisioning Decision. To find L_t^* in Λ without the deployment and placement decisions, we utilize the QoS metrics of utilization ratio (denoted by r_t) and cost (denoted by ϕ_t). These two metrics are the

¹SPEC power models: https://www.spec.org/power_ss2008/results/res2021q2/power_ss2008-20210528-01098.html

most affected by the provisioning decisions. We define these two metrics as,

$$\begin{aligned} r_t &= \frac{\sum_{w_t^k \in W_t} c_t^k}{\sum_{h_t^i \in \mathcal{H}_t} \bar{c}_t^i}, \\ \phi_t &= \sum_{h_t^i \in \mathcal{H}_t} \mu^i \cdot \Delta. \end{aligned} \quad (9.14)$$

Here, $\bar{c}_t^i$ is generated from co-simulations. These metrics also appear in prior work [Luo et al., 2021, Gu et al., 2015] and are easy to compute in our formulation. For provisioning, we combine the above two metrics using a convex combination as

$$\Gamma^L(\mathcal{Q}_t) = r_t - \gamma \cdot \phi_t, \quad (9.15)$$

where γ is a hyperparameter. We thus generate L_t^* as the provisioning decision that maximizes the above metric.

Optimizing Deployment Decision. When having an optimal provisioning decision L_t^* , to find D_t^* in Λ without the placement decisions, we utilize the QoS metrics of average inference accuracy (denoted by a_t) and average SLA violation rate (denoted by ρ_t). These two metrics are the most affected by the deployment decisions as different deployment types like layer and semantic splitting have distinct response time and accuracy characteristics. Here, a_t and ρ_t are generated from co-simulations using the random placement of deployed tasks onto the provisioned hosts as per L_t^* . These metrics also appear in prior work [Tuli, 2021, Yu et al., 2021]. For deployment, we use the QoS metric

$$\Gamma^D(\mathcal{Q}_t) = a_t - \xi \cdot \rho_t, \quad (9.16)$$

where ξ is a hyperparameter. We thus generate D_t^* as the deployment decision that maximizes the above metric.

Optimizing Placement Decision. When having an optimal provisioning and deployment decisions L_t^*, D_t^* , to find P_t^* in Λ , we utilize the QoS metrics of energy consumption (denoted by e_t) and average task response time (denoted by κ_t). These two metrics are the most affected by the placement decisions as different task placements types give distinct CPU utilization characteristics and hence the energy consumption and task response times based on IPS capacity sharing. Here, e_t and κ_t are generated from co-simulations. These metrics also appear in prior work [Basu et al., 2019]. For placement, we use the QoS metric

$$\Gamma^P(\mathcal{Q}_t) = e_t - \zeta \cdot \kappa_t, \quad (9.17)$$

where ζ is a hyperparameter. We thus generate P_t^* as the placement decision that maximizes the above metric. We use grid search to find the values of the hyperparameters γ , ξ and ζ .

To generate our training datasets Λ and Λ^* , we use random decision making module to cover as much state space as possible. Each dataset consists of input output pairs of the imitation learning model f_θ^{il} , i.e., $\{(W_{t-1}, V_{t-1}, P_{t-1}, \mathcal{H}_t, N_t, W_t), (L_t^*, D_t^*, P_t^*, W_t)\}_t$, where the oracle decision making strategy varies between Λ and Λ^* . However, to ensure that the f_θ^{il} is tuned to predict decisions corresponding to Λ^* , in each training epoch, we use $\alpha \in [0, 1]$ ratio of Λ , where α is annealed from 1 to 0.

Teacher Forcing. Our formulation assumes that the deployment decision module gets optimal provisioning decision as an input and placement decision modules get optimal provisioning and deployment decisions. To

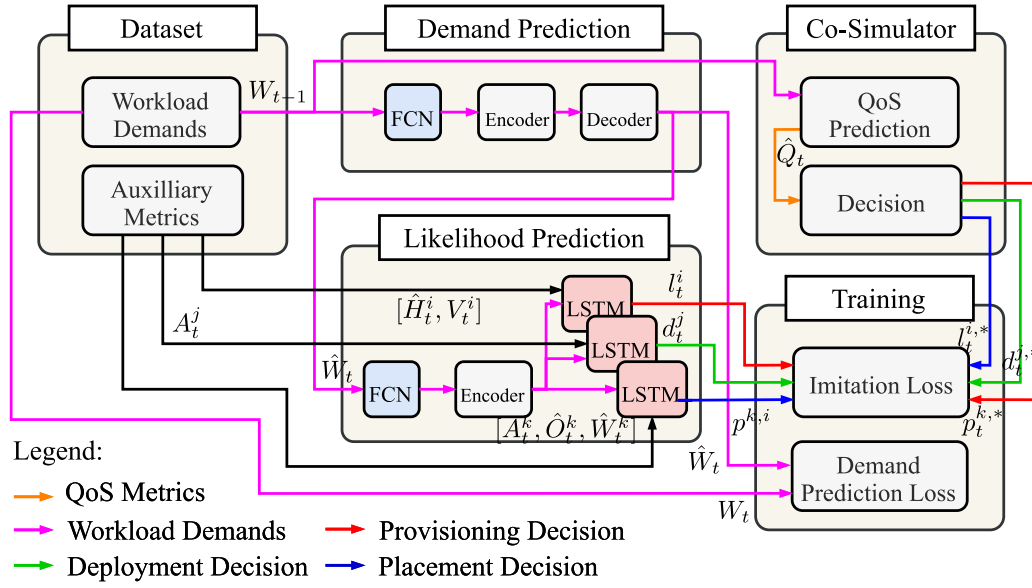


Figure 9.2: Imitation Learning pipeline.

resolve this, we utilize teacher forcing, wherein the input provisioning decision for the deployment decision LSTM is L_t^* instead of L_t . Similarly, we use L_t^* and D_t^* for the placement decision LSTM. Avoiding exposure bias gives us another reason to utilize gumbel_softmax to ensure that the exposure is consistent with the teacher-forced training data. However, at test time, we do not have oracle decisions. Thus, we use oracle decisions as inputs with β probability. Assuming that in the initial training phase, the decisions are sub-optimal, which get close to optimal in later training phases, we anneal β with epochs. Here, teacher forcing always gives one-hot decision vectors.

Offline Training Procedure. Our data collection and training process is interleaved, *i.e.*, as soon as we collect a datapoint, we use it to update the parameters of the neural network f_θ^{il} (see Fig. 9.2). Thus, for an input workload demand W_{t-1} and system characteristics, we generate oracle decisions L_t^*, D_t^*, P_t^* using independent optimization with probability α and joint optimization with $1 - \alpha$. The corresponding datapoint is added to Λ or Λ^* . We also generate W_t for training using the co-simulator and decisions from the imitation learner using Eq. (9.1). We define the *Demand Prediction Loss*, denoted by L_{DP} , as the mean-square error between the predicted and true task demands

$$L_{DP} = \frac{1}{|W_t|} \|\hat{W}_t - W_t\|^2. \quad (9.18)$$

We also define the *Imitation Loss* as the binary-cross-entropy loss between the predicted and oracle decision vectors. We generate oracle decision vectors $l_t^{i,*}, d_t^{j,*}, p_t^{k,*}$ as one-hot vectors representing the oracle decisions.

$$\begin{aligned} L_I = & \sum_i \text{BCE}(\text{softmax}(l_t^i), l_t^{i,*}) \\ & + \sum_j \text{BCE}(\text{softmax}(d_t^j), d_t^{j,*}) \\ & + \sum_k \text{BCE}(\text{softmax}(p_t^k), p_t^{k,*}), \end{aligned} \quad (9.19)$$

where $\text{BCE}(x, y)$ represents the binary-cross-entropy loss between the likelihood vectors x and y . We then backpropagate the gradient of loss $L_{DP} + L_I$ to update the parameters θ of the network f_θ^{il} . To advance to the

Algorithm 9.1 SciNet Model at test time**Require:**

-
- Pretrained model f_{θ}^{il} .
 - Initialize $W_0 \leftarrow \vec{0}$
 - 1: **for** $t \in \{1, \dots, T\}$
 - 2: $\hat{W}_t, L_t, D_t, P_t \leftarrow f_{\theta}^{il}(W_{t-1}, V_{t-1}, P_{t-1}, \mathcal{H}_t, N_t, W_t)$
 - 3: Execute L_t to provision hosts
 - 4: Execute D_t to deploy workloads N_t
 - 5: Execute P_t to place tasks on hosts
-

next execution interval, we use random provisioning, deployment and placement decisions.

We use exponential decay for the temperature parameter τ , teacher forcing probability α and joint optimization probability β using the decay rate of 0.99 after each execution interval. We use $T = 500$ for training. We use Adam optimizer [Kingma and Ba, 2014] with learning rate of 5×10^{-4} and weight decay parameter of 10^{-5} . To train the neural model, we randomly divide the $\Lambda^* \cup \Lambda$ dataset into 80% training data and 20% validation data. An early stopping criterion was applied for convergence.

Online Decision Making. Using a pre-trained SciNet model for online decision making is illustrated in Alg. 9.1. In each interval, we first predict the workload demands in the next interval and generate the provisioning, deployment and placement decisions (line 2). The final decisions are then executed in the cloud environment (lines 3-5) such that the hosts are provisioned as per the binary provisioning output, the splitting or compression decisions are as per the likelihood scores for each decision type and placement of new and orphaned tasks as per the placement score for each VM in the cloud environment.

9.3 Performance Evaluation

9.3.1 Setup

Workloads. In order to evaluate the performance of SciNet, we use the AIOtBench benchmarks as used in Chapter 8.

Baselines. We compare SciNet against 11 baselines. We integrate the ACO algorithm with two demand forecasting methods: AutoARIMA and LSTM, and call these ARIMA+ACO and LSTM+ACO. We also include other predict+optimize methods Decision-NN, Semi-Direct and GRAF. We also include combinations of state-of-the-art methods for each decision type. This includes UAHS, CAHS and Narya for provisioning, Gillis and SplitPlace of deployment and SplitPlace and GOSH for placement. This forms the set of baselines: UAHS+Gillis+GOSH, CAHS+Gillis+GOSH and Narya+SplitPlace. We do not include UAHS and CAHS with SplitPlace as SplitPlace needs response time estimates for each application type for placement, which the Narya provisioner provides. Finally, we also use co-design based methods as baselines, including HASCO, RecSim and CES (see Chapter 2).

Testbed. We perform our experiments on a Microsoft Azure cloud computing platform with tasks as **Azure Serverless Functions** having generated from the AIOtBench application suite. To generate our tasks as serverless functions, we train the layer splits, semantic splits and compressed models using the CIFAR100 dataset [Krizhevsky et al., 2009] and the models in $\mathcal{A}$. The CIFAR100 dataset is a popular computer-vision dataset used to train object detection models and has 32×32 RGB images with a 100-dimensional output corre-

sponding to the object classes. Based on the deployment decisions, we use the serverless functions corresponding to the specific deployment and application types.

We consider the complete set of provisionable hosts $\mathcal{H}_t$ to be static with time as is common in prior work for a fixed cloud platform [Luo et al., 2020, 2021, Stuckey et al., 2020]. We use diverse VM types, $\mathcal{V}_t$, in our cloud infrastructure, *i.e.*, **B2s** with a dual-core CPU and 4GB RAM, **B4ms** with a quad-core CPU and 16GB RAM and **B8ms** with an octa-core CPU and 32 GB RAM. We consider a geographically distributed cloud environment with at most 50 VMs that can be provisioned, *i.e.*, $|\mathcal{H}_t| = 50$. Our environment consists of 30 VMs in the **UK-South** Azure datacenter and 20 in the **East-US** datacenter. The **UK-South** cluster consist of 15 **B2s** and 15 **B4ms** nodes, while our **East-US** cluster consists of 10 **B4ms** and 10 **B8ms** nodes. The provisioners in our experiments can provision up to these VM limits at any interval. Our resource management policies are run on a cloud broker node in the **UK-South** location and is a **D16asv4** node with a 16 core CPU and 64 GB RAM. The $\bar{c}^i, \bar{r}^i, \bar{s}^i$ were set as per the resultant IPS from running the **sysbench**² CPU benchmarking tool, RAM and Disk capacities as per the Azure cloud VM specifications. We calculate the IPS, RAM and Disk utilizations of tasks using the **iozone**³ linux benchmarking tool. The ξ^i variables of these VM types, described in Section 9.1, are set using Gaussian regression based on 100 datapoints corresponding to the provisioning time for each type. The costs μ^i are taken from Azure pricing calculator⁴ for the **UK-South** and **East-US** Azure datacenters. The power consumption values of increments of 10% CPU utilization of Azure VM types are taken from Standard Performance Evaluation Corporation (SPEC) benchmark repository⁵.

To implement SciNet, we build upon the co-simulation primitives provided by the COSCO framework introduced in Chapter 3 by modifying and integrating with custom resource provisioning methods. Further, we use HTTP RESTful APIs for communication and seamless integration of a **Flask** based web-environment to deploy and monitor serverless functions in our distributed cloud setup [Grinberg, 2018]. All sharing of resource utilization characteristics across brokers uses the **rsync**⁶ utility. We utilize the HTTP Notification API to synchronize outputs and execute workloads.

Comparison Metrics. We use various QoS metrics to compare SciNet with baselines. These include the metrics introduced earlier: utilization ratio r_t , cost ϕ_t , average inference accuracy a_t , SLA violation rate ρ_t , energy consumption e_t and average response time κ_t . Our cumulative QoS score (q_t) is a convex combination of these metrics,

$$q_t \leftarrow q^r \cdot r_t + q^a \cdot a_t + (1 - q^\phi \cdot \phi_t - q^\rho \cdot \rho_t - q^e \cdot e_t - q^\kappa \cdot \kappa_t), \quad (9.20)$$

where $q^r, q^\phi, q^a, q^\rho, q^e, q^\kappa$ are convex-combination weights set as per user requirements. For our experiments, we set them to $q^x = 1/(6 \cdot nc^x)$ where nc^x is the normalization coefficient for metric x . The sign for each QoS metric is based on whether we want to maximize or minimize that metric. For instance, to minimize energy consumption and maximize accuracy, their signs are negative and positive respectively. We consider the relative definition of SLO where the deadline is the 90th percentile response time for the same application in $\mathcal{A}$ on the state-of-the-art baseline CES (baseline with highest QoS score q_t). All model training is performed on a separate server with this configuration: Intel Xeon Silver 2.20-3.00 GHz CPU, 16GB RAM, Nvidia RTX

²Sysbench: <http://manpages.ubuntu.com/manpages/trusty/man1/sysbench.1.html>.

³IOZone: <https://linux.die.net/man/1/iozone>.

⁴Azure Pricing Calculator: <https://azure.microsoft.com/en-us/pricing/calculator/>.

⁵SPEC power consumption repository: https://www.spec.org/cloud_iaas2018/results/

⁶RSync: <https://linux.die.net/man/1/rsync>.

Table 9.1: Comparison of QoS parameters (averaged over intervals) of SciNet against baselines and ablated models.

