Imperial College of Science, Technology and Medicine
Department of Electrical and Electronic Engineering

Enabling On-device Domain Adaptation of Convolutional Neural Networks

Aditya Rajagopal

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering of
the Diploma of Imperial College, September 2022

Abstract

Convolutional Neural Networks (CNN) are used ubiquitously in computer vision applications ranging from image classification to video-stream object detection. However due to the large memory and compute costs of executing CNNs, specialised hardware such as GPUs or ASICs are required to perform both CNN inference and training within reasonable time and memory budgets. Consequently, most applications today perform both CNN inference and training on servers where user data is sent from an edge device back to a server to process. This raises data privacy concerns and places a strict necessity for good edge-server communication links. Recently, with improvements in the specialised hardware (especially GPUs) available on edge devices, an increased number of applications have moved the inference stage onto the edge, but few to none have considered performing training on an edge device. With a focus on CNNs used for image classification, the work in this PhD explores when it would be useful to perform retraining of networks on an edge device, what the gains would be of doing so and how one can perform such training even in resource constrained settings.

This exploration begins with the assumption that the classes observed by the model upon deployment is a subset of the classes present in the dataset used to train the model initially. This scenario is simulated by constructing semantically meaningful subsets of classes from existing large image classification datasets (eg. ImageNet) and exploring the gains, in terms of classification accuracy and the memory consumption and latency of the inference and training stages, that can be achieved by pruning (architecture modification) and retraining (weights adaptation) a deployed network to the observed class distribution.

The exploration is split into three stages. First, an oracle is constructed that predicts the gains that can be achieved by pruning and retraining a network under the assumption that we know the exact label of each image observed upon deployment and do not have any hardware resource constraints. This demonstrates the accuracy and performance gains that can theoretically be achieved per network and subset combination. The significant gains demonstrated here for certain subsets of data motivate the remainder of the work in this PhD. The works that follow explore ways to perform such adaptation on hardware that is resource constrained and also when there is uncertainty in the labels of the observed data-points that are used to perform this adaptation.

Pruning was utilised as a method to enable training to be performed on resource constraint hardware by reducing the memory and latency footprints of the training process. When doing so, it was observed that depending on the manner in which a network is pruned, a set of networks that all consume the same amount of memory for storing weights, can each have drastically different latencies and memory consumptions while performing training. Hence, the size of a stored model is not a useful predictor of which networks can be feasibly trained within edge hardware resource budgets. To cater for this, a novel, accurate and data-driven model for predicting the training memory consumption and latency of a network on a specific target hardware and execution framework (PyTorch, Tensorflow, etc.) combination is proposed. Doing so enables the selection of a pruned network, whose memory consumption and latency of training fits within the available memory and latency budgets that are dictated by the target hardware and application. This then allows for the network to be adapted to the observed data distribution. An additional benefit of using the proposed data-driven model is that it allows to rapidly create new models specific to each network, hardware and execution framework combination.

Finally, the analysis is extended to account for uncertainty in the class labels of the observed data distribution. This uncertainty in the label distribution can negatively impact any attempts to retrain the network. To combat this, a novel Variational Auto-Encoder (VAE) based retraining methodology that uses uncertain predictions of the label of an image to adapt the weights of the network to the observed data distribution on-device is proposed.

In doing so, the work in this PhD answers the questions of why we should aim to train a network on the edge, how we can select networks that fit within the available hardware resource constraints and how we could account for the uncertainty in labels that arises when we do not have access to ground-truth labels when training. We also propose possibilities for future research directions that could extend and adapt the ideas of this thesis to other applications.

Acknowledgements

First I would like to thank my supervisor, Professor Christos-Savvas Bouganis for the wealth of support and ideas provided over the years of my PhD as well as constantly making me think of new ways to effectively describe and express my research contributions in papers and presentations. Not only has your guidance helped me grow as a researcher, but it has also been vital in furthering my communication skills.

I would also like to thank my friend, flatmate and fellow PhD student Diederik Vink for being a sounding board for ideas and a constant source of advice throughout my PhD. The experience would not have been the same without four years of our lab-disrupting conversations.

With COVID-19 making us work from home for a good part of two years of the PhD, I would also like to thank my friends and flatmates Vinay Maniam and Jesus Garcia for the endless amounts of fun that helped maintain sanity through the times of lockdown. I could not have asked for better flatmates.

I would like to sincerely thank my mum, Gauri Rajagopal and uncle, Ramesh Ramamurthy for providing their unending love and support from 7000 km away, despite the tough times at home these past years, that pushed me to always strive to do better in all my endeavours.

Although you aren't here to read this, I would also like to thank my dad, Rajagopal Ramakrishnan, who has constantly been a guiding voice in my head and has helped me progress through any challenges I've faced during the PhD and otherwise.

Last but not least, I would like to thank my partner Chhavi Maggu for being a pillar of love and support throughout the ups and down that every PhD student faces and making the experience of completing a PhD while in a global pandemic a thoroughly enjoyable one.

Statement of Originality

I certify that the content of this thesis is a produce of my own work and that all work and sources that have aided in preparing this thesis have been cited and acknowledged.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Deep learning has seen rapid growth over the last decade with applications ranging from computer vision, natural language processing (NLP), chip design [56] and drug discovery [37]. Depending on the use case, the latency, power consumption and memory budgets of these applications can vary drastically; and so do the devices used for their execution. For instance in the case of NLP, without considering the overheads of executing the network on a device, the parameters of the state-of-the-art GPT-3 [7] model alone take up approximately 350GB. In such cases, the norm is to execute the model on a server with a large number of devices (often GPUs) with access to large amounts of memory. For instance, the recently open-sourced language model BLOOM (based on GPT-3) took 3.5 months to train on a server with 384 GPUs each with 80GB of memory [83]. Consequently, running such models directly on an edge device such as a mobile phone is infeasible with today's edge hardware capabilities.

In other applications, such as chip-design or drug discovery, the nature of the use case does not quite necessitate the model to be available in real-time or on embedded devices. Consequently, many such applications are also executed on servers to capitalise on the massive scale of hardware resources available.