Model	util. ratio	cost	accuracy	energy	resp. time	SLA viol.	QoS
ARIMA+ACO	0.298 ± 0.082	0.521 ± 0.060	0.492 ± 0.029	5.432 ± 0.021	21.309 ± 1.022	0.341 ± 0.040	0.348 ± 0.028
LSTM+ACO	0.423 ± 0.117	0.503 ± 0.056	0.517 ± 0.049	5.217 ± 0.042	16.921 ± 2.907	0.212 ± 0.019	0.440 ± 0.048
Decision-NN	0.728 ± 0.092	0.772 ± 0.011	0.720 ± 0.076	4.224 ± 0.021	12.2255 ± 0.709	0.091 ± 0.016	0.609 ± 0.031
Semi-Direct	0.753 ± 0.079	0.697 ± 0.106	0.728 ± 0.101	4.194 ± 0.069	17.091 ± 1.041	0.131 ± 0.013	0.554 ± 0.042
GRAF	0.729 ± 0.046	0.562 ± 0.038	0.741 ± 0.060	4.095 ± 0.094	14.772 ± 2.218	0.127 ± 0.017	0.585 ± 0.045
UAHS+Gillis+GOSH	0.607 ± 0.022	0.792 ± 0.107	0.723 ± 0.019	3.721 ± 0.037	20.986 ± 3.033	0.271 ± 0.028	0.498 ± 0.044
CAHS+Gillis+GOSH	0.680 ± 0.071	0.659 ± 0.054	0.737 ± 0.041	3.672 ± 0.029	17.289 ± 2.091	0.266 ± 0.010	0.542 ± 0.035
Narya+SplitPlace	0.721 ± 0.120	0.601 ± 0.024	0.802 ± 0.091	4.625 ± 0.013	22.090 ± 1.178	0.282 ± 0.029	0.466 ± 0.042
HASCO	0.770 ± 0.031	0.521 ± 0.002	0.786 ± 0.044	4.297 ± 0.082	16.128 ± 0.446	0.2252 ± 0.033	0.568 ± 0.027
RecSim	0.809 ± 0.055	0.499 ± 0.037	0.811 ± 0.069	2.720 ± 0.098	15.293 ± 0.929	0.148 ± 0.026	0.649 ± 0.037
CES	0.858 ± 0.049	0.528 ± 0.098	0.837 ± 0.056	2.424 ± 0.107	13.742 ± 0.361	0.103 ± 0.021	0.695 ± 0.027
SciNet_Trans	0.844 ± 0.031	0.429 ± 0.031	0.800 ± 0.012	2.661 ± 0.090	10.293 ± 0.2257	0.084 ± 0.022	0.715 ± 0.015
SciNet_CoDesign	0.798 ± 0.009	0.530 ± 0.011	0.747 ± 0.007	3.129 ± 0.012	19.025 ± 0.021	0.124 ± 0.012	0.585 ± 0.005
SciNet	0.926 ± 0.023	0.401 ± 0.018	0.879 ± 0.016	1.561 ± 0.021	9.272 ± 0.729	0.042 ± 0.009	0.801 ± 0.014

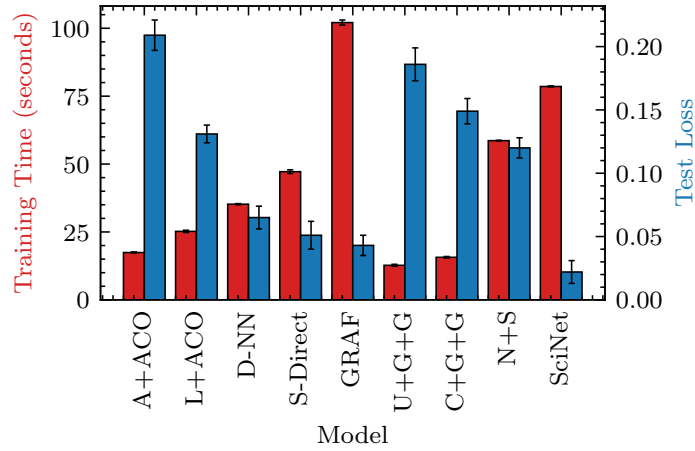


Figure 9.3: Comparison of training times and test loss.

3090 and Ubuntu 18.04 LTS OS. We run for $T = 200$ intervals where the cumulative QoS metric x is defined as $\frac{1}{T} \sum_{t=1}^T x$. We set the hyperparameters γ, ξ, ζ using grid-search over the average QoS score $\frac{1}{T} \sum_{t=1}^T q_t$ (see sensitivity analysis in Section 9.3.5).

9.3.2 Comparison of Training Time and Demand Prediction Loss

The training data for all predict+optimize models as well as SciNet was generated by running the cloud testbed for 500 intervals as described in Section 9.2.3. Generating this data for a scheduling interval $\Delta = 5$ seconds gives a data generation time of 2500 seconds. We use the workload demand data $W_t \forall t \in \{1, \dots, 2500\}$ for the predict+optimize models and additionally the oracle decisions to train SciNet. We use the same training policy as described in the respective chapters of the baseline methods. The independent optimization methods such as UAHS, Gillis and GOSH and training in parallel for the UAHS+Gillis+GOSH baseline. Figure 9.3 shows the training time and the demand prediction loss for the trained models, excluding the methods that do not rely on deep learning. Even though the SciNet model takes at least 34% longer to train than all baselines except GRAF, the self-attention operations in the transformer encoder and decoders enable the model to infer over multiple batches of the time-series windows in parallel, and does not depend on the output of a previous timestamp, improving the training time of SciNet. The highest training time across all models is of the GRAF method (102 seconds) with also the lowest test loss (0.043) that uses a graph neural network that performs

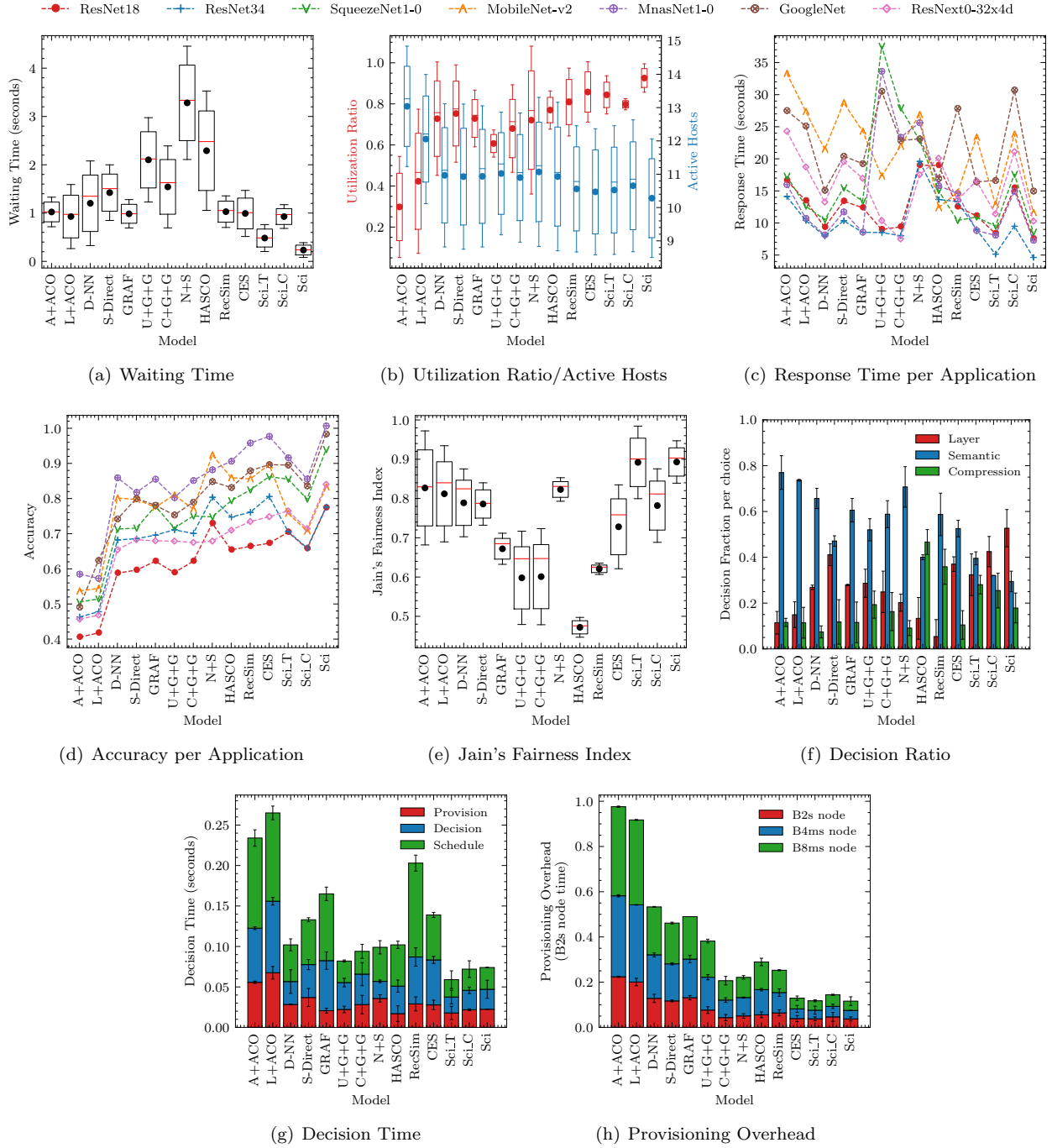


Figure 9.4: Comparison of QoS parameters (averaged over intervals) of SciNet against baselines and ablated models.

graph convolutions across multiple timesteps, giving rise to higher training times. Compared to GRAF, SciNet gives 23% and 49% lower training times and demand prediction loss.

9.3.3 Comparison of QoS Metrics

Table 9.1 presents the comparison metrics and the cumulative QoS scores of all models averaged over the execution intervals. Fig. 9.4 shows the additional metrics for each method. Across all metrics in Table 9.1, SciNet outperforms the baselines. The highest utilization ratio across all baselines is achieved by CES. SciNet improves the utilization ratio by allocating tasks, *i.e.*, serverless functions to the same hosts to minimize resource

Table 9.2: Sensitivity analysis with hyperparameters (γ, ξ, ζ). All standard deviations are <0.05 .

Value	γ			ξ			ζ		
	r	cost	QoS	acc.	SLA	QoS	energy	resp. t.	QoS
0.0	0.930	0.412	0.784	0.910	0.062	0.792	1.293	12.312	0.784
0.1	0.929	0.407	0.787	0.905	0.051	0.794	1.348	11.029	0.787
0.2	0.929	0.403	0.798	0.902	0.048	0.795	1.398	10.421	0.789
0.3	0.927	0.401	0.799	0.893	0.046	0.797	1.439	10.019	0.792
0.4	0.926	0.401	0.801	0.885	0.043	0.798	1.478	9.892	0.793
0.5	0.922	0.398	0.798	0.879	0.042	0.801	1.501	9.562	0.796
0.6	0.910	0.394	0.798	0.876	0.041	0.800	1.523	9.312	0.799
0.7	0.902	0.391	0.795	0.866	0.041	0.799	1.561	9.272	0.801
0.8	0.898	0.390	0.789	0.854	0.041	0.798	1.581	9.269	0.800
0.9	0.887	0.390	0.782	0.850	0.040	0.798	1.664	9.264	0.800
1.0	0.869	0.389	0.769	0.838	0.039	0.797	1.928	9.263	0.799

wastage and the active hosts in the system. We define a host to be active when the CPU utilization is $> 0\%$. We use the **Azure Automation**⁷ service to auto-hibernate and resume VMs based on the CPU utilization of the VMs. Based on the provisioning decisions, intelligent task placement in SciNet allows it to minimize the number of active hosts (see Fig. 9.4(b)). This also facilitates the minimization of the energy consumption and cost, where SciNet gives average energy consumption of 1.561 KW-hr, which is 35.43% lower than the lowest baseline value of 2.424 KW-hr by CES and an average cost of 0.401 USD per interval, 19.63% lower than the best baseline RecSim. SciNet also gives an average accuracy of 0.879, 5% higher than the best accuracy of 0.836 of the CES baseline. This is due to the higher ratio of the layer-splitting decision that does not alter the model by splitting the model by output type (semantic splitting) or pruning the DNN model (model compression) as is observed from Figs. 9.4(d) and 9.4(f). SciNet also gives the lowest average SLA violation rate of 0.042, *i.e.*, 59.2% lower compared to the lowest baseline score of CES, which is 0.103. This is due to the lower average response time (see Fig. 9.4(c)), task wait time to get placed to a host (see Fig. 9.4(a)) and decision time (see Figs. 9.4(g) and 9.4(h)). In terms of the overall QoS score, the best baseline is CES with a score of 0.695. SciNet improves upon this by 15.25%, giving the score of 0.801.

9.3.4 Ablation Analysis

To test the efficacy of the co-design aspect and the transformer based neural models in SciNet, we modify the approach as follows. First, we consider a model without the transfer for provisioning scores to the deployment network or the provisioning and deployment decisions to the placement network, but instead generate likelihood scores of each decision independently. We call this the SciNet_CoDesign model. Second, we replace the transformer with a feed-forward network to test the importance of temporal trends that the transformer captures. We call this the SciNet_Trans model. The results in Table 9.1 and Fig. 9.4 show a drop in all performance metrics for these models when compared to SciNet, demonstrating the effectiveness of the transformer based neural architecture and model training using auto-regressive decision making strategy in SciNet.

9.3.5 Sensitivity Analysis

Table 9.2 shows the performance of the SciNet method for different values of the hyperparameters γ, ξ, ζ . γ is a parameter to control the weight given to the cost metric when performing independent optimization of the provisioning decision using the co-simulator. Low γ corresponds to private-cloud deployments that do not

⁷Azure Automation Service: <https://azure.microsoft.com/en-us/services/automation/#overview>.

have any monetary running costs, but require maximization of utilization ratio to reduce resource wastage. High γ corresponds to workload execution on public clouds with limited cost budgets, where the user aims at optimizing both system utilization and running costs. Similarly, ξ and ζ are weights for the SLA violation rates and response time, respectively. High ξ and ζ correspond to time-critical application scenarios, such as self-driving cars and healthcare. For comparison with baselines, we use $\gamma = 0.4, \xi = 0.5, \zeta = 0.7$ as per grid-search, while maximizing QoS scores. The SciNet gives higher QoS scores than each baseline for every value of γ, ξ and ζ , providing clear evidence of the robustness of the SciNet model to different deployment scenarios and hyperparameter settings.

9.4 Summary

This chapter proposes SciNet a co-simulated digital twin based imitation learner that dynamically and concurrently predicts VM provisioning, workload deployment and task placement decisions to optimize QoS of a cloud computing environment. It uses a transformer based composite neural network to first predict the workload demands in the near future and auto-regressively find the optimal set of provisioning, deployment and placement actions. Imitation learning enables us to train a scalable and online action predictor in order to imitate the QoS effects of each action using a co-simulator. An offline co-simulations driven oracle facilitates accurate performance prediction for each action, allowing optimal decision making and avoiding excessive provisioning overheads at test time. Experiments with real-life AI-based benchmark applications on a public cloud testbed show that SciNet gives up to 48% lower execution cost, 79% higher inference accuracy, 71% lower energy consumption and 56% lower response times compared to the current state-of-the-art methods. Thus, we build on the ideas of co-simulated oracle developed in Chapters 3 and 4 to concurrently optimize resource management decisions, solving multiple objectives described in Chapter 1 at once.

Future work would investigate the application of the SciNet approach to also include additional resource management decisions such as autoscaling. We also aim to explore the application of auto-regressive decision making in the domain of fault-tolerant computing.

Chapter 10

Conclusions and Future Work

10.1 Summary of Contributions

In this thesis we demonstrate that, in the last decade, the landscape of computing paradigms has witnessed a gradual yet remarkable shift from monolithic computing to distributed and decentralized paradigms such as IoT, Edge, Fog, Cloud, and Serverless Computing (see Chapter 2). This thesis contributes to the frontiers of these computing technologies by boosting the shift from manually encoded algorithms to AI and Co-simulation driven autonomous systems for optimum and reliable management of distributed computational resources.

To this end, the presented thesis reviews the evolution of data-driven AI-augmented technologies and their impact on computing systems in Chapter 2. We demystify new techniques and draw key insights in Edge, Fog and Cloud resource management-related uses of AI methods and also look at how AI can innovate traditional applications for enhanced QoS in the presence of a continuum of resources. We emphasize the importance of leveraging both AI and co-simulation in tandem to optimize resource management decisions in fog environments in Chapter 3. To this end, we develop a novel framework, namely COSCO, that supports the interaction of three elements: AI-based resource management decision making module, real infrastructure as well as a co-simulated digital-twin of the system. This facilitates us to utilize co-simulations for blending AI-based decision-making approaches with heuristic-based optimization strategies. We demonstrate that deep neural networks, when used as surrogate models, can be utilized to perform quick and accurate gradient-based optimization of QoS metrics. We also show that an interactive dynamic, as in GOBI*, enables us to perform what-if tests to generate additional data points and augment data-driven solutions. This has been extended even to train a deep-learning-based resource provisioning system using a simulation-based oracle in Chapter 4, named the CILP approach, allowing us to take optimal resource provisioning decisions (objective 1(a)). Considering a provisioned fog environment, we show that the approach is versatile enough to be integrated with other data-driven policies, such as Multi-Armed-Bandits (MABs), to make more informed decisions. A specific example of this is SplitPlace in Chapter 5, where we leverage deep learning and simulations in tandem with MABs to optimally deploy AI workloads by placing them on edge devices to minimize energy consumption and response time. This allows us to have optimal application deployment in fog environments (objective 1(b)). Furthermore, we show that the gradient optimization using back-propagation to input (GOBI) is not restricted to single-step or first-order optimization. We show in Chapter 6 that when we utilize higher-order optimization strategies, stochastic neural networks and

long-horizon optimization using adaptations of Monte-Carlo learning, we are able to improve the optimization parameters in fog environments such as energy consumption and response time. This translates to the GOSH and MCDS approaches that subsequently reduce the fraction of violations of service level agreements. The GOSH and MCDS schedulers enable us to outperform state-of-the-art methods in terms of system QoS (objective 2). Considering a system where we can dynamically provision, deploy and schedule tasks, we illustrate the need for fault-tolerant computing in Chapter 7. We develop supervised (PreGAN) and unsupervised (DeepFT) fault-tolerance strategies to detect, diagnose and classify faults for informed remediation in the form of preemptive migrations. We also extend these approaches to semi-supervised training, namely PreGAN+, to enable us to adapt the approaches to dynamically changing settings. Compared to fault-oblivious models, our methods facilitate improving QoS parameters for volatile scenarios with stochastic workloads (achieving objective 3(b)). Finally, even with fault-tolerant approaches, the devices may fail, leading to pernicious system downtimes. These directly affect distributed fog architectures, which follow a hierarchical design based on broker-worker topology. To eschew adverse effects on services, we build worker (DRAGON) and broker (CAROL) resilience models in Chapter 8, which update the system topology for seamless workload execution and minimum impact on system QoS. Both these models, and their extensions allow us to build a resilient and dependable fog environment (for objective 3(b)). Finally, we discuss ways to synergistically make decisions on disparate aspects of resource management in Chapter 9.