However, convolutional neural networks (CNNs), the state-of-the-art in deep learning based computer vision, have smaller memory resource requirements and are often deployed on resource

constrained edge-devices and in scenarios, such as autonomous driving [69, 3], which require real-time outputs. Within the domain of computer vision, CNNs are used for a wide range of applications ranging from image classification [44], segmentation [51], drone navigation [41] to object detection [42, 21]. Their state-of-the-art (SOTA) performance in these tasks can be attributed to the vast amounts of curated data available through datasets such as ILSVRC'12 [17] that are used to train them. This training, which involves optimising the network's weights and topology, is performed offline on large data-centres with access to devices such as server grade GPUs and TPUs [36]. Once deployed, the weights and topology of a network typically do not change over time.

However, large quantities of domain specific data that can help further improve the performance of these networks in terms of accuracy and/or inference time, reside on the edge on each user's personal device. This has been observed in the domain of federated learning both from the perspective of statistical distribution shift of the data observed on each device [72] and security concerns of preserving a user's privacy [4] by not sharing data across devices or back to a centralised server for processing. The domain of federated learning tackles the problem of learning a single global model based on data that is obtained from multiple nodes (user devices), without sharing that data between nodes or with a centralised server. Typically this requires some initial processing (training) of a model to be performed on the device. Outputs from this processing, that do not contain user sensitive information, are then shared between devices or a server in order to aggregate these results and update a global model. [72] demonstrate that using such an aggregation technique as compared to learning a single model per device can obtain up to 60% improvement in prediction accuracy depending on the prediction task at hand.

However, as discussed in [27], there is limited use of federated learning for updating deployed CNN models as the hardware constraints of edge devices prevent the training of CNNs on such devices. The currently adopted approach to updating such resource intensive models is to send information regarding the observed data distribution back to a server for processing [29]. However, this raises both data-privacy concerns and can also be infeasible in scenarios where

there is limited or no edge-server connectivity. Consequently, there has been a push to increase the intelligence of the processing performed on edge devices [91, 70]. This coupled with the increased compute and memory capabilities of modern edge devices such as the NVIDIA Jetson TX2, has motivated the questions targeted in this PhD.

Although used as a motivating example to demonstrate the benefits of utilising the vast amount of user specific data present at the edge, federated learning is not the focus of this PhD. The work in this PhD does not aim to tackle the problem of aggregating data obtained from multiple nodes in a network. Rather, it focuses on adapting a deployed image classification CNN model on-device, by training a model per node using only the data observed by that node without any edge-server communication. The first challenge in doing so arises due to the highly resource intensive nature of training CNNs coupled with the constrained resources of edge devices. Additionally, a second challenge is the uncertainty that arises due to the lack of ground-truth labels available for images observed on the edge, when using these images to retrain a deployed network.

The rest of this chapter is organised as follows. Sec.1.1 motivates the reasons for exploring on-device adaptation of CNN models to the observed data. Sec.1.2 introduces the problem that is tackled by the works in the PhD. Sec.1.3 provides further context into the various assumptions made by the problem setting as well as the envisioned deployment scenario for the proposed solutions. Sec.1.4 summarises the novel contributions of the work done during the PhD.

1.1 Motivation

Consider the scenario of an image classification CNN network deployed on a robotic vacuum cleaner [32, 1]. The typical deployment pipeline would involve training the network offline on a large dataset consisting of classes that cover a wide variety of household objects. Following this, the deployed network typically remains static regardless of the type of household in which it is deployed. However, depending on the household the vacuum is deployed in, the style of furniture, the type of flooring and the number of objects can vary drastically. In such cases, it

would be beneficial to be able to adapt the network on-device to better classify images that it is frequently exposed to as opposed to successfully classifying a larger range of classes as seen in the training set.

Another scenario in which such adaptation could be useful is in autonomous vehicles that use CNNs for image classification. A similar argument can be made where a vehicle deployed in India as opposed to one in the UK would be required to detect different objects. For instance, the UK has almost no 3 wheeled vehicles on the road, whereas rickshaws are wide spread in India. Similarly, classification of images in differing weather conditions (snow vs dessert for instance) across countries could benefit from CNNs that are specialised to classify that particular subset of images.

Due to this uncertainty of the classes that are expected to be observed upon deployment, the datasets that are used to train networks offline such as ILSVRC'12 [17] are large and extensive. Consequently, it is likely that the classes of images observed upon deployment (target domain) will be a subset of those that were used during training (source domain). To simulate this scenario, 4 semantically meaningful subsets of the ILSVRC'12 dataset were created, namely City, Motorway, Countryside, and Off-Road. Each subset only consisted of images of objects found in each of these settings. For instance, images of tractors would not be found in the City subset. For a MobileNetV2 [67] network trained initially on the entire ILSVRC'12 dataset, the average increase across subsets in classification accuracy that was obtained by retraining the network to the target domain classes was 7 percentage points (*pp*) with a maximum increase of 15*pp*. This difference is relative to not retraining the network and classifying images based on the original network trained on the entire ILSVRC'12 dataset. Consequently, in the scenario where the observed data upon deployment consists only of a subset of classes of the original training dataset, there are significant benefits to classification accuracy of retraining a network to the observed data distribution.

However, this raises the question of where such training should be performed. In the scenario of the robotic vacuum, network connectivity is expected to be available as the operation occurs in a household setting where internet connection can be reliable. However, there are significant

privacy considerations with regards to sending images of a user's home to a centralised server for model adaptation. On the other hand, in the autonomous vehicle scenario, network connectivity cannot be reliably expected and hence sending data back to a centralised server for model adaptation may not be feasible at all. Consequently, there is benefit to exploring ways to improve the prediction capabilities of these networks in a manner that is completely on-device.