We next present the latest trends and impact areas, such as optimizing AI models that are deployed *on* or *for* computing systems and possible future directions that this thesis opens.

10.2 Future Directions

We first discuss upcoming application areas in which the methods that we have developed may add value and how additional work may be required to improve QoS further, considering the various beneficiaries discussed in Chapter 1.

Healthcare. With increased adoption of body sensors, there is a massive increase in the volume and variety of available health data, paving the way for the development of more reliable and robust AI models in the areas of proactive monitoring, disease prevention, and more in smart healthcare [Panesar, 2019]. However, this results in challenges related to ensuring data quality and security, especially in the context of distributed EdgeAI, particularly in the case of handling sensitive healthcare information like patient records. To overcome these challenges, future research could focus on leveraging Blockchain and AI to improve data integrity [Yaqoob et al., 2021]. Moreover, low latency communication technologies like 5G/6G enable novel technologies like Augmented Reality, Virtual Reality and Tactile Internet, thus improving and expanding services such as robot-assisted surgery. This is critical for application scenarios where resource management approaches such as GOBI and MCDS are required to handle workloads with sensitive healthcare data.

Next Generation Networking. Due to the high data rate, reliability, ultra-low latency, and ultra-low energy consumption provided by 5G/6G, these wireless communication technologies are identified as the key enablers of future IoT applications. However, to support the ever-evolving service requirements of the IoT services, 5G/6G technologies have to be able to observe environment variations and dynamically self organize the network accordingly [Li et al., 2017]. This can be achieved through AI-empowered management and orchestration of

cellular resources within 5G/6G networks. With the advancements in Software Defined Networking (SDN) and network function virtualization (NFV), 5G/6G technologies introduce Network Slicing (NS) to support this. NS is a mechanism for provisioning virtualized network resources intelligently based on the service performance metrics [Letaief et al., 2019]. Learning algorithms such as deep learning and reinforcement learning can improve the dynamism of NS through prediction-based proactive resource allocation in Edge or Cloud architectures with dynamic slice creations for different applications [Wijethilaka and Liyanage, 2021]. We also need to consider upcoming cases where the internet is provided through satellites (such as the Starlink network) in lieu of the conventional copper/fibre connections. In such cases, the latency and bandwidth characteristics might be significantly different, requiring re-tuning hyperparameters or adapting existing AI based resource management policies [Song et al., 2021, Wang et al., 2019]. This would augment approaches such as SplitPlace, where several models may need speedy communication to reduce response times.

Production and Supply Chain. In the coming age of IIoT, most industrial pipelines are managed by smart devices. Such devices may utilize AI methods to self-monitor and predict potential problems in the supply chain to optimize the overall service efficiency. The COVID-19 pandemic is an example demonstrating the importance of automation in logistics to avoid service downtimes [Salehi-Amiri et al., 2021]. AI based forecasting approaches, such as recurrent neural models and GANs can be used to predict stock shortage and proactively order additional stocks to prevent shortage [Salehi-Amiri et al., 2021]. Similarly, large-scale ML models can aid the development of smart-manufacturing technologies that utilize several IoT and Fog devices to collaboratively monitor, control manufacturing and production related equipment. Thus, effective provisioning (such as using CILP) may be required to scale production systems for large-scale services.

Smart Cities. Advancements in wireless communications such as 5G/6G enable a massive amount of data to be transmitted towards Edge nodes in real-time. This has resulted in the rise of novel technologies like crowd-sensing and crowd-sourcing [Kong et al., 2019]. Distributed collections and processing of such massive volumes of data demands future research to focus on distributed and reliable AI, specifically in data-sensitive applications at the Edge. This requires methods such as GOBI and GOSH to be extended for fair service delivery. At the same time, data security becomes crucial in future smart city services, with the widespread use of crowd-sensing and crowd-sourcing for data collection. Moreover, ultra-low latency communication provided future radio access networks to support services related to hazard avoidance and safety [Rudd-Orthner and Mihaylova, 2020], which requires AI models with higher reliability, accuracy and lower latency.

We next discuss future directions along the lines of emerging challenges.

Legacy Deployment. A key challenge faced by some neural models, such as GOBI, GOBI*, PreGAN, is the assumption of a fixed size input. This makes them inapplicable in changing infrastructures or deployment scenarios. Stemming from the scalability limitations of state-of-the-art, the integration of large-scale DNN models within legacy edge or cloud machines has become a challenging problem. As research progresses and more accurate and better performing models are developed, it typically follows the adoption of advanced AI models by industry. However, more complex AI models are usually more resource-hungry and need more powerful systems to be deployed on. To deploy an enhanced AI model, technology based companies such as Meta, Amazon, Netflix and Google frequently upgrade their devices, raising many sustainability concerns [Gill et al., 2019]. As research moves in the direction of the neural design of sophisticated AI models, it becomes crucial to ensure that these new models can be deployed on legacy infrastructures to bring down deployment

costs and carbon footprint of AI. Further, having running sophisticated deep neural networks, Monte Carlo methods and tree searches in GOSH and MCDS require resource intensive broker nodes. A research direction could augment these approaches and their lightweight versions.

Automated and Generic Modelling. Another challenge being faced by AI models, specifically imitation learning methods such as CILP and SciNet, is the inability to tune AI models in settings different from the ones used for training [Liaw et al., 2018]. This is because CILP and SciNet assume the action set to be of a constant size. This may not be the case and retraining may be required if the action space changes. As the performance of AI models is highly dependent on the proper tuning of a large number of hyperparameters, these variables need to be re-tuned when deploying a pre-trained model in a new setting of scheduling or fault detection in Edge/Cloud. This problem stems from generalizability, but needs to be solved specifically for each application domain of deployment, scheduling or maintenance. In such cases, either the hyperparameter values of the models need to be decided in an automated fashion, or the neural design needs to be generic enough to accommodate new settings, with possibly different number of nodes, workload characteristics and user demands.

Interpretability. Many state-of-the-art AI methods developed in this work, including GOBI, GOBI*, CILP, PreGAN, DeepFT, DRAGON and CAROL, are black-box models. They do not have any transparency on the process that led to the various resource management level decisions made by an AI agent. Interpretability mainly entails explaining the main reasons for choosing or not choosing certain management decisions and exploring the unknown state spaces to ensure exhaustive coverage of the decision space. For sensitive industrial segments, such as healthcare and autonomous vehicles, it is crucial to expose the underlying patterns and features in the decision-making process to gain credibility for the end-user. Building such white-box or explainable models is an emerging field of research for the use of trustworthy AI models. Many AI models, such as decision trees and regression algorithms allow interpretability, but are not as accurate or scalable as deep-learning based counterparts [Ehsan et al., 2021].

We now discuss various methods and ways in which these emerging challenges may be addressed.

Self-supervised AI. As we have seen for PreGAN+, self-supervised learning technique enables learning with unlabeled data by solving pretext tasks [Saeed et al., 2020]. In contrast to this, supervised learning depends on the availability of labeled data. Even though a massive amount of data gets generated by the sensors, the lack of annotated data poses an obstacle for using supervised learning. This is specifically applicable in scenarios where fault-detection or workload scheduling is required for previously unseen workload or device characteristics in Edge or Cloud platforms. As generating expert-labeled fault labels or optimal scheduling decisions is infeasible for large-scale systems, self-supervised learning offers a possible solution for such scenarios. This approach is capable of generating a more generalizable model by removing the heavy dependency on labeled data by automatically generated data annotations, possibly using a co-simulator [Saeed et al., 2020]. Moreover, self-supervised learning has the potential to achieve higher reliability due to its robustness to adversarial examples, label corruption, and input corruptions [Hendrycks et al., 2019]. Thus, other approaches, such as for service resilience, DRAGON and CAROL, may be extended with self-supervised modelling to enable learning even with little or no labelled data.

Model Driven RL. As Fog environments and service demands are dynamic; algorithms should have the capability to adapt accordingly. IoT applications (*i.e.*, healthcare, smart cities, etc.) and their enabling telecommunication technologies (*i.e.*, 5G, 6G) benefit from RL-based intelligence due to higher adaptability

of RL techniques and their ability to learn without prior knowledge [Sami et al., 2021]. However, exploration errors, long learning time and distributed learning within resource constraint devices in Edge environments are some of the challenges in utilizing RL techniques within resource-constrained and distributed Fog environments. A canonical case is the development of RL based methods for deploying large-scale application workflows on constrained Edge or Cloud clusters. Addressing these challenges in future research is vital for the RL-based approaches to reach their full potential within EdgeAI scenarios. In an attempt to address these challenges, EdgeAI research is exploring advanced RL approaches such as model-based RL [Sutton and Barto, 2018] and co-simulated RL [Amini et al., 2020]. These approaches can be used to extend imitation learning models, including CILP and SciNet.

Analog AI. The current implementation of AI is targeted for digital systems where the values are stored in a binary format. Herein, the major challenge posed by digital implementations of DNNs is the linear dependence of memory footprint with the number of parameters of the neural model. Upcoming analog memory-based chips present new ways to perform the same operations, but with orders of magnitude lower amount of memory requirement, computational load and energy consumed [Channamadhavuni et al., 2021]. This is particularly useful in memory-constrained Edge and MEC devices where sophisticated DNNs need to be executed in *AI on Fog* setting or decentralized resource management is required in *AI for Fog* settings. However, there are some drawbacks in the loss of precision in computation across layers within a DNN. The trade-off offered by such DNN implementations is similar to model pruning and splitting, but with possibly more extreme energy/compute benefits. This direction has been explored to a limited extent and requires further investigation and software development to efficiently harness the potential of analog hardware accelerators and neuromorphic computing. Such approaches would facilitate all high memory footprint models, specifically those based on Transformer encoders, such as PreGAN+. Analog AI should facilitate in the reduction of such resource footprints.

Decentralized Modeling. The success of distributed EdgeAI, by utilizing layer and semantic splitting strategies for AI deployment, shows some promise in other domains of scheduling and maintenance. For scheduling applications and maintaining Fog systems at scale, it is possible to decentralize the training and inference procedures of the resource management level AI applications across multiple broker nodes. The decentralized fashion of resource management has a two-fold benefit. Firstly, there is no single point of failure in the system as the management steps are run in multiple broker nodes. Secondly, it allows the distribution of resource management load across multiple computing devices, facilitating the scalability of the model. All centralized models, such as SplitPlace and GOBI, can be made decentralized to allow robust resource management.

AI Driven Simulations. A major advantage of co-simulators in Fog systems is the ability to generate new data points for tuning AI models and resolving issues like exposure bias and data saturation [Renda et al., 2020]. However, another important benefit of co-simulators is the ability to run multiple simulations concurrently and pick the best resource management decisions, allowing interpretable decision-making. This is due to the ability of co-simulators to generate a complete execution trace, possible for several future states of the systems and allowing developers or end-users to visualize the long-term effects of various decisions. They are able to do this much faster than executing the decisions on a physical infrastructure and waiting for several minutes to reflect changes in QoS. This is primarily due to the discrete event-driven execution style of modern simulators. This is applicable for all three types of resource management decisions, *i.e.*, deployment, scheduling and maintenance. Simulations can indicate changes in QoS scores for each model compression or splitting

type, application placement or fault-remediation steps like preemptive migrations. These signals can facilitate decision making. However, co-simulators are mere approximations of the entire Fog system and typically fail to map the entire complexity of real infrastructure. However, the success of deep surrogate models hints us to build simulators with possibly millions of parameters and utilize DNNs to estimate optimal parameter values, such that our simulators resemble the real systems as closely as possible. An increased number of parameters could, in principle, give a higher representative capacity to our simulators, now being able to map complex real-life workloads and device characteristics. Thus, AI-driven simulators could help improve system performance with the added bonus of interpretable decision making. Directions in improving co-simulators using AI should facilitate all online fine-tuning models such as PreGAN+, GOBI, GOBI* and SplitPlace. Some preliminary evidence has been shown in prior work on this direction [Collins et al., 2020, Renda et al., 2020].

Bibliography

- [1] PyTorch Ray Tune. https://pytorch.org/tutorials/beginner/hyperparameter_tuning_tutorial.html. Accessed: 2021-10-20.
- [2] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *ACM SIGSAC conference on computer and communications security*, pages 308–318, 2016.
- [3] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie. Mobile edge computing: A survey. *IEEE Internet of Things Journal*, 5(1):450–465, 2017.
- [4] M. Abdullah, W. Iqbal, A. Mahmood, F. Bukhari, and A. Erradi. Predictive autoscaling of microservices hosted in fog microdata center. *IEEE Systems Journal*, 15(1):1275–1286, 2020.
- [5] M. Adhikari, T. Amgoth, and S. N. Srirama. A survey on scheduling strategies for workflows in cloud environment and emerging trends. *ACM Computing Surveys (CSUR)*, 52(4):1–36, 2019.
- [6] J. Ahlgren, M. E. Berezin, K. Bojarczuk, E. Dulskyte, I. Dvortsova, J. George, N. Gucevskas, M. Harman, R. Laemmel, E. Meijer, et al. WES: Agent-based user interaction simulation on real infrastructure. In *IEEE/ACM International Conference on Software Engineering Workshops*, pages 276–284, 2020.
- [7] A. Ahmed and G. Pierre. Docker container deployment in fog computing infrastructures. In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 1–8. IEEE, 2018.
- [8] E. Ahmed and M. H. Rehmani. Mobile edge computing: Opportunities, solutions, and challenges, 2017. ISSN 0167-739X.
- [9] S. P. Ahuja, S. Mani, and J. Zambrano. A survey of the state of cloud computing in healthcare. *Network and Communication Technologies*, 1(2):12, 2012.
- [10] M. S. Al-Asaly, M. A. Bencherif, A. Alsanad, and M. M. Hassan. A deep learning-based resource usage prediction model for resource provisioning in an autonomic cloud computing environment. *Neural Computing and Applications*, 34(13):10211–10228, 2022.
- [11] A. Alelaiwi. An efficient method of computation offloading in an edge cloud platform. *Journal of Parallel and Distributed Computing*, 127:58–64, 2019.
- [12] M. Aliyu, M. Murali, A. Y. Gital, and S. Boukari. Efficient metaheuristic population-based and deterministic algorithm for resource provisioning using ant colony optimization and spanning tree. *International Journal of Cloud Applications and Computing (IJCAC)*, 10(2):1–21, 2020.