Training a CNN on an edge device is not trivial due to the resource constraints of such devices. For instance amongst edge devices, the NVIDIA Jetson TX2 edge GPU resides on the higher end of the available resources spectrum with 8GB of memory and a processor capable of performing 1.33 Tera-Floating Point Operations per second (TFLOPs). Training ResNet50 [28], a CNN that is widely accepted as a strong performer on the ILSVRC'12 dataset, can take more than 11GB of memory during training depending on the training configuration (i.e. image dimensions and batch size) used. Furthermore MobileNetV2, a network developed with the goal of being able to deploy CNNs on edge devices consumes $\sim 10GB$ of memory during training on ILSVRC'12 images with a standard training batch size of 128. Thus, the first challenge of on-device training resides in being able to perform training within the memory constraints of an edge device.

The works in this PhD propose novel solutions to adapting a network directly on an edge device to the observed data distribution, under the assumption that the classes observed upon deployment are a subset of those used to train the original network. In doing so, the works propose novel solutions to the challenging problems of how to reduce the training memory and latency of a CNN, how to predict these metrics accurately so as to choose favourable deployments of these networks and how to perform this edge re-training without ground-truth labels for the observed images upon deployment.

1.2 Problem Setting

This section provides a formal definition for the on-device domain adaptation problem that is explored in the PhD. Additionally, the section also introduces metrics of interest that are

Metric	Device Agnostic?	Description
$\bar{\alpha}(\mathcal{M})$	Yes	The mean Top1 test-set accuracy across all **target** domain classes for model $\mathcal{M}$.
$\alpha_0(\mathcal{M})$	Yes	The mean Top1 test-set accuracy across all **source** domain classes for model $\mathcal{M}$.
$\rho(\mathcal{M})$	Yes	Number of floating point operations performed during model inference for model $\mathcal{M}$.
$\gamma(\mathcal{M})$	Yes	Static memory consumption of model $\mathcal{M}$.
$\delta(\mathcal{M})$	No	Runtime memory consumption of model $\mathcal{M}$.
$\phi(\mathcal{M})$	No	Latency of the inference stage of model $\mathcal{M}$.
$\Delta(\mathcal{P})$	Yes	Static memory consumption of the adaptation process $\mathcal{P}$.
$\Theta(\mathcal{P})$	No	Latency of the adaptation process.
$\Lambda(\mathcal{P})$	No	Runtime memory consumption of the adaptation process.

Table 1.1: The metrics of interest which are used to evaluate various deployment systems and domain adapted models.

important to consider when evaluating methodologies that tackle this problem.

Let us consider a large training dataset $\mathcal{D} = \{\mathcal{I}, \mathcal{C}\}$, where $\mathcal{I}$ is the set of images in the dataset, and $\mathcal{C}$ is the set of classes represented by the images. Let us define a model $\mathcal{M}$ as a tuple $(\mathcal{W}, \mathcal{A})$, where $\mathcal{W}$ represents the weights of the model and $\mathcal{A}$ represents the topology of the model. After performing training on $\mathcal{D}$, the resulting model $\mathcal{M}_{\mathcal{D}} = (\mathcal{W}_{\mathcal{D}}, \mathcal{A}_{\mathcal{D}})$ is such that for class $i \in C$, a_i is the accuracy that the model predicts that class with.

Consider the case where the model $\mathcal{M}_{\mathcal{D}}$ is deployed in an environment $\mathcal{D}' = \{\mathcal{I}', \mathcal{C}'\}$ where the classes that are expected to be seen upon deployment are not known before deployment. To account for this uncertainty, researchers utilise large and extensive datasets such as ILSVRC'12 to train CNNs offline before deployment. Consequently, a reasonable assumption to make is that the classes observed upon deployment are a subset of the classes in the original training dataset, i.e. $\mathcal{C}' \subseteq \mathcal{C}$.

Given these definitions, the metrics on which proposed solutions will be evaluated on are introduced and then the problem itself is formalised as an optimisation process that aims to find a model with certain constraints on these metrics.

1.2.1 Metrics of Interest

This section introduces metrics that should be measured in order to evaluate an adaptation process $\mathcal{P}$ and resulting domain adapted model $\mathcal{M}$. The metrics introduced are detailed in

Table 1.1.

Outside the scope of training a CNN on an edge device, when considering just deploying one for inference, there are a number of dimensions that are important to jointly optimise for. First, the test-set classification accuracy of the deployed model on the observed data distribution is the metric that is primarily optimised for ($\bar{\alpha}(\mathcal{M})$). Note the distinction between $\bar{\alpha}(\mathcal{M})$ and $\alpha_0(\mathcal{M})$ where the former corresponds to the classification accuracy on the observed data distribution, while the latter corresponds to that on the original training distribution. However in addition to this, it is important to consider the memory consumption of the weights of the model as the model is now being stored on memory constrained edge devices. This is referred to as the static memory consumption of the model ($\gamma(\mathcal{M})$) in Table 1.1. Following this the memory ($\delta(\mathcal{M})$) and latency ($\phi(\mathcal{M})$) of the inference (classification) stage of the model should also be minimised. This memory consumption differs from γ as this accounts for more than just the memory consumption of the weights, but also that of intermediate results that need to be stored on-device during the classification of an image. As there are multiple ways to execute a network on a given device, δ and ϕ vary depending on the device on which execution occurs and the framework (PyTorch, Tensorflow etc.) used to execute the network. This information is captured in Table 1.1 under the *DeviceAgnostic*? column. Additionally, to provide a device agnostic way of comparing device specific metrics such as δ and ϕ, the number of floating point operations that a network performs during inference (ρ) is also a commonly reported metric in the literature.

When considering training (adapting) a network on-device, the following metrics should also be considered when evaluating the chosen adaptation strategy. First, the static memory consumption of the adaptation process ($\Delta(\mathcal{P})$). This includes aspects such as supplementary models that may need to be used to aid the adaptation of a network (and their storage costs) and storage of the images themselves that are used for retraining. Similar to the inference stage, the runtime memory consumption ($\Lambda(\mathcal{P})$) and latency ($\Theta(\mathcal{P})$) of the adaptation process also need to be considered when deciding on a feasible process that can execute within the hardware constraints of edge devices.