- [13] R. Alsurdeh, R. N. Calheiros, K. M. Matawie, and B. Javadi. Hybrid workflow scheduling on edge cloud computing systems. *IEEE Access*, 9:134783–134799, 2021.
- [14] A. Amini, I. Gilitschenski, J. Phillips, J. Moseyko, R. Banerjee, S. Karaman, and D. Rus. Learning robust control policies for end-to-end autonomous driving from data-driven simulation. *IEEE Robotics and Automation Letters*, 5(2):1143–1150, 2020.
- [15] H. Arabnejad, C. Pahl, G. Estrada, A. Samir, and F. Fowley. A fuzzy load balancer for adaptive fault tolerance management in cloud platforms. In *European Conference on Service-Oriented and Cloud Computing*, pages 109–124. Springer, 2017.
- [16] A. Aral and I. Brandic. Quality of service channelling for latency sensitive edge applications. In *2017 IEEE International Conference on Edge Computing (EDGE)*, pages 166–173. IEEE, 2017.
- [17] A. Aral and I. Brandic. Dependency mining for service resilience at the edge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 228–242, 2018. doi: 10.1109/SEC.2018.00024.
- [18] A. Aral and I. Brandić. Learning spatiotemporal failure dependencies for resilient edge computing services. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1578–1590, 2021. doi: 10.1109/TPDS.2020.3046188.
- [19] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.
- [20] A. Asghari, M. K. Sohrabi, and F. Yaghmaee. Task scheduling, resource provisioning, and load balancing on scientific workflows using parallel SARSA reinforcement learning agents and genetic algorithm. *The Journal of Supercomputing*, 77(3):2800–2828, 2021.
- [21] J. Audibert, P. Michiardi, F. Guyard, S. Marti, and M. A. Zuluaga. USAD: UnSupervised Anomaly Detection on Multivariate Time Series. In *ACM International Conference on Knowledge Discovery & Data Mining*, pages 3395–3404, 2020.
- [22] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.
- [23] J. Bae, J. Lee, and S. Chong. Beyond max-weight scheduling: A reinforcement learning-based approach. In *2019 International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOPT)*, pages 1–8. IEEE, 2019.
- [24] J. Baek and G. Kaddoum. Heterogeneous task offloading and resource allocations via deep recurrent reinforcement learning in partial observable multifog networks. *IEEE Internet of Things Journal*, 8(2): 1041–1056, 2020.
- [25] S. Bagchi, M.-B. Siddiqui, P. Wood, and H. Zhang. Dependability in edge computing. *Communications of the ACM*, 63(1):58–66, 2019.
- [26] D. Bahdanau, K. H. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015.

-
- [27] Z. Bakhshi, G. Rodriguez-Navas, and H. Hansson. Dependable fog computing: A systematic literature review. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 395–403. IEEE, 2019.
- [28] D. Barrett and B. Dherin. Implicit gradient regularization. In *International Conference on Learning Representations*, 2020.
- [29] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.
- [30] D. Basu, X. Wang, Y. Hong, H. Chen, and S. Bressan. Learn-as-you-go with Megh: Efficient live migration of virtual machines. *IEEE Transactions on Parallel and Distributed Systems*, 30(8):1786–1801, 2019.
- [31] S. Basu, A. Banerjee, and R. Mooney. Semi-supervised clustering by seeding. In *International Conference on Machine Learning*, 2002.
- [32] D. Bega, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez. DeepCog: Optimizing resource provisioning in network slicing with AI-based capacity forecasting. *IEEE Journal on Selected Areas in Communications*, 38(2):361–376, 2019.
- [33] L. Belcastro, A. Falcone, A. Garro, and F. Marozzo. Evaluation of large scale roi mining applications in edge computing environments. In *IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pages 1–8, 2021.
- [34] A. Beloglazov and R. Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency and Computation: Practice and Experience*, 24(13):1397–1420, 2012.
- [35] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. Characterization of scientific workflows. In *2008 third workshop on workflows in support of large-scale science*, pages 1–10. IEEE, 2008.
- [36] K. Bilal, E. Baccour, A. Erbad, A. Mohamed, and M. Guizani. Collaborative joint caching and transcoding in mobile edge networks. *Journal of Network and Computer Applications*, 136:86–99, 2019.
- [37] C. Bitsakos, I. Konstantinou, and N. Koziris. Derp: A deep reinforcement learning cloud system for elastic resource provisioning. In *2018 IEEE international conference on cloud computing technology and science (CloudCom)*, pages 21–29. IEEE, 2018.
- [38] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana. The internet of things, fog and cloud continuum: Integration and challenges. *Internet of Things*, 3:134–155, 2018.
- [39] L. Bogolubsky, P. Dvurechenskii, A. Gasnikov, G. Gusev, Y. Nesterov, A. M. Raigorodskii, A. Tikhonov, and M. Zhukovskii. Learning supervised pagerank with gradient-based and gradient-free optimization methods. In *Advances in neural information processing systems*, pages 4914–4922, 2016.
-

- [40] K. Bloor, R. Chirkova, Y. Viniotis, and T. Salo. Dynamic request allocation and scheduling for context aware applications subject to a percentile response time sla in a distributed cloud. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 464–472. IEEE, 2010.
- [41] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.
- [42] S. Bond-Taylor and C. G. Willcocks. Gradient origin networks. In *International Conference on Learning Representations*, 2021.
- [43] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012.
- [44] S. Bosmans, S. Mercelis, J. Denil, and P. Hellinckx. Testing IoT systems using a hybrid simulation based testing approach. *Computing*, 101(7):857–872, 2019.
- [45] D. Bouneffouf, I. Rish, and C. Aggarwal. Survey on applications of multi-armed and contextual bandits. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2020.
- [46] D.-M. Bui, Y. Yoon, E.-N. Huh, S. Jun, and S. Lee. Energy efficiency for cloud computing system based on predictive optimization. *Journal of Parallel and Distributed Computing*, 102:103–114, 2017.
- [47] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.
- [48] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.
- [49] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya. Workload prediction using arima model and its impact on cloud applications’ qos. *IEEE transactions on cloud computing*, 3(4):449–458, 2014.
- [50] D. Callegaro, Y. Matsubara, and M. Levorato. Optimal task allocation for time-varying edge computing systems with split dnns. In *IEEE Global Communications Conference*, pages 1–6, 2020.
- [51] X. Cao, G. Tang, D. Guo, Y. Li, and W. Zhang. Edge federation: Towards an integrated service provisioning model. *IEEE/ACM Transactions on Networking*, 28(3):1116–1129, 2020.
- [52] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini. Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(5): 871–875, 2020.
- [53] G. Carvalho, B. Cabral, V. Pereira, and J. Bernardino. Edge computing: current trends, research challenges and future directions. *Computing*, 103(5):993–1023, 2021.

-
- [54] G. Casale, M. Artac, W.-J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Panikolaou, D. Presenza, A. Russo, et al. RADON: rational decomposition and orchestration for serverless computing. *SICS Software-Intensive Cyber-Physical Systems*, 35(1):77–87, 2020.
- [55] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, 2019.
- [56] S. Channamadhavuni, S. Thijssen, S. K. Jha, and R. Ewetz. Accelerating AI Applications using Analog In-Memory Computing: Challenges and Opportunities. In *Great Lakes Symposium on VLSI*, pages 379–384, 2021.
- [57] J. Chen and X. Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.
- [58] J. Chen and Y. Wang. A resource demand prediction method based on eemd in cloud computing. *Procedia Computer Science*, 131:116–123, 2018.
- [59] J. Chen, S. Chen, Q. Wang, B. Cao, G. Feng, and J. Hu. iRAF: A deep reinforcement learning approach for collaborative mobile edge computing IoT networks. *IEEE Internet of Things Journal*, 6(4):7011–7024, 2019.
- [60] M. Chen, T. Wang, S. Zhang, and A. Liu. Deep reinforcement learning for computation offloading in mobile edge computing environment. *Computer Communications*, 175:1–12, 2021.
- [61] N. Chen, Y. Chen, S. Song, C.-T. Huang, and X. Ye. Smart urban surveillance using fog computing. In *2016 IEEE/ACM symposium on edge computing (SEC)*, pages 95–96. IEEE, 2016.
- [62] X. Chen, Q. Shi, L. Yang, and J. Xu. Thriftyedge: Resource-efficient edge computing for intelligent iot applications. *IEEE network*, 32(1):61–65, 2018.
- [63] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis. Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning. *IEEE Internet of Things Journal*, 6(3):4005–4018, 2018.
- [64] X. Chen, H. Wang, Y. Ma, X. Zheng, and L. Guo. Self-adaptive resource allocation for cloud-based software services based on iterative qos prediction model. *Future Generation Computer Systems*, 105:287–296, 2020.
- [65] Y. Chen, Z. Liu, Y. Zhang, Y. Wu, X. Chen, and L. Zhao. Deep reinforcement learning-based dynamic resource management for mobile edge computing in industrial internet of things. *IEEE Transactions on Industrial Informatics*, 17(7):4925–4934, 2020.
- [66] Z. Cheng, M. Min, M. Liwang, L. Huang, and Z. Gao. Multi-agent ddpg-based joint task partitioning and power control in fog computing networks. *IEEE Internet of Things Journal*, 2021.
- [67] J. Chorowski, D. Bahdanau, D. Serdyuk, K. Cho, and Y. Bengio. Attention-based models for speech recognition. In *Advances in Neural Information Processing Systems*, 2015.
-

- [68] S. Chouliaras and S. Sotiriadis. Detecting performance degradation in cloud systems using LSTM autoencoders. In *International Conference on Advanced Information Networking and Applications*, pages 472–481. Springer, 2021.
- [69] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.
- [70] C. Cicconetti, M. Conti, and A. Passarella. A decentralized framework for serverless edge computing in the internet of things. *IEEE Transactions on Network and Service Management*, 18(2):2166–2180, 2020.
- [71] C. N. Coelho, A. Kuusela, S. Li, H. Zhuang, J. Ngadiuba, T. K. Aarrestad, V. Loncar, M. Pierini, A. A. Pol, and S. Summers. Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. *Nature Machine Intelligence*, pages 1–12, 2021.
- [72] T. Coleman, H. Casanova, L. Pottier, M. Kaushik, E. Deelman, and R. F. da Silva. WfCommons: A Framework for Enabling Scientific Workflow Research and Development. *arXiv preprint arXiv:2105.14352*, 2021.
- [73] J. Collins, R. Brown, J. Leitner, and D. Howard. Traversing the reality gap via simulator tuning. *arXiv preprint arXiv:2003.01369*, 2020.
- [74] A. M. Connor and K. Shea. A comparison of semi-deterministic and stochastic search techniques. In *Evolutionary Design and Manufacture*, pages 287–298. Springer, 2000.
- [75] M. Copeland, J. Soh, A. Puca, M. Manning, and D. Gollob. Microsoft azure and cloud computing. In *Microsoft Azure*, pages 3–26. Springer, 2015.
- [76] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *SOSP*, pages 153–167, 2017.
- [77] P. A. Costa, X. Bai, F. M. Ramos, and M. Correia. Medusa: An efficient cloud fault-tolerant mapreduce. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 443–452. IEEE, 2016.
- [78] R. A. da Silva and N. L. da Fonseca. Resource allocation mechanism for a fog-cloud infrastructure. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018.
- [79] A. V. Dastjerdi, H. Gupta, R. N. Calheiros, S. K. Ghosh, and R. Buyya. Fog computing: Principles, architectures, and applications. In *Internet of things*, pages 61–75. Elsevier, 2016.
- [80] A. Deng and B. Hooi. Graph neural network-based anomaly detection in multivariate time series. *AAAI*, 2021.
- [81] S. Deng, Z. Xiang, J. Taheri, M. A. Khoshkholghi, J. Yin, A. Y. Zomaya, and S. Dustdar. Optimal application deployment in resource constrained distributed edges. *IEEE Transactions on Mobile Computing*, 20(5):1907–1923, 2020.

-
- [82] R. Ding, X. Li, X. Liu, and J. Xu. A cost-effective time-constrained multi-workflow scheduling strategy in fog computing. In *International Conference on Service-Oriented Computing*, pages 194–207. Springer, 2018.
- [83] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg. Byzantine fault tolerance, from theory to reality. In *International Conference on Computer Safety, Reliability, and Security*, pages 235–248. Springer, 2003.
- [84] W. Du, X. Zhang, Q. He, W. Liu, G. Cui, F. Chen, Y. Ji, C. Cai, and Y. Yang. Fault-tolerating edge computing with server redundancy based on a variant of group degree centrality. In *International Conference on Service-Oriented Computing*, pages 198–214. Springer, 2020.
- [85] J. Durbin and S. J. Koopman. *Time series analysis by state space methods*. Oxford university press, 2012.
- [86] F. Ebadifard and S. M. Babamir. Autonomic task scheduling algorithm for dynamic workloads through a load balancing technique for the cloud-computing environment. *Cluster Computing*, 24(2):1075–1101, 2021.
- [87] U. Ehsan, P. Wintersberger, Q. V. Liao, M. Mara, M. Streit, S. Wachter, A. Riener, and M. O. Riedl. Operationalizing human-centered perspectives in explainable ai. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–6, 2021.
- [88] C. Eksin and A. Ribeiro. Distributed fictitious play for multiagent systems in uncertain environments. *IEEE Transactions on Automatic Control*, 63(4):1177–1184, 2017.
- [89] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott. Proactive fault tolerance using preemptive migration. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 252–257. IEEE, 2009.
- [90] A. E. Eshratifar, A. Esmaili, and M. Pedram. Bottlenet: A deep learning architecture for intelligent mobile cloud computing services. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2019.
- [91] M. Etemadi, M. Ghobaei-Arani, and A. Shahidinejad. A learning-based resource provisioning approach in the fog computing environment. *Journal of Experimental & Theoretical Artificial Intelligence*, pages 1–24, 2020.
- [92] M. Etemadi, M. Ghobaei-Arani, and A. Shahidinejad. A cost-efficient auto-scaling mechanism for IoT applications in fog computing environment: a deep learning-based approach. *Cluster Computing*, pages 1–16, 2021.
- [93] H. Feng, Y. Jiang, D. Niyato, F.-C. Zheng, and X. You. Content popularity prediction via deep learning in cache-enabled fog radio access networks. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2019.
- [94] Y. Feng, Z. Liu, J. Chen, H. Lv, J. Wang, and J. Yuan. Make the Rocket Intelligent at IoT Edge: Stepwise GAN for Anomaly Detection of LRE with Multi-source Fusion. *IEEE Internet of Things Journal*, 2021.
-

- [95] C. Findling, V. Skvortsova, R. Dromnelle, S. Palminteri, and V. Wyart. Computational noise in reward-guided learning drives behavioral variability in volatile environments. *Nature neuroscience*, 22(12):2066–2077, 2019.
- [96] M. Firdhous, O. Ghazali, and S. Hassan. Fog computing: Will it be the future of cloud computing? In *International Conference on Informatics and Applications*, 2014.
- [97] G. Fox, J. Glazier, J. Kadupitiya, V. Jadhao, M. Kim, J. Qiu, J. P. Sluka, E. Somogy, M. Marathe, A. Adiga, et al. Learning everywhere: Pervasive machine learning for effective high-performance computation. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 422–429. IEEE, 2019.
- [98] U. Franke. Optimal it service availability: Shorter outages, or fewer? *IEEE Transactions on Network and Service Management*, 9(1):22–33, 2011.
- [99] Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059. PMLR, 2016.
- [100] Y. Gan, S. Dev, D. Lo, and C. Delimitrou. Sage: Leveraging ML To Diagnose Unpredictable Performance in Cloud Microservices. *ML for Computer Architecture and Systems*, 2020.
- [101] P. Gazori, D. Rahbari, and M. Nickray. Saving time and cost on the scheduling of fog-based iot applications using deep reinforcement learning approach. *Future Generation Computer Systems*, 2019.
- [102] S. Ghanavati, J. H. Abawajy, and D. Izadi. An energy aware task scheduling model using ant-mating optimization in fog computing environment. *IEEE Transactions on Services Computing*, 2020.
- [103] G. R. Ghosal, D. Ghosal, A. Sim, A. V. Thakur, and K. Wu. A deep deterministic policy gradient based network scheduler for deadline-driven data transfers. In *2020 IFIP Networking Conference (Networking)*, pages 253–261. IEEE, 2020.
- [104] S. S. Gill, S. Tuli, M. Xu, I. Singh, K. V. Singh, D. Lindsay, S. Tuli, D. Smirnova, M. Singh, U. Jain, et al. Transformative effects of IoT, Blockchain and Artificial Intelligence on cloud computing: Evolution, vision, trends and open challenges. *Internet of Things*, 8:100–118, 2019.
- [105] S. S. Gill, S. Tuli, A. N. Toosi, F. Cuadrado, P. Garraghan, R. Bahsoon, H. Lutfiyya, R. Sakellariou, O. Rana, S. Dustdar, et al. ThermoSim: Deep learning based framework for modeling and simulation of thermal-aware resource management for cloud computing environments. *Journal of Systems and Software*, 166:110596, 2020.
- [106] K. Gilly, S. Alcaraz, N. Akin, S. Filiposka, and A. Mishev. Modelling edge computing in urban mobility simulation scenarios. In *2020 IFIP Networking Conference (Networking)*, pages 539–543. IEEE, 2020.
- [107] L. Girish and S. K. Rao. Anomaly detection in cloud environment using artificial intelligence techniques. *Computing*, pages 1–14, 2021.

-
- [108] A. Goli, O. Hajihassani, H. Khazaei, O. Ardakanian, M. Rashidi, and T. Dauphinee. Migrating from monolithic to serverless: A fintech case study. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, pages 20–25, 2020.
- [109] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe. Co-simulation: a survey. *ACM Computing Surveys (CSUR)*, 51(3):1–33, 2018.
- [110] C. Gong, Z. Jiang, D. Wang, Y. Lin, Q. Liu, and D. Z. Pan. Mixed precision neural architecture search for energy efficient deep learning. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7. IEEE, 2019.
- [111] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.
- [112] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.
- [113] V. S. Gordon and J. Crouson. Self-splitting modular neural network-domain partitioning at boundaries of trained regions. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 1085–1091. IEEE, 2008.
- [114] T. Gouasmi, W. Louati, and A. H. Kacem. Exact and heuristic mapreduce scheduling algorithms for cloud federation. *Computers & Electrical Engineering*, 69:274–286, 2018.
- [115] M. Grinberg. *Flask web development: developing web applications with python*. “O’Reilly Media, Inc.”, 2018.
- [116] J. Grover and R. M. Garimella. Reliable and fault-tolerant iot-edge architecture. In *2018 IEEE sensors*, pages 1–4. IEEE, 2018.
- [117] J. Gu, X. Wu, B. Zhu, Y. Xia, B. Zang, H. Guan, and H. Chen. Enclavisor: A hardware-software co-design for enclaves on untrusted cloud. *IEEE Transactions on Computers*, 2020.
- [118] L. Gu, D. Zeng, S. Guo, A. Barnawi, and Y. Xiang. Cost efficient resource management in fog computing supported medical cyber-physical system. *IEEE Transactions on Emerging Topics in Computing*, 5(1): 108–119, 2015.
- [119] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 443–462, 2020.
- [120] J. R. Gunasekaran, C. S. Mishra, P. Thinakaran, M. T. Kandemir, and C. R. Das. Implications of public cloud resource heterogeneity for inference serving. In *International Workshop on Serverless Computing*, pages 7–12, 2020.
- [121] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya. iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.
-

- [122] S. Gupta and D. A. Dinesh. Resource usage prediction of cloud workloads using deep bidirectional long short term memory networks. In *2017 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*, pages 1–6. IEEE, 2017.
- [123] S. Gupta, G. Joshi, and O. Yağın. Correlated multi-armed bandits with a latent random source. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3572–3576, 2020.
- [124] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim. Musical chair: Efficient real-time recognition using collaborative iot devices. *arXiv preprint arXiv:1802.02138*, 2018.
- [125] A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for byzantine fault detection. In *HotDep*, 2006.
- [126] J. Hamer, M. Mohri, and A. T. Suresh. Fedboost: A communication-efficient algorithm for federated learning. In *International Conference on Machine Learning*, pages 3973–3983. PMLR, 2020.
- [127] K. Han, Z. Xie, and X. LV. Fog computing task scheduling strategy based on improved genetic algorithm. *Computer Science*, 4:22, 2018.
- [128] C. Hao, J. Dotzel, J. Xiong, L. Benini, Z. Zhang, and D. Chen. Enabling Design Methodologies and Future Trends for Edge AI: Specialization and Co-design. *IEEE Design & Test*, 2021.
- [129] T. Hao, Y. Huang, X. Wen, W. Gao, F. Zhang, C. Zheng, L. Wang, H. Ye, K. Hwang, Z. Ren, et al. Edge aibench: towards comprehensive end-to-end edge computing benchmarking. In *International Symposium on Benchmarking, Measuring and Optimization*, pages 23–30. Springer, 2018.
- [130] F. Haouari, R. Faraj, and J. M. AlJa’am. Fog computing potentials, applications, and challenges. In *2018 International Conference on Computer and Applications (ICCA)*, pages 399–406. IEEE, 2018.
- [131] J. Haussmann, W. Blochinger, and W. Kuechlin. Cost-optimized parallel computations using volatile cloud resources. In *International Conference on the Economics of Grids, Clouds, Systems, and Services*, pages 45–53. Springer, 2019.
- [132] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [133] T. He, A. N. Toosi, and R. Buyya. SLA-aware multiple migration planning and scheduling in SDN-NFV-enabled clouds. *Journal of Systems and Software*, page 110943, 2021.
- [134] Z. He, P. Chen, X. Li, Y. Wang, G. Yu, C. Chen, X. Li, and Z. Zheng. A spatiotemporal deep learning approach for unsupervised anomaly detection in cloud systems. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.
- [135] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. In *Workshop on hot topics in cloud computing (HotCloud)*, 2016.
- [136] D. Hendrycks, M. Mazeika, S. Kadavath, and D. Song. *Using Self-Supervised Learning Can Improve Model Robustness and Uncertainty*. Curran Associates Inc., Red Hook, NY, USA, 2019.

-
- [137] K. Hoang. Image classification with fashion-mnist and cifar10. *California State University*, 2018.
- [138] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [139] C.-H. Hong and B. Varghese. Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms. *ACM Computing Surveys (CSUR)*, 52(5):1–37, 2019.
- [140] S. Hosseinalipour, C. G. Brinton, V. Aggarwal, H. Dai, and M. Chiang. From federated to fog learning: Distributed machine learning over heterogeneous wireless networks. *IEEE Communications Magazine*, 58(12):41–47, 2020.
- [141] L. Hou, Z. Yuan, L. Huang, H. Shen, X. Cheng, and C. Wang. Slimmable generative adversarial networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 7746–7753, 2021.
- [142] K.-J. Hsu, K. Bhardwaj, and A. Gavrilovska. Couper: Dnn model slicing for visual analytics containers at the edge. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 179–194, 2019.
- [143] P. Hu, S. Dhelim, H. Ning, and T. Qiu. Survey on fog computing: architecture, key technologies, applications and open issues. *Journal of network and computer applications*, 98:27–42, 2017.
- [144] X. Hu, Y. Li, L. Jia, and M. Qiu. A novel two-stage unsupervised fault recognition framework combining feature extraction and fuzzy clustering for collaborative AIoT. *IEEE Transactions on Industrial Informatics*, 2021.
- [145] Y. Hu, C. de Laat, and Z. Zhao. Learning workflow scheduling on multi-resource clusters. In *2019 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–8. IEEE, 2019.
- [146] J. Huang, C. Samplawski, D. Ganesan, B. Marlin, and H. Kwon. CLIO: Enabling automatic compilation of deep learning pipelines across IoT and Cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–12, 2020.
- [147] S. Huang, Y. Liu, C. Fung, R. He, Y. Zhao, H. Yang, and Z. Luan. HitAnomaly: Hierarchical Transformers for Anomaly Detection in System Log. *IEEE Transactions on Network and Service Management*, 17(4):2064–2076, 2020.
- [148] T. Huang, W. Lin, C. Xiong, R. Pan, and J. Huang. An ant colony optimization-based multiobjective service replicas placement strategy for fog computing. *IEEE Transactions on Cybernetics*, 2020.
- [149] X. Huang and S. Belongie. Arbitrary style transfer in real-time with adaptive instance normalization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1501–1510, 2017.
- [150] Y. Huang, X. Qiao, J. Tang, P. Ren, L. Liu, C. Pu, and J. Chen. DeepAdapter: A Collaborative Deep Learning Framework for the Mobile Web Using Context-Aware Network Pruning. *Proceedings - IEEE INFOCOM*, 2020-July:834–843, 7 2020. doi: 10.1109/INFOCOM41043.2020.9155379.
- [151] K. Hundman, V. Constantinou, C. Laporte, I. Colwell, and T. Soderstrom. Detecting spacecraft anomalies using LSTMs and nonparametric dynamic thresholding. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 387–395, 2018.
-

- [152] K. Hwang, J. Dongarra, and G. C. Fox. Distributed and cloud computing: from parallel processing to the internet of things. 2013.
- [153] R. J. Hyndman and G. Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2018.
- [154] M. Imani, S. Gupta, Y. Kim, and T. Rosing. Floatpim: In-memory acceleration of deep neural network training with high precision. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 802–815. IEEE, 2019.
- [155] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.
- [156] G. Ismayilov and H. R. Topcuoglu. Neural network based multi-objective evolutionary algorithm for dynamic workflow scheduling in cloud computing. *Future Generation computer systems*, 102:307–322, 2020.
- [157] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, P. Chuang, and L. Chang. Compensated-dnn: Energy efficient low-precision deep neural networks by compensating quantization errors. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.
- [158] F. Jalali, O. J. Smith, T. Lynar, and F. Suits. Cognitive IoT gateways: automatic task sharing and switching between cloud and edge/fog computing. In *Proceedings of the SIGCOMM Posters and Demos*, pages 121–123. 2017.
- [159] F. Jalali, T. Lynar, O. J. Smith, R. R. Kolluri, C. V. Hardgrove, N. Waywood, and F. Suits. Dynamic edge fabric environment: Seamless and automatic switching among resources at the edge of IoT network and cloud. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 77–86. IEEE, 2019.
- [160] P. Jamshidi and G. Casale. An uncertainty-aware approach to optimal configuration of stream processing systems. In *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 39–48. IEEE, 2016.
- [161] M. A. Jan, M. Zakarya, M. Khan, S. Mastorakis, V. G. Menon, V. Balasubramanian, and A. U. Rehman. An AI-enabled lightweight data fusion and load optimization approach for Internet of Things. *Future Generation Computer Systems*, 122:40–51, 2021.
- [162] E. Jang, S. Gu, and B. Poole. Categorical reparameterization with gumbel-softmax. *International Conference on Learning on Learning Representations (ICLR)*, 2017.
- [163] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.
- [164] B. Javadi, J. Sun, and R. Ranjan. Serverless architecture for edge computing. *Edge Computing: Models, Technologies and Applications*, pages 249–264, 2020.
- [165] A. Javed, K. Heljanko, A. Buda, and K. Främbling. CEFIoT: A fault-tolerant iot architecture for edge and cloud. In *2018 IEEE 4th world forum on internet of things (WF-IoT)*, pages 813–818. IEEE, 2018.

-
- [166] M. Jawad, M. B. Qureshi, U. Khan, S. M. Ali, A. Mehmood, B. Khan, X. Wang, and S. U. Khan. A robust optimization technique for energy cost minimization of cloud data centers. *IEEE Transactions on Cloud Computing*, 2018.
- [167] S. Jeddi and S. Sharifian. A water cycle optimized wavelet neural network algorithm for demand prediction in cloud computing. *Cluster Computing*, 22(4):1397–1412, 2019.
- [168] Z. Ji, C. Chen, J. He, S. Zhu, and X. Guan. Learning-based edge sensing and control co-design for industrial cyber-physical system. *IEEE Transactions on Automation Science and Engineering*, 2021.
- [169] F. Jiang, K. Wang, L. Dong, C. Pan, W. Xu, and K. Yang. Deep-learning-based joint resource scheduling algorithms for hybrid MEC networks. *IEEE Internet of Things Journal*, 7(7):6252–6265, 2019.
- [170] Y. Jiang, S. Wang, V. Valls, B. J. Ko, W.-H. Lee, K. K. Leung, and L. Tassiulas. Model pruning enables efficient federated learning on edge devices. *arXiv preprint arXiv:1909.12326*, 2019.
- [171] I. Kadota, A. Sinha, E. Uysal-Biyikoglu, R. Singh, and E. Modiano. Scheduling policies for minimizing age of information in broadcast wireless networks. *IEEE/ACM Transactions on Networking*, 26(6):2637–2650, 2018.
- [172] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017.
- [173] A. Kaplunovich and Y. Yesha. Automatic tuning of hyperparameters for neural networks in serverless cloud. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 2751–2756. IEEE, 2020.
- [174] A. Kaur, P. Singh, R. Singh Batth, and C. Peng Lim. Deep-Q learning-based heterogeneous earliest finish time scheduling algorithm for scientific workflows in cloud. *Software: Practice and Experience*, 2020.
- [175] M. Kaur and R. Aron. Focalb: Fog computing architecture of load balancing for scientific workflow applications. *Journal of Grid Computing*, 19(4):1–22, 2021.
- [176] M. J. Kaur, V. P. Mishra, and P. Maheshwari. The convergence of digital twin, iot, and machine learning: transforming data into action. In *Digital twin technologies and smart cities*, pages 3–17. Springer, 2020.
- [177] A. Kendall and Y. Gal. What uncertainties do we need in bayesian deep learning for computer vision? In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 5580–5590, 2017.
- [178] J. Khamse-Ashari, I. Lambadaris, G. Kesidis, B. Urgaonkar, and Y. Zhao. An efficient and fair multi-resource allocation mechanism for heterogeneous servers. *IEEE Transactions on Parallel and Distributed Systems*, 29(12):2686–2699, 2018.
- [179] A. Khanna, A. Sah, and T. Choudhury. Intelligent mobile edge computing: A deep learning based approach. In *International Conference on Advances in Computing and Data Sciences*, pages 107–116. Springer, 2020.
-

- [180] J. Kim, Y. Park, G. Kim, and S. J. Hwang. Splitnet: Learning to semantically split deep networks for parameter reduction and model parallelization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1866–1874. JMLR. org, 2017.
- [181] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [182] M. J. Kochenderfer and T. A. Wheeler. *Algorithms for optimization*. Mit Press, 2019.
- [183] R. Kohavi and R. Longbotham. Online controlled experiments and a/b testing. *Encyclopedia of machine learning and data mining*, 7(8):922–929, 2017.
- [184] X. Kong, X. Liu, B. Jedari, M. Li, L. Wan, and F. Xia. Mobile crowdsourcing in smart cities: Technologies, applications, and future challenges. *IEEE Internet of Things Journal*, 6(5):8095–8113, 2019.
- [185] F. A. Kraemer, A. E. Braten, N. Tamkittikhun, and D. Palma. Fog computing in healthcare—a review and discussion. *IEEE Access*, 5:9206–9222, 2017.
- [186] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker. Recent development and applications of sumo-simulation of urban mobility. *International journal on advances in systems and measurements*, 5(3&4), 2012.
- [187] S. Krishnasamy, P. Akhil, A. Arapostathis, R. Sundaresan, and S. Shakkottai. Augmenting max-weight with explicit learning for wireless scheduling with switching costs. *IEEE/ACM Transactions on Networking*, 26(6):2501–2514, 2018.
- [188] A. Krizhevsky et al. Learning multiple layers of features from tiny images. *University of Toronto, Technical Report*, 2009.
- [189] P. Kumari and P. Kaur. A survey of fault tolerance in cloud computing. *Journal of King Saud University-Computer and Information Sciences*, 2018.
- [190] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–12. IEEE, 2016.
- [191] H. F. Langroudi, Z. Carmichael, and D. Kudithipudi. Deep learning training on the edge with low-precision posits. *arXiv preprint arXiv:1907.13216*, 2019.
- [192] H. F. Langroudi, Z. Carmichael, D. Pastuch, and D. Kudithipudi. Cheetah: Mixed low-precision hardware & software co-design framework for dnns on the edge. *arXiv preprint arXiv:1908.02386*, 2019.
- [193] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane. SPINN: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2020.
- [194] Q. Le, L. Miralles-Pechuán, S. Kulkarni, J. Su, and O. Boydell. An overview of deep learning in industry. *Data Analytics and AI*, pages 65–98, 2020.