In this thesis, the framework used is PyTorch and the devices on which results are evaluated are the NVIDIA Jetson Tx2 and the RTx 2080Ti for embedded and server settings respectively. Although on the higher end of the available resources spectrum for edge devices, the Jetson Tx2 provides a unified CPU and GPU memory system with 8GB of available memory. This places constraints on the sizes of training workloads that can be executed on it, however still has enough memory to and compute power to be able to prototype solutions that train CNNs on these devices. The RTx 2080Ti is a fully server grade device with 11GB of GPU-only memory and was used in the context of evaluating proposed training memory and latency prediction models on server grade devices.

1.2.2 Problem Definition

Given an observed data distribution upon deployment $\mathcal{D}'$ and the following compute and memory budgets of the adaptation process, $\phi_{max}, \gamma_{max}, \Lambda_{max}, \Theta_{max}$ (dictated either by the hardware resources available and/or the classification task at hand), the works that follow aim to find a domain adapted model ($\mathcal{M}_{\mathcal{D}'}$) such that:

$$\left.\begin{aligned} &\phi(\mathcal{M}_{\mathcal{D}'}) < \phi(\mathcal{M}_{\mathcal{D}}) \\ &\gamma(\mathcal{M}_{\mathcal{D}'}) < \gamma(\mathcal{M}_{\mathcal{D}}) \\ &\gamma(\mathcal{M}_{\mathcal{D}'}) < \gamma_{max} \\ &\phi(\mathcal{M}_{\mathcal{D}'}) < \phi_{max} \end{aligned}\right\} \text{Inference stage costs of the adapted model are less than the corresponding costs of the original deployed model and the available compute and memory budgets.}$$

$$\left.\begin{aligned} &\Theta(\mathcal{P}) < \Theta_{max} \\ &\Lambda(\mathcal{P}) < \Lambda_{max} \end{aligned}\right\} \text{Adaptation process costs are less than the available compute and memory budgets to perform the domain adaptation on-device.}$$

$$\left.\begin{aligned} &\bar{\alpha}(\mathcal{M}_{\mathcal{D}'}) \geq \bar{\alpha}(\mathcal{M}_{\mathcal{D}}) \end{aligned}\right\} \text{The mean classification accuracy of the resulting adapted network on the classes observed in the target domain is at least as good as that of the original deployed model.}$$

Furthermore, the methodologies to obtain $\mathcal{M}_{\mathcal{D}'}$ from $\mathcal{M}_{\mathcal{D}}$ adapt the weights (through retraining) and/or topology of the model. In the scope of this thesis, alternative techniques to weight adaptation such as quantisation are not explored.

1.3 Context and vision

The previous sections introduced the motivation behind the work carried out in this PhD and formally defined the on-device domain adaptation problem. Building on these, this section provides further context to and outlines the envisioned problem setting in a deployment scenario.

Let us first explore further the introduced scenario of an autonomous car. Before exploring solutions to the proposed problem setting of retraining the deployed network to changes in the observed classes, there are simpler solutions that should first be considered. For instance, in the case of different countries it might be more prudent to train separate models for each location and deploy that particular model based on the car's location. In the case of different countries, this may be a viable solution. However using multiple stored models still limits the granularity in classes to which the deployed model can be adapted. Having the ability to retrain a model on-device instead of having separate models enables the adaptation process to account for the scenario where the observed data distribution consists of classes that are a mixture of those that each individual model was optimised for. In the case of the robotic vacuum cleaner, having a model per type of household would not be feasible as unlike features on the road, it is much more likely to see a mixture of household items from different regions in a single house. Hence, having the ability to optimise the model's classification ability for each combination of observed classes is useful. Thus, the work in this PhD aims to propose solutions for the most general scenario where the deployed model can adapt to any subset of the classes that were present in $\mathcal{D}$ when training $\mathcal{M}_{\mathcal{D}}$ offline.

Additionally, both in the case of the autonomous vehicle and the robotic vacuum cleaner, there is an argument to be made for sharing data between vacuum cleaners or cars that are in the same location. Doing so would prevent redundant retraining and result in significant energy

savings. As discussed earlier, solutions to this are explored in the field of federated learning by works such as [72] which aim to train a single model from data aggregated from multiple nodes. However, this is not the focus of the work proposed in this PhD as the problem setting aims to also cater for the scenario where there is no communication capabilities on the edge due to poor network connectivity. Such a scenario would preclude any data sharing. Furthermore, as discussed in [27], one of the challenges in implementing federated learning for the case of CNNs is the computational cost of training a CNN on an edge device. As the solutions proposed in this PhD also aim to enable training of CNNs on edge devices, we hope that they could be applied in the future to the federated learning scenario in order to capitalise on the ability to share data between models.

Within the proposed problem setting however, there are some notable constraints. First, since we do not know the observed class distribution a priori (i.e. at training time), the proposed solutions will have to establish a way to estimate the observed class distribution upon deployment before initiating the retraining process. Consequently, the problem setting constrains the observed class distribution to be a subset of that of $\mathcal{D}$ under the assumption that the training dataset $\mathcal{D}$ is extensive enough to capture a large majority of the classes that are expected to be observed upon deployment. The problem setting does not include scenarios where previously unseen classes are observed. Such scenarios are tackled by works such as [64] which explore methods in the research area of online incremental learning.

Another constraint of the proposed problem setting is that it only explores the scenario where the observed data distribution and the training data distribution vary only in the classes observed but not the distribution of the data itself. For instance, we do not expect the training data to consist of images from a camera for the autonomous car, but upon deployment the data observed is actually that from a much lower resolution webcam or a completely different type of camera. This is an important distinction as it positions the problem being tackled against similar but not identical domains of research such as Partial Domain Adaptation (PDA) where both the class and data distribution vary between the training and observed data.

Having discussed the context of the problem setting and the various constraints involved, the

remainder of this section will detail our vision in terms of applying such adaptation in a real deployment scenario. The solutions proposed in this PhD aim to adapt a network to the observed class distribution by first adapting the topology of the network in a manner that reduces the computational cost of executing the network and then retraining this network to the observed data in order to achieve gains in classification accuracy. The adaptation process should also be low-cost enough to be performed directly on an edge device. Definitions of what entails an edge device vary and as discussed earlier, for the works in this PhD the NVIDIA Jetson TX2 is utilised as the edge device on which the solutions are prototyped and evaluated. The Jetson TX2 is on the higher end of the available resource spectrum when it comes to edge devices where Microcontrollers lie on the opposite end of this spectrum. However, the memory consumption of retraining modern CNNs is larger than even the available memory budget of a Jetson TX2. Consequently, we aim to first provide solutions that can perform such adaptation on such a large edge device but envision that the proposed methodologies could reduce the computational costs of adaptation to an extent that in the future such methodologies could be applied to a device as resource constrained as a Microcontroller.

Finally, the envisioned deployment scenario would be to have a system that can identify the observed class distribution without any supervision, retrain a smaller version of the initially deployed model to this observed distribution and deploy the specialised model while the device is still in the newly observed environment. However, whether such instantaneous update of the model is possible depends both on deployment scenario as well as the time taken for the proposed adaptation process to complete. For instance, certain scenarios my not necessitate such rapidly changing models. In the cases where we do want almost instantaneous adaptation, we envision that the adapted model ($\mathcal{M}_{\mathcal{D}'}$) is retrained simultaneously to inference of the originally deployed model ($\mathcal{M}_{\mathcal{D}}$). In such a scenario the training would also need to be continual, i.e. once sufficient data to perform one iteration of retraining is collected, the model is trained and the data is discarded. However, in other scenarios where we can either afford to adapt the model periodically or are required to do so due to the time taken by the adaptation process, we envision that the deployed system would need to account for the extra memory required to store a dataset of images observed on the edge in order to trigger the retraining when the

system is idle. For instance, this could be when an autonomous car is parked overnight or at the office and can optimise the classification capabilities of a network for the specific route from home to office for which data has been collected.

1.4 Contributions

As described in Sec.1.1, the main challenges with performing on-device adaptation of CNNs are the hardware resource limitations of edge devices and the uncertainty in the labels of images observed upon deployment. Consequently, the novel contributions of the PhD that tackle these challenges are as follows.

A feasibility study of on-device retraining of CNNs The PhD explores gains in classification accuracy ($\bar{\alpha}$) that can be achieved by performing pruning (topology adaptation) and retraining (weights adaptation) of a CNN on an edge device when the observed data distribution upon deployment consists of a subset of classes of the original training data. Furthermore, the trade-off between the accuracy gains that can be achieved and the inference and training costs of doing so is also studied. This study focuses on the no-label-uncertainty case in order to establish a baseline to compare to.

Autoprune An open-source framework that automates the process of pruning a network for a wide range of network architectures. By automating both the identification and pruning of convolution filters that are dependent on each other due to network topology constraints, the tool enables dynamic pruning and retraining solutions to be deployed directly on edge devices without user intervention.

Pruning free adaptation strategy The costs of performing pruning directly on-device are high due to the significant amount of training data required when pruning a model and the iterative pruning and retraining nature of the process. Consequently, the PhD proposes

a methodology that can rapidly search a large space of small sub-networks from a super-network and identify a sub-network to deploy, that fits within the desired inference and training budgets. In doing so, this work provides a novel and feasible methodology for on-device network adaptation by significantly reducing the costs of the adaptation process to account for the hardware constraints of the target edge device.

Perf4sight A novel and highly accurate data-driven methodology to model the memory consumption and latency of both the training and inference stage of a CNN on an edge device. Although developed to enable the rapid selection of a sub-network from a super network that fits within the target hardware and classification task constraints, the proposed modelling methodology also allows to rapidly create memory and latency prediction models for new CNNs, target devices and frameworks without requiring in-depth knowledge of the execution process of each device and/or framework.

LoCO-PDA A novel methodology that utilises Conditional Variational Autoencoders (CVAEs) [73] to retrain a CNN to the observed data distribution that not only accounts for the uncertainty in labels of the observed data distribution, but also performs the entire adaptation process on a CPU within acceptable time constraints.

1.5 Publications

The following publications contributed directly to this thesis. In the list below, (1) is the publication linked directly to the feasibility study and *Autoprune* tool contributions. (2) is the publication linked directly to the pruning free adaptation and *perf4sight* tool contributions. (3) is linked directly to the LoCO-PDA contribution.

1. A. Rajagopal and C.-S. Bouganis. Now that i can see, i can improve: Enabling data-driven finetuning of cnns on the edge. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020

2. A. Rajagopal and C.-S. Bouganis. perf4sight: A toolflow to model cnn training performance on edge gpus. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, pages 963–971, October 2021

3. A. Rajagopal and C.-S. Bouganis. Low-cost on-device partial domain adaptation (locopda): Enabling efficient cnn retraining on edge devices. arXiv, 2022

The following publications did not contribute directly towards the thesis.

1. A. Rajagopal, D. Vink, S. Venieris, and C.-S. Bouganis. Multi-precision policy enforced training (MuPPET) : A precision-switching strategy for quantised fixed-point training of CNNs. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 7943–7952. PMLR, 13–18 Jul 2020

2. D. A. Vink, A. Rajagopal, S. I. Venieris, and C.-S. Bouganis. Caffe barista: Brewing caffe with fpgas in the training loop. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 317–322, 2020

1.6 Thesis Structure

The rest of the thesis is organised as follows. Chapter 2 describes various state-of-the-art techniques that are compared against and evaluates their performance on the metrics proposed in Table.1.1. Additionally, the chapter introduces various concepts such as CNNs, the network training process and Variational Autoencoders (VAEs) that are important to understand the contributions of the PhD.

Chapter 3 describes the feasibility study for on-device adaptation that was performed as well as the Autoprune tool.

Chapter 4 describes the methodology to rapidly select a sub-network from a super-network that enables model adaptation without pruning to be performed on-device. This chapter also describes the proposed memory and latency modelling methodology Perf4sight.

Chapter 5 describes the proposed methodology that handles the uncertainty in the class labels of data observed on the edge. In doing so, this creates a system that can adapt a deployed CNN to the data observed without having access to ground-truth labels, all within the restricted hardware constraints of the Jetson TX2 edge device.