-
- [195] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
 - [196] Y. LeCun, S. Chopra, M. Ranzato, and F.-J. Huang. Energy-based models in document recognition and computer vision. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, volume 1, pages 337–341. IEEE, 2007.
 - [197] J. B. Lee, R. A. Rossi, S. Kim, N. K. Ahmed, and E. Koh. Attention models in graphs: A survey. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 13(6):1–25, 2019.
 - [198] K. B. Letaief, W. Chen, Y. Shi, J. Zhang, and Y.-J. A. Zhang. The roadmap to 6G: AI empowered wireless networks. *IEEE Communications Magazine*, 57(8):84–90, 2019.
 - [199] S. Levy, R. Yao, Y. Wu, Y. Dang, P. Huang, Z. Mu, P. Zhao, T. Ramani, N. Govindaraju, X. Li, et al. Narya: Predictive and adaptive failure mitigation to avert production cloud vm interruptions. In *Symposium on Operating Systems Design and Implementation (OSDI’20)*, 2020.
 - [200] C. Li, M. Cerrada, D. Cabrera, R. V. Sanchez, F. Pacheco, G. Ulutagay, and J. Valente de Oliveira. A comparison of fuzzy clustering algorithms for bearing fault diagnosis. *Journal of Intelligent & Fuzzy Systems*, 34(6):3565–3580, 2018.
 - [201] C. Li, J. Bai, Y. Ge, and Y. Luo. Heterogeneity-aware elastic provisioning in cloud-assisted edge computing systems. *Future Generation Computer Systems*, 112:1106–1121, 2020.
 - [202] D. Li, D. Chen, B. Jin, L. Shi, J. Goh, and S.-K. Ng. MAD-GAN: Multivariate anomaly detection for time series data with generative adversarial networks. In *International Conference on Artificial Neural Networks*, pages 703–716. Springer, 2019.
 - [203] E. Li, L. Zeng, Z. Zhou, and X. Chen. Edge AI: On-demand accelerating deep neural network inference via edge computing. *IEEE Transactions on Wireless Communications*, 19(1):447–457, 2019.
 - [204] G. Li, X. Zhou, J. Sun, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li. opengauss: An autonomous database system. *Proceedings of the VLDB Endowment*, 14(12):3028–3042, 2021.
 - [205] H. Li, K. Ota, and M. Dong. Deep reinforcement scheduling for mobile crowdsensing in fog computing. *ACM Transactions on Internet Technology (TOIT)*, 19(2):1–18, 2019.
 - [206] P. Li, H. Seferoglu, V. R. Dasari, and E. Koyuncu. Model-distributed dnn training for memory-constrained edge computing devices. In *2021 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–6. IEEE, 2021.
 - [207] R. Li, Z. Zhao, X. Zhou, G. Ding, Y. Chen, Z. Wang, and H. Zhang. Intelligent 5G: When cellular networks meet artificial intelligence. *IEEE Wireless communications*, 24(5):175–183, 2017.
 - [208] S. Li and T. Lan. Hotdedup: Managing hot data storage at network edge through optimal distributed deduplication. In *IEEE Conference on Computer Communications*, pages 247–256. IEEE, 2020.
 - [209] S. Li, L. Wang, W. Wang, Y. Yu, and B. Li. George: Learning to place long-lived containers in large clusters with operation constraints. In *ACM Symposium on Cloud Computing*, pages 258–272, 2021.
-

- [210] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3):50–60, 2020.
- [211] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.
- [212] Z. Li, C. Yan, X. Yu, and N. Yu. Bayesian network-based virtual machines consolidation method. *Future Generation Computer Systems*, 69:75–87, 2017.
- [213] Q. Liang, P. Shenoy, and D. Irwin. AI on the edge: Characterizing AI-based IoT applications using specialized edge architectures. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 145–156. IEEE, 2020.
- [214] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.
- [215] W. Y. B. Lim, N. C. Luong, D. T. Hoang, Y. Jiao, Y.-C. Liang, Q. Yang, D. Niyato, and C. Miao. Federated learning in mobile edge networks: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22(3):2031–2063, 2020.
- [216] B. Lin, F. Zhu, J. Zhang, J. Chen, X. Chen, N. N. Xiong, and J. L. Mauri. A time-driven data placement strategy for a scientific workflow combining edge computing and cloud computing. *IEEE Transactions on Industrial Informatics*, 15(7):4254–4265, 2019.
- [217] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.
- [218] G. Liu, F. A. Reda, K. J. Shih, T.-C. Wang, A. Tao, and B. Catanzaro. Image inpainting for irregular holes using partial convolutions. In *The European Conference on Computer Vision (ECCV)*, 2018.
- [219] J. Liu and M. Leon. Scale-out edge storage systems with embedded storage nodes to get better availability and cost-efficiency at the same time. *Workshop on Hot Topics in Edge Computing*, 2020.
- [220] J. Liu, S. Wang, A. Zhou, S. A. Kumar, F. Yang, and R. Buyya. Using proactive fault-tolerance approach to enhance cloud service reliability. *IEEE Transactions on Cloud Computing*, 6(4):1191–1202, 2016.
- [221] S. Liu and N. Wang. Collaborative optimization scheduling of cloud service resources based on improved genetic algorithm. *IEEE Access*, 8:150878–150890, 2020.
- [222] X. Liu, B. Li, P. Shi, and L. Ying. POND: Pessimistic-Optimistic oNline Dispatch. *arXiv preprint arXiv:2010.09995*, 2020.
- [223] Y. Liu, B. Cao, and H. Li. Improving ant colony optimization algorithm with epsilon greedy and levy flight. *Complex & Intelligent Systems*, pages 1–12, 2020.
- [224] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. Learning efficient convolutional networks through network slimming. In *IEEE international conference on computer vision*, pages 2736–2744, 2017.

-
- [225] L. Lockhart, P. Harvey, P. Imai, P. Willis, and B. Varghese. Scission: Performance-driven and context-aware cloud-edge distribution of deep neural networks. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 257–268. IEEE, 2020.
- [226] I. Loshchilov and F. Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.
- [227] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2018.
- [228] C. Luo, F. Zhang, C. Huang, X. Xiong, J. Chen, L. Wang, W. Gao, H. Ye, T. Wu, R. Zhou, et al. AIoT bench: towards comprehensive benchmarking mobile and embedded device intelligence. In *International Symposium on Benchmarking, Measuring and Optimization*, pages 31–35. Springer, 2018.
- [229] C. Luo, B. Qiao, X. Chen, P. Zhao, R. Yao, H. Zhang, W. Wu, A. Zhou, and Q. Lin. Intelligent virtual machine provisioning in cloud computing. In *International Joint Conference on AI*, pages 1495–1502, 2020.
- [230] C. Luo, B. Qiao, W. Xing, X. Chen, P. Zhao, C. Du, R. Yao, H. Zhang, W. Wu, S. Cai, et al. Correlation-aware heuristic search for intelligent virtual machine provisioning in cloud systems. In *AAAI Conference on AI*, volume 35, pages 12363–12372, 2021.
- [231] P. Luo, F. R. Yu, J. Chen, J. Li, and V. C. Leung. A novel adaptive gradient compression scheme: Reducing the communication overhead for distributed deep learning in the internet of things. *IEEE Internet of Things Journal*, 2021.
- [232] W. Luping, W. Wei, and L. Bo. CMFL: Mitigating communication overhead for federated learning. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 954–964. IEEE, 2019.
- [233] M. M. Mahmoud, J. J. Rodrigues, K. Saleem, J. Al-Muhtadi, N. Kumar, and V. Korotaev. Towards energy-aware fog-enabled cloud of things for healthcare. *Computers & Electrical Engineering*, 67:58–69, 2018.
- [234] R. Mahmud, R. Kotagiri, and R. Buyya. Fog computing: A taxonomy, survey and future directions. In *Internet of everything*, pages 103–130. Springer, 2018.
- [235] R. Mahmud, K. Ramamohanarao, and R. Buyya. Application management in fog computing environments: A taxonomy, review and future directions. *ACM Computing Surveys (CSUR)*, 53(4):1–43, 2020.
- [236] R. Mahmud, S. N. Srirama, K. Ramamohanarao, and R. Buyya. Profit-aware application placement for integrated fog-cloud computing environments. *Journal of Parallel and Distributed Computing*, 135: 177–190, 2020.
- [237] R. Mahmud, S. Pallewatta, M. Goudarzi, and R. Buyya. Ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments. *arXiv preprint arXiv:2109.05636*, 2021.
-

- [238] A. A. Majeed, P. Kilpatrick, I. Spence, and B. Varghese. Continuer: maintaining distributed dnn services during edge failures. In *IEEE International Conference on Edge Computing and Communications (EDGE)*, pages 143–152. IEEE, 2022.
- [239] S. Malik and F. Huet. Adaptive fault tolerance in real time cloud computing. In *2011 IEEE World Congress on services*, pages 280–287. IEEE, 2011.
- [240] N. Mansouri, R. Ghafari, and B. M. H. Zade. Cloud computing simulators: A comprehensive review. *Simulation Modelling Practice and Theory*, 104:102144, 2020.
- [241] Y. Mao, J. Zhang, and K. B. Letaief. Dynamic computation offloading for mobile-edge computing with energy harvesting devices. *IEEE Journal on Selected Areas in Communications*, 34(12):3590–3605, 2016.
- [242] A. Marahatta, Q. Xin, C. Chi, F. Zhang, and Z. Liu. PEFS: AI-driven Prediction based Energy-aware Fault-tolerant Scheduling Scheme for Cloud Data Center. *IEEE Transactions on Sustainable Computing*, 2020.
- [243] A. Maroti. RBED: Reward Based Epsilon Decay. *arXiv preprint arXiv:1910.13701*, 2019.
- [244] J. Martens and R. Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.
- [245] T. Matiisen, A. Oliver, T. Cohen, and J. Schulman. Teacher–student curriculum learning. *IEEE transactions on neural networks and learning systems*, 31(9):3732–3740, 2019.
- [246] K. Matrouk and K. Alatoun. Scheduling algorithms in fog computing: A survey. *International Journal of Networked and Distributed Computing*, 2021.
- [247] Y. Matsubara, S. Baidya, D. Callegaro, M. Levorato, and S. Singh. Distilled split deep neural networks for edge-assisted real-time systems. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, pages 21–26, 2019.
- [248] J. McChesney, N. Wang, A. Tanwer, E. de Lara, and B. Varghese. Defog: fog computing benchmarks. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 47–58, 2019.
- [249] C. Medina, A. Devos, and M. Grossglauser. Self-Supervised Prototypical Transfer Learning for Few-Shot Classification. *International Conference on Machine Learning (ICML) - Workshop on Automated Machine Learning*, 2020.
- [250] M.-E. Miranda-Varela and E. Mezura-Montes. Constraint-handling techniques in surrogate-assisted evolutionary optimization. an empirical study. *Applied Soft Computing*, 73:215–229, 2018.
- [251] M. Mladenov, C.-w. Hsu, V. Jain, E. Ie, C. Colby, N. Mayoraz, H. Pham, D. Tran, I. Vendrov, and C. Boutilier. Demonstrating principled uncertainty modeling for recommender ecosystems with recsimng. In *Fourteenth ACM Conference on Recommender Systems*, pages 591–593, 2020.
- [252] J. Moon, J. Kim, Y. Shin, and S. Hwang. Confidence-aware learning for deep neural networks. In *International Conference on Machine Learning*, pages 7034–7044. PMLR, 2020.

-
- [253] M. Mudassar, Y. Zhai, L. Liao, and J. Shen. A decentralized latency-aware task allocation and group formation approach with fault tolerance for iot applications. *IEEE Access*, 8:49212–49223, 2020.
 - [254] N. Mustafee, J. Powell, S. C. Brailsford, S. Diallo, J. Padilla, and A. Tolk. Hybrid simulation studies and hybrid simulation systems: definitions, challenges, and benefits. In *2015 Winter Simulation Conference (WSC)*, pages 1678–1692. IEEE, 2015.
 - [255] A. A. Mutlag, M. K. Abd Ghani, N. a. Arunkumar, M. A. Mohammed, and O. Mohd. Enabling technologies for fog computing in healthcare iot systems. *Future Generation Computer Systems*, 90:62–78, 2019.
 - [256] R. K. Naha, S. K. Garg, M. B. Amin, and R. Ranjan. Fuzzy logic-based robust failure handling mechanism for fog computing. *CoRR*, abs/2103.06381, 2021. URL <https://arxiv.org/abs/2103.06381>.
 - [257] S. Nandi, S. Ghosh, S. S. Tambe, and B. D. Kulkarni. Artificial neural-network-assisted stochastic process optimization strategies. *AIChE journal*, 47(1):126–141, 2001.
 - [258] H. Nashaat, N. Ashry, and R. Rizk. Smart elastic scheduling algorithm for virtual machine migration in cloud computing. *The Journal of Supercomputing*, 75(7):3842–3865, 2019.
 - [259] Z. M. Nayeri, T. Ghafarian, and B. Javadi. Application placement in fog computing with ai approach: Taxonomy and a state of the art survey. *Journal of Network and Computer Applications*, 185:103078, 2021.
 - [260] M. Ndiaye, S. S. Oyewobi, A. M. Abu-Mahfouz, G. P. Hancke, A. M. Kurien, and K. Djouani. IoT in the wake of COVID-19: A survey on contributions, challenges and evolution. *IEEE Access*, 8:186821–186839, 2020.
 - [261] S. Negi, M. M. S. Rauthan, K. S. Vaisla, and N. Panwar. CMODLB: an efficient load balancing approach in cloud computing environment. *The Journal of Supercomputing*, pages 1–53, 2021.
 - [262] T. Nguyen, T. Le, H. Zhao, H. Q. Tran, T. Nguyen, and D. Phung. TIDOT: A teacher imitation learning approach for domain adaptation with optimal transport. In *IJCAI*, 2021.
 - [263] T. Nguyen-Thien and T. Tran-Cong. Approximation of functions and their derivatives: A neural network implementation with applications. *Applied Mathematical Modelling*, 23(9):687–704, 1999.
 - [264] B. Nicoletti. Cloud computing. In *Cloud Computing in Financial Services*, pages 29–64. Springer, 2013.
 - [265] M. O’Grady, D. Langton, and G. O’Hare. Edge computing: A tractable model for smart agriculture? *Artificial Intelligence in Agriculture*, 3:42–51, 2019.
 - [266] R. Oma, S. Nakamura, D. Duolikun, T. Enokido, and M. Takizawa. An energy-efficient model for fog computing in the Internet of Things (IoT). *Internet of Things*, 1:14–26, 2018.
 - [267] B. S. Onggo, N. Mustafee, A. Smart, A. A. Juan, and O. Molloy. Symbiotic simulation system: hybrid systems model meets big data analytics. In *2018 Winter Simulation Conference (WSC)*, pages 1358–1369. IEEE, 2018.
-

- [268] B. S. Onggo, C. G. Corlu, A. A. Juan, T. Monks, and R. de la Torre. Combining symbiotic simulation systems with enterprise data storage systems for real-time decision-making. *Enterprise Information Systems*, 15(2):230–247, 2021.
- [269] Open Mainframe Project. Anomaly Detection Engine for Linux Logs (ADE). URL <https://www.openmainframeproject.org/projects/anomaly-detection-engine-for-linux-logs-ade>.
- [270] S. Ouhamme, Y. Hadi, and A. Ullah. An efficient forecasting approach for resource utilization in cloud data center using cnn-lstm model. *Neural Computing and Applications*, pages 1–13, 2021.
- [271] J. K. O’herrin, N. Fost, and K. A. Kudsk. Health insurance portability accountability act (hipaa) regulations: effect on medical record research. *Annals of surgery*, 239(6):772, 2004.
- [272] R. G. Pacheco, K. Bochie, M. S. Gilbert, R. S. Couto, and M. E. M. Campista. Towards edge computing using early-exit convolutional neural networks. *Information*, 12(10):431, 2021.
- [273] S. Panda, B. Neha, and S. Sathua. An uncertainty-based task scheduling for heterogeneous multi-cloud systems. *International Journal of Information Processing*, 9(2):13–24, 2015.
- [274] A. Panesar. *Machine learning and AI for healthcare*. Springer, 2019.
- [275] D. Park, Y. Hoshi, and C. C. Kemp. A multimodal anomaly detector for robot-assisted feeding using an LSTM-based variational autoencoder. *IEEE Robotics and Automation Letters*, 3(3):1544–1551, 2018.
- [276] D. Park, S. Kim, Y. An, and J.-Y. Jung. LiReD: A light-weight real-time fault detection system for edge computing using LSTM recurrent neural networks. *Sensors*, 18(7):2110, 2018.
- [277] S.-H. Park, O. Simeone, and S. Shamai. Joint cloud and edge processing for latency minimization in fog radio access networks. In *2016 IEEE 17th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 1–5. IEEE, 2016.
- [278] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.
- [279] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimeshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8026–8037, 2019.
- [280] M. G. Pecht and M. Kang. Machine learning: Anomaly detection. 2019.
- [281] C. Perera, Y. Qin, J. C. Estrella, S. Reiff-Marganiec, and A. V. Vasilakos. Fog computing for sustainable smart cities: A survey. *ACM Computing Surveys (CSUR)*, 50(3):1–43, 2017.
- [282] T.-P. Pham and T. Fahringer. Evolutionary multi-objective workflow scheduling for volatile resources in the cloud. *IEEE Transactions on Cloud Computing*, 2020.
- [283] A. Polino, R. Pascanu, and D.-A. Alistarh. Model compression via distillation and quantization. In *6th International Conference on Learning Representations*, 2018.

-
- [284] J. Postels, F. Ferroni, H. Coskun, N. Navab, and F. Tombari. Sampling-free epistemic uncertainty estimation using approximated variance propagation. In *IEEE/CVF International Conference on Computer Vision*, pages 2931–2940, 2019.
- [285] L. Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998.
- [286] D. Preuveneers, V. Rimmer, I. Tsingenopoulos, J. Spooren, W. Joosen, and E. Ilie-Zudor. Chained anomaly detection models for federated learning: An intrusion detection case study. *Applied Sciences*, 8(12):2663, 2018.
- [287] W. Rafique, L. Qi, I. Yaqoob, M. Imran, R. U. Rasool, and W. Dou. Complementing IoT services through software defined networking and edge computing: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22(3):1761–1804, 2020.
- [288] A. Rakitianskaia and A. Engelbrecht. Measuring saturation in neural networks. In *2015 IEEE Symposium Series on Computational Intelligence*, pages 1423–1430. IEEE, 2015.
- [289] A. Rawat, R. Sushil, A. Agarwal, A. Sikander, and R. S. Bhadoria. A new adaptive fault tolerant framework in the cloud. *IETE Journal of Research*, pages 1–13, 2021.
- [290] B. Ray, A. Saha, S. Khatua, and S. Roy. Proactive fault-tolerance technique to enhance reliability of cloud service in cloud federation environment. *IEEE Transactions on Cloud Computing*, 2020.
- [291] K. Rehman, O. Kipouridis, S. Karnouskos, O. Frendo, H. Dickel, J. Lipps, and N. Verzano. A cloud-based development environment using hla and kubernetes for the co-simulation of a corporate electric vehicle fleet. In *2019 IEEE/SICE International Symposium on System Integration (SII)*, pages 47–54. IEEE, 2019.
- [292] A. Reisizadeh, A. Mokhtari, H. Hassani, A. Jadbabaie, and R. Pedarsani. Fedpaq: A communication-efficient federated learning method with periodic averaging and quantization. In *International Conference on Artificial Intelligence and Statistics*, pages 2021–2031. PMLR, 2020.
- [293] A. Renda, Y. Chen, C. Mendis, and M. Carbin. DiffTune: Optimizing CPU simulator parameters with learned differentiable surrogates. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 442–455. IEEE, 2020.
- [294] F. Rezaei, A. A. Najafi, and R. Ramezani. Mean-conditional value at risk model for the stochastic project scheduling problem. *Computers & Industrial Engineering*, 142:106356, 2020.
- [295] L. M. Rios and N. V. Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293, 2013.
- [296] S. Ristov, T. Fahringer, D. Peer, T.-P. Pham, M. Gusev, and C. Mas-Machuca. Resilient techniques against disruptions of volatile cloud resources. In *Guide to Disaster-Resilient Communication Networks*, pages 379–400. Springer, 2020.
-

- [297] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, et al. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):4–1, 2009.
- [298] D. Rolnick, A. Ahuja, J. Schwarz, T. Lillicrap, and G. Wayne. Experience replay for continual learning. *Advances in Neural Information Processing Systems*, 32, 2019.
- [299] R. N. Rudd-Orthner and L. Mihaylova. Repeatable determinism using non-random weight initialisations in smart city applications of deep learning. *Journal of Reliable Intelligent Environments*, 6(1):31–49, 2020.
- [300] L. Ruiz, F. Gama, and A. Ribeiro. Gated graph recurrent neural networks. *IEEE Transactions on Signal Processing*, 68:6303–6318, 2020.
- [301] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009. ISBN 0136042597.
- [302] A. Saeed, F. D. Salim, T. Ozcelebi, and J. Lukkien. Federated self-supervised learning of multisensor representations for embedded intelligence. *IEEE Internet of Things Journal*, 8(2):1030–1040, 2020.
- [303] U. Sahlin, I. Helle, and D. Perepolkin. “this is what we don’t know”: Treating epistemic uncertainty in bayesian networks for risk assessment. *Integrated Environmental Assessment and Management*, 17(1): 221–232, 2021.
- [304] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 168–178. IEEE, 2016.
- [305] N. Saleh and M. Mashaly. A dynamic simulation environment for container-based cloud data centers using containercloudsim. In *2019 Ninth International Conference on Intelligent Computing and Information Systems (ICICIS)*, pages 332–336. IEEE, 2019.
- [306] A. Salehi-Amiri, A. Jabbarzadeh, A. Zahedi, N. Akbarpour, and M. Hajiaghahi-Keshteli. Relief Supply Chain Management Using Internet of Things to Address COVID-19 Outbreak. *Computers & Industrial Engineering*, page 107429, 2021.
- [307] A. Samanta, Y. Li, and F. Esposito. Battle of microservices: Towards latency-optimal heuristic scheduling for edge computing. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 223–227. IEEE, 2019.
- [308] N. Samarji and M. Salamah. A fault tolerance metaheuristic-based scheme for controller placement problem in wireless software-defined networks. *International Journal of Communication Systems*, 34(4): e4624, 2021.
- [309] H. Sami, H. Otrók, J. Bentahar, and A. Mourad. Ai-based resource provisioning of ioe services in 6g: A deep reinforcement learning approach. *IEEE Transactions on Network and Service Management*, 2021.

-
- [310] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.
 - [311] C. H. d. Santos, J. A. de Queiroz, F. Leal, and J. A. B. Montevechi. Use of simulation in the industry 4.0 context: Creation of a digital twin to optimise decision making on non-automated process. *Journal of Simulation*, pages 1–14, 2020.
 - [312] A. Satpathy, S. K. Addya, A. K. Turuk, B. Majhi, and G. Sahoo. Crow search based virtual machine placement strategy in cloud data centers with live migration. *Computers & Electrical Engineering*, 69: 334–350, 2018.
 - [313] F. Sattler, S. Wiedemann, K.-R. Müller, and W. Samek. Robust and communication-efficient federated learning from non-iid data. *IEEE transactions on neural networks and learning systems*, 31(9):3400–3413, 2019.
 - [314] M. Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.
 - [315] F. Scarselli and A. C. Tsoi. Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results. *Neural networks*, 11(1):15–37, 1998.
 - [316] P. Schulz, M. Matthe, H. Klessig, M. Simsek, G. Fettweis, J. Ansari, S. A. Ashraf, B. Almeroth, J. Voigt, I. Riedel, et al. Latency critical IoT applications in 5G: Perspective on the design of radio interface and network architecture. *IEEE Communications Magazine*, 55(2):70–78, 2017.
 - [317] A. Semenov. Elastic computing self-organizing for artificial intelligence space exploration. In *Journal of Physics: Conference Series*, volume 1925, page 012071. IOP Publishing, 2021.
 - [318] M. H. Shaker and E. Hüllermeier. Aleatoric and epistemic uncertainty with random forests. In *International Symposium on Intelligent Data Analysis*, pages 444–456. Springer, 2020.
 - [319] J. Shao and J. Zhang. Bottlenet++: An end-to-end approach for feature compression in device-edge co-inference systems. In *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6. IEEE, 2020.
 - [320] J. Shao and J. Zhang. Communication-computation trade-off in resource-constrained edge inference. *IEEE Communications Magazine*, 58(12):20–26, 2020.
 - [321] Y. Shao, C. Li, Z. Fu, L. Jia, and Y. Luo. Cost-effective replication management and scheduling in edge computing. *Journal of Network and Computer Applications*, 129:46–61, 2019.
 - [322] A. Sharif, M. Nickray, and A. Shahidinejad. Fault-tolerant with load balancing scheduling in a fog-based IoT application. *IET Communications*, 14(16):2646–2657, 2020.
 - [323] B. Sharma, L. Sharma, and C. Lal. Anomaly detection techniques using deep learning in IoT: A survey. In *2019 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE)*, pages 146–149. IEEE, 2019.
-

- [324] S. Shen, V. van Beek, and A. Iosup. Statistical characterization of business-critical workloads hosted in cloud datacenters. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 465–474. IEEE, 2015.
- [325] S. Sheng, P. Chen, Z. Chen, L. Wu, and Y. Yao. Deep reinforcement learning-based task scheduling in iot edge computing. *Sensors*, 21(5):1666, 2021.
- [326] Y. Shi, K. Yang, T. Jiang, J. Zhang, and K. B. Letaief. Communication-efficient edge AI: Algorithms and systems. *IEEE Communications Surveys & Tutorials*, 22(4):2167–2191, 2020.
- [327] A. Shukla, G. S. Cheema, and S. Anand. Semi-supervised clustering with neural networks. In *2020 IEEE Sixth International Conference on Multimedia Big Data (BigMM)*, pages 152–161. IEEE, 2020.
- [328] A. Siffer, P.-A. Fouque, A. Termier, and C. Largouet. Anomaly detection in streams with extreme value theory. In *ACM International Conference on Knowledge Discovery and Data Mining*, pages 1067–1075, 2017.
- [329] D. Silver and G. Tesauro. Monte-carlo simulation balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 945–952, 2009.
- [330] P. Singh, P. Gupta, and K. Jyoti. TASM: Technocrat ARIMA and SVR model for workload prediction of web applications in cloud. *Cluster Computing*, 22(2):619–633, 2019.
- [331] Y. Siriwardhana, P. Porambage, M. Liyanage, and M. Ylianttila. A Survey on Mobile Augmented Reality With 5G Mobile Edge Computing: Architectures, Applications, and Technical Aspects. *IEEE Communications Surveys & Tutorials*, 23(2):1160–1192, 2021.
- [332] V. Sivagami and K. Easwarakumar. An improved dynamic fault tolerant management algorithm during vm migration in cloud data center. *Future Generation Computer Systems*, 98:35–43, 2019.
- [333] J. Snell, K. Swersky, and R. Zemel. Prototypical networks for few-shot learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 4080–4090, 2017.
- [334] M. Sohani and S. Jain. A predictive priority-based dynamic resource provisioning scheme with load balancing in heterogeneous cloud computing. *IEEE Access*, 9:62653–62664, 2021.
- [335] Z. Song and E. Tilevich. Win with what you have: Qos-consistent edge services with unreliable and dynamic resources. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 530–540. IEEE, 2020.
- [336] Z. Song, Y. Hao, Y. Liu, and X. Sun. Energy-efficient multiaccess edge computing for terrestrial-satellite internet of things. *IEEE Internet of Things Journal*, 8(18):14202–14218, 2021.
- [337] Standard Performance Evaluation Corporation. SPEC Power Consumption Models. URL https://www.spec.org/cloud_iaas2018/results/.
- [338] P. J. Stuckey, T. Guns, J. Bailey, C. Leckie, K. Ramamohanarao, J. Chan, et al. Dynamic programming for predict+optimise. In *AAAI*, volume 34, pages 1444–1451, 2020.

-
- [339] Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei. Robust anomaly detection for multivariate time series through stochastic recurrent neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2828–2837, 2019.
 - [340] W. Sun, A. Venkatraman, G. J. Gordon, B. Boots, and J. A. Bagnell. Deeply aggravated: Differentiable imitation learning for sequential prediction. In *International Conference on Machine Learning*, pages 3309–3318, 2017.
 - [341] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
 - [342] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI Conference on Artificial Intelligence*, volume 31, 2017.
 - [343] F. M. Talaat, S. H. Ali, A. I. Saleh, and H. A. Ali. Effective load balancing strategy (ELBS) for real-time fog computing environment using fuzzy and probabilistic neural networks. *Journal of Network and Systems Management*, 27(4):883–929, 2019.
 - [344] F. M. Talaat, M. S. Saraya, A. I. Saleh, H. A. Ali, and S. H. Ali. A load balancing and optimization strategy (LBOS) using reinforcement learning in fog computing environment. *Journal of Ambient Intelligence and Humanized Computing*, pages 1–16, 2020.
 - [345] A. S. Tanenbaum and A. S. Woodhull. *Operating systems: design and implementation*, volume 68. Prentice Hall Englewood Cliffs, 1997.
 - [346] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao. Migration modeling and learning algorithms for containers in fog computing. *IEEE Transactions on Services Computing*, 12(5):712–725, 2018.
 - [347] F. Tao, Q. Qi, L. Wang, and A. Nee. Digital twins and cyber–physical systems toward smart manufacturing and industry 4.0: Correlation and comparison. *Engineering*, 5(4):653–661, 2019.
 - [348] S. J. Taylor and B. Letham. Forecasting at scale. *The American Statistician*, 72(1):37–45, 2018.
 - [349] A. Tchernykh, U. Schwiegelsohn, V. Alexandrov, and E.-g. Talbi. Towards understanding uncertainty in cloud computing resource provisioning. *Procedia Computer Science*, 51:1772–1781, 2015.
 - [350] S. Teerapittayanon, B. McDanel, and H.-T. Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 328–339. IEEE, 2017.
 - [351] T. N. Theis and H.-S. P. Wong. The end of moore’s law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.
 - [352] B. Tian, C. Tian, H. Dai, and B. Wang. Scheduling coflows of multi-stage jobs to minimize the total weighted job completion time. In *IEEE Conference on Computer Communications*, pages 864–872. IEEE, 2018.
 - [353] N. H. Tran, W. Bao, A. Zomaya, M. N. Nguyen, and C. S. Hong. Federated learning over wireless networks: Optimization model design and analysis. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1387–1395. IEEE, 2019.
-

- [354] S. Tuli. Splitplace: Intelligent placement of split neural nets in mobile edge environments. *arXiv preprint arXiv:2110.04841*, 2021.
- [355] S. Tuli. Splitplace: Intelligent placement of split neural nets in mobile edge environments. *SIGMETRICS Perform. Eval. Rev.*, 49(2):63–65, jan 2022. ISSN 0163-5999. doi: 10.1145/3512798.3512821. URL <https://doi.org/10.1145/3512798.3512821>.
- [356] S. Tuli. Splitplace: Intelligent placement of split neural nets in mobile edge environments. *ACM SIGMETRICS Performance Evaluation Review*, 49(2):63–65, 2022.
- [357] S. Tuli, R. Mahmud, S. Tuli, and R. Buyya. Fogbus: A blockchain-based lightweight framework for edge and fog computing. *Journal of Systems and Software*, 154:22–36, 2019.
- [358] S. Tuli, N. Basumatary, S. S. Gill, M. Kahani, R. C. Arya, G. S. Wander, and R. Buyya. HealthFog: An ensemble deep learning based Smart Healthcare System for Automatic Diagnosis of Heart Diseases in integrated IoT and fog computing environments. *Future Generation Computer Systems*, 104:187–200, 2020.
- [359] S. Tuli, S. S. Gill, G. Casale, and N. R. Jennings. iThermoFog: IoT-Fog based automatic thermal profile creation for cloud data centers using artificial intelligence techniques. *Internet Technology Letters*, 3(5): e198, 2020.
- [360] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya. Dynamic scheduling for stochastic edge-cloud computing environments using A3C learning and residual recurrent neural networks. *IEEE Transactions on Mobile Computing*, 2020.
- [361] S. Tuli, R. Bansal, R. Paul, et al. TANGO: Commonsense Generalization in Predicting Tool Interactions for Mobile Manipulators. *International Joint Conference on Artificial Intelligence (IJCAI)*, 2021.
- [362] S. Tuli, S. S. Gill, P. Garraghan, R. Buyya, G. Casale, and N. Jennings. START: Straggler Prediction and Mitigation for Cloud Computing Environments using Encoder LSTM Networks. *IEEE Transactions on Services Computing*, 2021.
- [363] S. Tuli, S. S. Gill, M. Xu, P. Garraghan, R. Bahsoon, S. Dustdar, R. Sakellariou, O. Rana, R. Buyya, G. Casale, et al. HUNTER: AI based Holistic Resource Management for Sustainable Cloud Computing. *Journal of Systems and Software*, pages 111–124, 2021.
- [364] S. Tuli, S. Poojara, S. N. Srirama, G. Casale, and N. Jennings. Cosco: Container orchestration using co-simulation and gradient based optimization for fog computing environments. *IEEE Transactions on Parallel and Distributed Systems*, 2021.
- [365] S. Tuli, S. Tuli, G. Casale, and N. R. Jennings. Generative optimization networks for memory efficient data generation. *Advances in Neural Information Processing Systems, Workshop on ML for Systems*, 2021.
- [366] S. Tuli, G. Casale, and N. R. Jennings. GOSH: Task Scheduling using Deep Surrogate Models in Fog Computing Environments. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