Finally, Chapter 6 summarises the main contributions of the PhD and describes possible future directions of research that the work done here has enabled.

Chapter 2

Background

As described in Chapter 1, the focus of the PhD is on exploring methods of adapting the weights (through retraining) and topology (through pruning), of a CNN on resource constrained edge devices. The target application is that of image classification, where the underlying assumption is that the classes of images observed upon deployment are a subset of the classes present in the original training dataset. Consequently, the aim is to train the network on the edge device to improve the classification capabilities of the deployed CNN on the specific observed target domain.

This chapter provides the background necessary to better understand the proposed methodologies and puts into perspective the performance of state-of-the-art techniques and baselines relevant to the works introduced in later chapters. The rest of this chapter is organised as follows.

Sec.2.1 is an introduction to Convolutional Neural Networks (CNNs), the state-of-the-art technique in computer vision for the task of image classification. It describes the topology of a CNN as well as the steps involved in the inference and training processes. While on the topic of types of neural networks, Sec.2.2 introduces generative models called Variational Autoencoders (VAEs) which were utilised in one of the contributions of this PhD. Sec.2.3 then discusses relevant works that were explored as viable methods for performing topology adaptation on-device.

Sec.2.4 introduces the state-of-the-art in the domain of inference and training stage memory and latency modelling, another area of research explored in this PhD. Finally, Sec.2.5 describes the related research areas of Partial Domain Adaptation (PDA) and on-device training and positions the work done in this PhD against them.

2.1 Convolutional Neural Networks (CNNs)

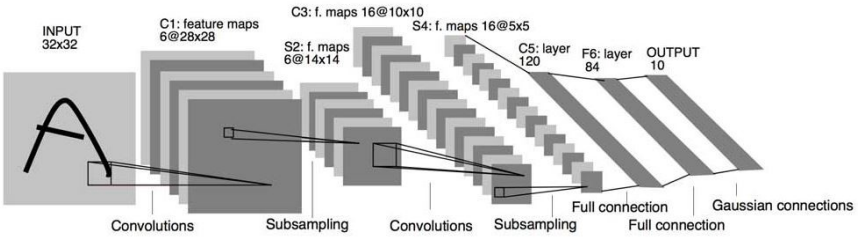


Figure 2.1: LeNet [46] architecture

CNNs were developed as specialised deep neural networks for processing image inputs. They are comprised of multiple non-linear convolutional layers which progressively map input images into features maps (or activations) that decrease in spatial dimensions while maintaining distinguishing information between images. Compared to densely connected neural networks, where all input neurons are connected to all output neurons, CNNs use a receptive field architecture where this dense connectivity is only present in the receptive field (kernel) which slides along the input. This significantly reduces the computational requirements of the process thus enabling it to be performed on large dimensional inputs such as images. This structure is visualised in Fig.2.1.

The earlier convolutional layers act as generic feature extractors that identify features of images, such as edges, that are common across all types of images in the training dataset. The later convolutional layers extract features that are unique to images of a specific class. These feature maps are then provided to fully connected (FC) layers (all input neurons connected to all output neurons) which act as classifiers (or regressors).

The typical non-linearity used in CNNs is the ReLU [85] layer which outputs 0 for inputs less

than 0 and has the graph $y = x$ for $x > 0$. Modern day CNNs also have additional layers such as batch-normalisation [34] and pooling [20] which are not discussed here as they are not relevant to the discussion of the proposed work in the PhD.

In the context of this thesis, the inference stage is defined as the classification of an input image and the training stage is defined as the process of updating the weights of a network in a manner that improves its final classification accuracy. Both training and inference can be performed in either single image (one image at a time) or batched (multiple images are processed simultaneously, generating multiple classifications) modes. However, it is typical for inference to be performed one image at a time and training to be performed in a batched manner. In this setting, the notion of batch size refers to the number of images that are being processed simultaneously.

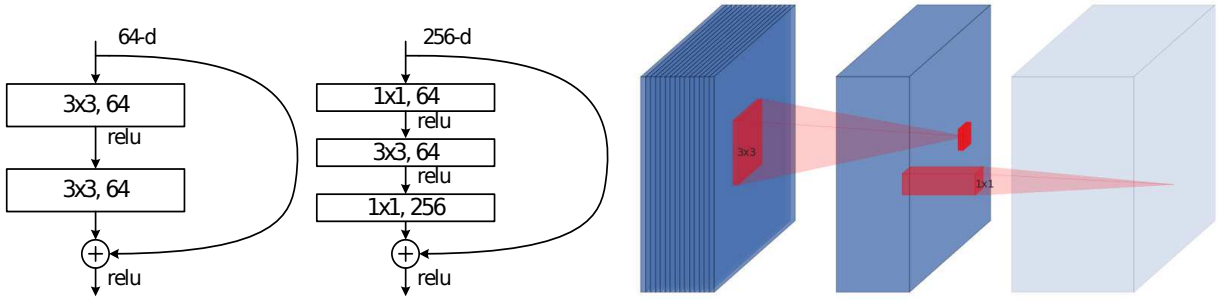
The rest of this section formally defines various terms related to the topology of a CNN, describes the topology of various architectures that are evaluated in this thesis, details the CNN inference and training processes and finally discusses the algorithms that are used to perform training on devices such as GPUs. The terminology introduced here is then maintained throughout the rest of this thesis.