-
- [367] S. Tuli, G. Casale, and N. R. Jennings. MCDS: AI Augmented Workflow Scheduling in Mobile Edge Cloud Computing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2022.
 - [368] S. Tuli, G. Casale, and N. R. Jennings. PreGAN: Preemptive Migration Prediction Network for Proactive Fault-Tolerant Edge Computing. In *IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2022.
 - [369] S. Tuli, G. Casale, and N. R. Jennings. CAROL: Confidence-Aware Resilience Model for Edge Federations. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2022.
 - [370] S. Tuli, G. Casale, and N. R. Jennings. DRAGON: Decentralized Fault Tolerance in Edge Federations. In *IEEE Transactions on Network and Service Management (TNSM)*. IEEE, 2022.
 - [371] S. Tuli, G. Casale, and N. R. Jennings. TranAD: Deep Transformer Networks for Anomaly Detection in Multivariate Time Series Data. *Proceedings of VLDB*, 15(6):1201–1214, 2022.
 - [372] S. Tuli, F. Mirhakimi, S. Pallewatta, S. Zawad, G. Casale, B. Javadi, F. Yan, R. Buyya, and N. R. Jennings. AI Augmented Edge and Fog Computing: Trends and Challenges. *arXiv preprint arXiv:2208.00761*, 2022.
 - [373] G. Tychogiorgos, A. Gkelias, and K. K. Leung. A non-convex distributed optimization framework and its application to wireless ad-hoc networks. *IEEE Transactions on Wireless Communications*, 12(9):4286–4296, 2013.
 - [374] Y. A. Ushakov, P. N. Polezhaev, A. E. Shukhman, M. V. Ushakova, and M. Nadezhda. Split neural networks for mobile devices. In *2018 26th Telecommunications Forum (TELFOR)*, pages 420–425. IEEE, 2018.
 - [375] P. van de Ven, S. Borst, and S. Shneer. Instability of maxweight scheduling algorithms. In *IEEE INFOCOM 2009*, pages 1701–1709. IEEE, 2009.
 - [376] P. M. van de Ven, S. C. Borst, and L. Ying. Inefficiency of maxweight scheduling in spatial wireless networks. *Computer Communications*, 36(12):1350–1359, 2013.
 - [377] B. Varghese and R. Buyya. Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems*, 79:849–861, 2018.
 - [378] X. Vasilakos, W. Featherstone, N. Uniyal, A. Bravalheri, A. S. Muqaddas, N. Solhjoo, D. Warren, S. Moazzeni, R. Nejabati, and D. Simeonidou. Towards Zero Downtime Edge Application Mobility for Ultra-Low Latency 5G Streaming. In *2020 IEEE Cloud Summit*, pages 25–32. IEEE, 2020.
 - [379] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30:5998–6008, 2017.
 - [380] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *ICLR*, 2018.
 - [381] R. S. Venkatesh, T. Smejkal, D. S. Milojicic, and A. Gavrilovska. Fast in-memory criu for docker containers. In *The International Symposium on Memory Systems*, pages 53–65, 2019.
-

- [382] B. Wang, F. Liu, and W. Lin. Energy-efficient vm scheduling based on deep reinforcement learning. *Future Generation Computer Systems*, 125:616–628, 2021.
- [383] C. Wang, Y. Xu, J. Zhang, and B. Ran. Integrated traffic control for freeway recurrent bottleneck based on deep reinforcement learning. *IEEE Transactions on Intelligent Transportation Systems*, 2022.
- [384] H. Wang, X. Shi, and D.-Y. Yeung. Natural-parameter networks: a class of probabilistic neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pages 118–126, 2016.
- [385] H. Wang, G. Cai, Z. Huang, and F. Dong. ADDA: Adaptive Distributed DNN Inference Acceleration in Edge Computing Environment. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 438–445. IEEE, 2019.
- [386] H. Wang, S. Gui, H. Yang, J. Liu, and Z. Wang. GAN Slimming: All-in-One GAN Compression by A Unified Optimization Framework. In *European Conference on Computer Vision*, pages 54–73. Springer, 2020.
- [387] J. Wang, N. Wu, W. X. Zhao, F. Peng, and X. Lin. Empowering A* search algorithms with neural networks for personalized route recommendation. In *The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 539–547, 2019.
- [388] L. Wang, W. Mao, J. Zhao, and Y. Xu. DDQP: A double deep Q-learning approach to online fault-tolerant SFC placement. *IEEE Transactions on Network and Service Management*, 18(1):118–132, 2021.
- [389] N. Wang, M. Matthaiou, D. S. Nikolopoulos, and B. Varghese. DYVERSE: dynamic vertical scaling in multi-tenant edge environments. *Future Generation Computer Systems*, 2020.
- [390] P. Wang, Y. Lei, P. R. Agbedanu, and Z. Zhang. Makespan-driven workflow scheduling in clouds using immune-based PSO algorithm. *IEEE Access*, 8:29281–29290, 2020.
- [391] Q. Wang and H. Mu. Privacy-preserving and lightweight selective aggregation with fault-tolerance for edge computing-enhanced iot. *Sensors*, 21(16):5369, 2021.
- [392] T. Wang, Y. Liang, Y. Zhang, X. Zheng, M. Arif, J. Wang, and Q. Jin. An intelligent dynamic offloading from cloud to edge for smart IoT systems with big data. *IEEE Transactions on Network Science and Engineering*, 7(4):2598–2607, 2020.
- [393] Y. Wang, H. Liu, W. Zheng, Y. Xia, Y. Li, P. Chen, K. Guo, and H. Xie. Multi-objective workflow scheduling with deep-q-network-based multi-agent reinforcement learning. *IEEE access*, 7:39974–39982, 2019.
- [394] Y. Wang, J. Yang, X. Guo, and Z. Qu. Satellite edge computing for the internet of things in aerospace. *Sensors*, 19(20):4375, 2019.
- [395] Y. Wang, Y. Guo, Z. Guo, T. Baker, and W. Liu. Closure: A cloud scientific workflow scheduling algorithm based on attack–defense game model. *Future Generation Computer Systems*, 111:460–474, 2020.

-
- [396] Y. Wang, S. Wang, B. Yang, B. Gao, and S. Wang. An effective adaptive adjustment method for service composition exception handling in cloud manufacturing. *Journal of Intelligent Manufacturing*, pages 1–17, 2020.
- [397] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM Computing Surveys (CSUR)*, 53(3):1–34, 2020.
- [398] J. B. Weissman and A. S. Grimshaw. A federated model for scheduling in wide-area systems. In *Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*, pages 542–550. IEEE, 1996.
- [399] S. Werner and S. Tai. Application-platform co-design for serverless data processing. In *International Conference on Service-Oriented Computing*, pages 627–640. Springer, 2021.
- [400] M. Wiering and M. Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3), 2012.
- [401] S. Wijethilaka and M. Liyanage. Survey on network slicing for internet of things realization in 5g networks. *IEEE Communications Surveys & Tutorials*, 23(2):957–994, 2021.
- [402] B. Wilder, B. Dilkina, and M. Tambe. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *AAAI*, volume 33, pages 1658–1665, 2019.
- [403] H. Won and Y. Kim. Performance analysis of machine learning based fault detection for cloud infrastructure. In *2021 International Conference on Information Networking (ICOIN)*, pages 877–880. IEEE, 2021.
- [404] D. Wu, R. Ullah, P. Harvey, P. Kilpatrick, I. Spence, and B. Varghese. Fedadapt: Adaptive offloading for iot devices in federated learning. *IEEE Internet of Things Journal*, 9(21):20889–20901, 2022.
- [405] W. Wu, L. He, W. Lin, and R. Mao. Accelerating federated learning over reliability-agnostic clients in mobile edge computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 2020.
- [406] C. Xia, J. Zhao, H. Cui, X. Feng, and J. Xue. Dnntune: Automatic benchmarking dnn models for mobile-cloud computing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(4):1–26, 2019.
- [407] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.
- [408] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *Symposium on Operating Systems Design and Implementation*, pages 595–610, 2018.
- [409] Y. Xie, Y. Zhu, Y. Wang, Y. Cheng, R. Xu, A. S. Sani, D. Yuan, and Y. Yang. A novel directional and non-local-convergent particle swarm optimization based workflow scheduling in cloud–edge environment. *Future Generation Computer Systems*, 97:361–378, 2019.
- [410] X. Xiong, K. Zheng, L. Lei, and L. Hou. Resource allocation based on deep reinforcement learning in IoT edge computing. *IEEE Journal on Selected Areas in Communications*, 38(6):1133–1146, 2020.
-

- [411] T. Xu, G. Han, X. Qi, J. Du, C. Lin, and L. Shu. A hybrid machine learning model for demand prediction of edge-computing-based bike-sharing system using internet of things. *IEEE Internet of Things Journal*, 7(8):7345–7356, 2020.
- [412] Y.-H. Xu, C.-C. Yang, M. Hua, and W. Zhou. Deep deterministic policy gradient (ddpg)-based resource allocation scheme for noma vehicular communications. *IEEE Access*, 8:18797–18807, 2020.
- [413] Z. Xu, J. Tang, C. Yin, Y. Wang, G. Xue, J. Wang, and M. C. Gursoy. ReCARL: Resource Allocation in Cloud RANs with Deep Reinforcement Learning. *IEEE Transactions on Mobile Computing*, 2020.
- [414] L. Yang, C. Zhong, Q. Yang, W. Zou, and A. Fathalla. Task offloading for directed acyclic graph applications based on edge computing in industrial internet. *Information Sciences*, 540:51–68, 2020.
- [415] Z. Yang, M. Chen, W. Saad, C. S. Hong, and M. Shikh-Bahaei. Energy efficient federated learning over wireless communication networks. *IEEE Transactions on Wireless Communications*, 20(3):1935–1949, 2020.
- [416] Z. Yao, A. Gholami, Q. Lei, K. Keutzer, and M. W. Mahoney. Hessian-based analysis of large batch training and robustness to adversaries. *Advances in Neural Information Processing Systems*, 31, 2018.
- [417] Z. Yao, A. Gholami, K. Keutzer, and M. W. Mahoney. Pyhessian: Neural networks through the lens of the hessian. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 581–590. IEEE, 2020.
- [418] Z. Yao, A. Gholami, S. Shen, M. Mustafa, K. Keutzer, and M. Mahoney. Adahessian: An adaptive second order optimizer for machine learning. In *AAAI conference on Artificial Intelligence*, volume 35, pages 10665–10673, 2021.
- [419] I. Yaqoob, K. Salah, R. Jayaraman, and Y. Al-Hammadi. Blockchain for healthcare data management: Opportunities, challenges, and future recommendations. *Neural Computing and Applications*, pages 1–16, 2021.
- [420] P. Yazdanian and S. Sharifian. E2LG: a multiscale ensemble of LSTM/GAN deep learning architecture for multistep-ahead cloud workload prediction. *The Journal of Supercomputing*, pages 1–31, 2021.
- [421] K. Ye, Y. Liu, G. Xu, and C.-Z. Xu. Fault injection and detection for artificial intelligence applications in container-based clouds. In *International Conference on Cloud Computing*, pages 112–127. Springer, 2018.
- [422] F. Yu, L. Cui, P. Wang, C. Han, R. Huang, and X. Huang. Easiedge: A novel global deep neural networks pruning method for efficient edge computing. *IEEE Internet of Things Journal*, 8(3):1259–1271, 2020.
- [423] M. Yu, Z. Jiang, H. C. Ng, W. Wang, R. Chen, and B. Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *International Conference on Distributed Computing Systems*, 2021.
- [424] S. Yun, M. Jeong, R. Kim, J. Kang, and H. J. Kim. Graph transformer networks. *Advances in Neural Information Processing Systems*, 32:11983–11993, 2019.
- [425] E. Zeydan, E. Bastug, M. Bennis, M. A. Kader, I. A. Karatepe, A. S. Er, and M. Debbah. Big data caching for networking: Moving from cloud to edge. *IEEE Communications Magazine*, 54(9):36–42, 2016.

-
- [426] C. Zhang, D. Song, Y. Chen, X. Feng, C. Lumezanu, W. Cheng, J. Ni, B. Zong, H. Chen, and N. V. Chawla. A deep neural network for unsupervised anomaly detection and diagnosis in multivariate time series data. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1409–1416, 2019.
- [427] L. Zhang and L. Zhang. Deep learning-based classification and reconstruction of residential scenes from large-scale point clouds. *IEEE Transactions on Geoscience and Remote Sensing*, 56(4):1887–1897, 2017.
- [428] L. Zhang, Z. Li, and C. Wu. Dynamic resource provisioning in cloud computing: A randomized auction approach. In *IEEE Conference on Computer Communications*, pages 433–441. IEEE, 2014.
- [429] S. Zhang, H. Tong, J. Xu, and R. Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 6(1):1–23, 2019.
- [430] S. Zhang, S. Zhang, Z. Qian, J. Wu, Y. Jin, and S. Lu. Deepslicing: collaborative and adaptive cnn inference with low latency. *IEEE Transactions on Parallel and Distributed Systems*, 32(9):2175–2187, 2021.
- [431] Y. Zhang, P. Cai, C. Pan, and S. Zhang. Multi-agent deep reinforcement learning-based cooperative spectrum sensing with upper confidence bound exploration. *IEEE Access*, 7:118898–118906, 2019.
- [432] Y. Zhang, H. Huang, L.-X. Yang, Y. Xiang, and M. Li. Serious challenges and potential solutions for the industrial Internet of Things with edge intelligence. *IEEE Network*, 33(5):41–45, 2019.
- [433] Y. Zhang, Y. Chen, J. Wang, and Z. Pan. Unsupervised deep anomaly detection for multi-sensor time-series signals. *IEEE Transactions on Knowledge and Data Engineering*, 2021.
- [434] H. Zhao, J. Wang, F. Liu, Q. Wang, W. Zhang, and Q. Zheng. Power-aware and performance-guaranteed virtual machine placement in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, 29(6):1385–1400, 2018.
- [435] H. Zhao, S. Deng, Z. Liu, J. Yin, and S. Dustdar. Distributed redundancy scheduling for microservice-based applications at the edge. *IEEE Transactions on Services Computing*, 2020.
- [436] H. Zhao, Y. Wang, J. Duan, C. Huang, D. Cao, Y. Tong, B. Xu, J. Bai, J. Tong, and Q. Zhang. Multivariate time-series anomaly detection via graph attention network. *International Conference on Data Mining*, 2020.
- [437] Z. Zhao, K. Wang, N. Ling, and G. Xing. EdgeML: An AutoML Framework for Real-Time Deep Learning on the Edge. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, pages 133–144, 2021.
- [438] Z. Zheng, T. C. Zhou, M. R. Lyu, and I. King. Component ranking for fault-tolerant cloud applications. *IEEE Transactions on Services Computing*, 5(4):540–550, 2011.
- [439] H. Zhou, W. Zhang, C. Wang, X. Ma, and H. Yu. Bbnet: A novel convolutional neural network structure in edge-cloud collaborative inference. *Sensors*, 21(13):4494, 2021.
-

- [440] M. Zhou, R. Zhang, W. Xie, W. Qian, and A. Zhou. Security and privacy in cloud computing: A survey. In *2010 Sixth International Conference on Semantics, Knowledge and Grids*, pages 105–112. IEEE, 2010.
- [441] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proceedings of the IEEE*, 107(8):1738–1762, 2019.
- [442] H. Zhu, H. Dai, S. Yang, Y. Yan, and B. Lin. Estimating power consumption of servers using gaussian mixture model. In *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, pages 427–433. IEEE, 2017.
- [443] H. Zhu, M. Akrouf, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 88–100. IEEE, 2018.
- [444] Z. Zhu, J. Peng, Z. Zhou, X. Zhang, and Z. Huang. PSO-SVR-based resource demand prediction in cloud computing. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, 20(2):324–331, 2016.
- [445] A. Zinnen and T. Engel. Deadline constrained scheduling in hybrid clouds with gaussian processes. In *2011 International Conference on High Performance Computing & Simulation*, pages 294–300. IEEE, 2011.
- [446] B. Zong, Q. Song, M. R. Min, W. Cheng, C. Lumezanu, D. Cho, and H. Chen. Deep autoencoding gaussian mixture model for unsupervised anomaly detection. In *International Conference on Learning Representations*, 2018.