2.1.1 CNN Topology Terminology

For a CNN with $\mathcal{L}$ convolutional layers, layer $l \in \mathcal{L}$ is said to have n_l *convolutional filters* (Weights) of size $(m_l \times k_l \times k_l)$ each, where n_l is the number of *output feature maps* (OFM), m_l the number of *input feature maps* (IFM) and k_l the spatial dimension of the convolutional filter. Each of the n_l convolutional filters are independently convolved with an IFM of dimension $bs \times m_l \times ip_l \times ip_l$ where bs is the batch size and $ip_l \times ip_l$ are the input spatial dimensions. This produces an OFM of dimension $bs \times n_l \times op_l \times op_l$. $\{m_l, n_l\}$ correspond to depth dimensions, bs the number of images in a batch, and $\{k_l, ip_l, op_l\}$ to spatial dimensions. The output spatial dimensions, op_l, can be inferred from ip_l and the convolution operation parameters of stride (s_l) and padding (p_l) as $op_l = 1 + \lfloor \frac{ip_l + 2p_l - k_l}{s_l} \rfloor$. The stride dictates the amount by which the

$k_l \times k_l$ sliding window moves in each step and the padding dictates the number of zeros that are placed on each border of the IFM in order to ensure that the dimensions of the OFM is a whole number.

2.1.2 Relevant CNN Architectures



(a) Residual Blocks [28] **(b)** Depthwise Separable Unit [67]

Figure 2.2: Structural blocks that require consideration of dependencies when pruning

Fig.2.1 describes the topology of one of the earliest proposed CNNs, LeNet [46], that was used to classify handwritten digits. Since then, numerous modern architectures have been proposed that work on much larger inputs. LeNet expected 32×32 inputs, modern networks that classify ILSVRC'12 images expect inputs of dimensions 224×224. This section briefly describes the CNN networks evaluated in this thesis.

AlexNet [44] This network was one of the first networks to successfully classify ILSVRC'12 images and has a similar structure to LeNet but with an increased number of filters per layer (wider) and a larger number of layers (deeper).

ResNets [28] These networks are made up of a repeated set of *Residual* building blocks that were proposed in order to aid the training of deeper (increased number of layers) networks. The structure of the residual blocks are described in Fig.2.2a. The key feature of a residual basic block is the bypass connection that sums in the input to the residual with the output of the convolution layers. This was proposed to facilitate the network to zero the weights of certain

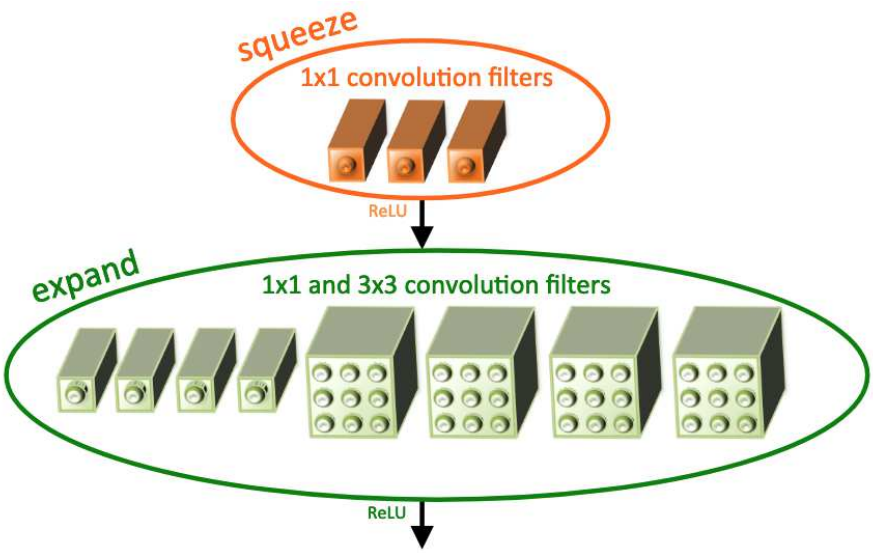


Figure 2.3: Fire Module in SqueezeNet [33]

convolution layers, creating an identity residual block that forwards the input to the output, during training. Hence very deep networks, such as ResNet-101 which has 101 convolutional layers, could be developed and the training process would zero out convolution layers that were not useful for classification.

MobileNetV2 [67] These networks were developed in order to reduce the computational cost of performing convolutions by proposing the *MBConv* building block which performs depth-wise convolutions. As described in Sec.2.1.1, a typical convolution applies an $m_l \times k_l \times k_l$ kernel to an $m_l \times ip_l \times ip_l$ IFM. This means that each of the n_l kernels are applied to all m_l channels of the IFM. The depth-wise convolution instead applies a $1 \times k_l \times k_l$ to each of the m_l channels of the IFM, thus reducing the computational cost by a factor of m_l. Note that this places a restriction on n_l in that $n_l = m_l$ for such convolutions as we cannot have more convolutional filters (n_l) than IFM channels. This is then followed by a 1×1 convolution which computes linear combinations of the input channels, thereby producing a similar effect to a typical convolution which acts on all input channels for each filter. Fig.2.2b describes this building block. These building blocks are also used in constructing MnasNet [78], another network that is evaluated in this thesis.

SqueezeNet [33] Also with the goal of reducing the number of parameters and model size, the *Fire Module* building block (Fig.2.3) was proposed. The authors noted that by maximising the number of 1×1 convolutions, they could reduce the number of parameters by $9\times$ compared to 3×3 convolutions. Additionally, they also aimed to limit the number of input channels (m_l) to 3×3 convolutions to reduce the computational cost and hence suggested the squeeze-expand architecture of the fire module seen in Fig.2.3. The architecture allows to control the number of 1×1 convolutions in the squeeze stage, which limits the number input channels to 3×3 convolutions. Additionally, the number of 3×3 and 1×1 convolutions in the expand stage can also be controlled. This functions as a strategy to limit the number of 3×3 convolutions in general. The OFMs generated by the various convolutions in the expand stage are then concatenated before being passed to the next layer. This computation pattern of expand followed by concatenation is also found in GoogLeNet [75], which is another network that is evaluated in this thesis.

To provide some context on the computational costs of and classification accuracy achieved by these networks, Table.2.1 summarises the device agnostic metrics introduced in Table.1.1 of number of floating point operations performed during inference (ρ), the static memory consumption of the model (γ) and the achieved Top-1 test set accuracy on the entire ILSVRC'12 dataset (α_0). Table.2.1 demonstrates that ResNet50 has the highest classification accuracy but also consumes the second highest amount of memory for storing the model and the largest number of inference stage FLOPs. On the other hand, both MobileNetV2 and MnasNet (architectures constructed from the *MBConv* basic block), perform about 4pp in classification accuracy lower than ResNet50, but on average consume 6.6x less storage memory and 13.2x less inference stage FLOPs. This indicates that for the edge deployment scenario, *MBConv* based networks have the strongest performance when accounting for the accuracy to compute cost trade-off.

The following section describes the various algorithms that are used to execute such networks on hardware devices. Analysing these algorithms then allows one to reason about the metrics of interest described in Table.1.1 that are device or framework specific.

Network	α_0 (%)	γ (MB)	ρ (GFLOPs)
AlexNet [44]	63.3	244.0	0.66
ResNet18 [28]	69.8	46.8	1.81
ResNet50 [28]	76.2	102.0	4.09
MobileNetV2 [67]	71.9	14.0	0.31
MnasNet [78]	73.5	17.5	0.31
SqueezeNet [33]	58.1	5.0	0.82
GoogLeNet [75]	69.8	26.5	1.50

Table 2.1: Device agnostic metrics of interest for models evaluated in this thesis. α_0 refers to the top-1 classification accuracy on the entire ILSVRC'12 validation datasets. γ refers to the static memory consumption of the model. ρ refers to the number of floating point operations performed by the network during inference.

2.1.3 Algorithms for executing CNNs on hardware

This section will introduce the CNN inference and training processes in terms of the operations that are performed and describe three algorithms that are commonly used to execute these networks on devices such as GPUs.

The CNN training process is based on the back-propagation [65] algorithm that involves a forward pass to calculate a loss followed by a backward pass that calculates the gradient w.r.t. the weights and inputs for each layer. A gradient descent step then updates the weights with the calculated gradients.

For the purpose of classification, the loss function that is typically used is the cross-entropy loss. For a CNN that is classifying between $\mathcal{C}$ classes, the output from the forward pass is a $\mathcal{C}$ dimensional vector $[p_1 \dots p_\mathcal{C}]$ where p_i is the probability that a particular image belongs to class i. The training process also receives as input the ground-truth label of each image. This information is one-hot encoded into a vector $[y_1 \dots y_\mathcal{C}]$ where $y_i = 1$ if the image belongs to class i and $y_i = 0$ otherwise. The cross-entropy loss (L) is then calculated as $-\sum_{i=1}^{\mathcal{C}} y_i log(p_i)$. The rest of this section describes the operations performed during the forward and backward passes.

For each convolution layer $l \in \mathcal{L}$ the following operations are performed. For each input feature

map (IFM) x, output feature map (OFM) y and weight matrix w, the forward pass performs the convolution [1]

$$y = x * w \tag{2.1}$$

where x has dimensions $bs \times m_l \times ip_l \times ip_l$, w has dimensions $n_l \times m_l \times k_l \times k_l$ and y has dimensions $bs \times n_l \times op_l \times op_l$. After computation of the loss L, the back-propagation algorithm requires the following computations per layer. The gradient w.r.t. inputs is calculated as [2]

$$\frac{\delta L}{\delta x} = \frac{\delta L}{\delta y} * rot180(w) \tag{2.2}$$

and the gradient w.r.t. weights is calculated as

$$\frac{\delta L}{\delta w} = x * \frac{\delta L}{\delta y} \tag{2.3}$$

Note that in order to compute the gradient w.r.t weights, the IFM to a layer x, which was computed during the forward pass (OFM of previous layer is IFM of current layer), is required. In most implementations of backpropagation on hardware, the activation is stored until the backward pass computation is completed. This contributes significantly to the memory consumption of the training process [10]. On NVIDIA GPUs, these operations are commonly implemented using the cuDNN [35] library. There are three ways in which a convolution operation is commonly executed on a GPU.

Matrix Multiplication This algorithm transforms the IFM using the *im2col* operation to perform a convolution as a matrix multiplication [14]. For a single IFM x of dimensions $m_l \times ip_l \times ip_l$ and a single filter w of dimensions $m_l \times k_l \times k_l$, the *im2col* operation first transforms x into x_{i2c} of dimensions $op_l^2 \times (m_l \cdot k_l^2)$. The operation is taking each step of the sliding window (there are total of op_l^2 steps), and flattening the part of the IFM that corresponds to that step into a $(m_l \cdot k_l^2)$ vector. Each of these vectors are then multiplied by a flattened w vector of dimensions $(m_l \cdot k_l^2)$. As there is overlap between sliding windows when the stride of

[1] $*$ refers to the convolution operator

[2] $rot180$ refers to a 180° rotation of the matrix

a convolution is less than k_l, this algorithm results in significant memory storage redundancy when creating x_{i2c}.

Fast Fourier Transform (FFT) This algorithm perform convolutions as a product between weights and IFMs in the frequency domain [55]. It benefits from reduced operations compared to matrix multiplication and the ability to precompute and reuse the FFT of the weights during the training process.

Winograd Convolution This algorithm reduces the number of operations performed by up to 4x compared to a matrix multiplication and uses up to 163x less temporary memory compared to the FFT operation [45].

All three algorithms discussed above display variability in performance with respect to the characteristics of the layer. For instance 1×1 convolutions do not have any overlap between sliding windows and hence can be expressed as matrix multiplications without any memory storage redundancies. Thus, the choice of algorithm here is always the matrix multiplication algorithm. However in the case of 3×3 convolutions, the Winograd algorithm is about 50% faster than the other two options in about 75% of the cases [35]. Cases here differ in other parameters such as batch size and stride which change from layer to layer. Hence, this necessitates the cuDNN library to use proprietary heuristics to decide which of these three algorithms to apply on a per layer basis.

This section introduced CNNs, the class of deep neural networks that is the primary focus of the work in this PhD. It described the topology of a CNN from an abstract point of view as well as the details of commonly found CNN architectures. It introduced a notation that will be used throughout this thesis to refer to aspects of a CNN's topology and finally described the various operations that occur during execution of both the forward and backward passes on hardware. The following section provides a much briefer introduction to a different class of deep neural networks called Variational Autoencoders (VAEs) that are utilised in contributions of this PhD.

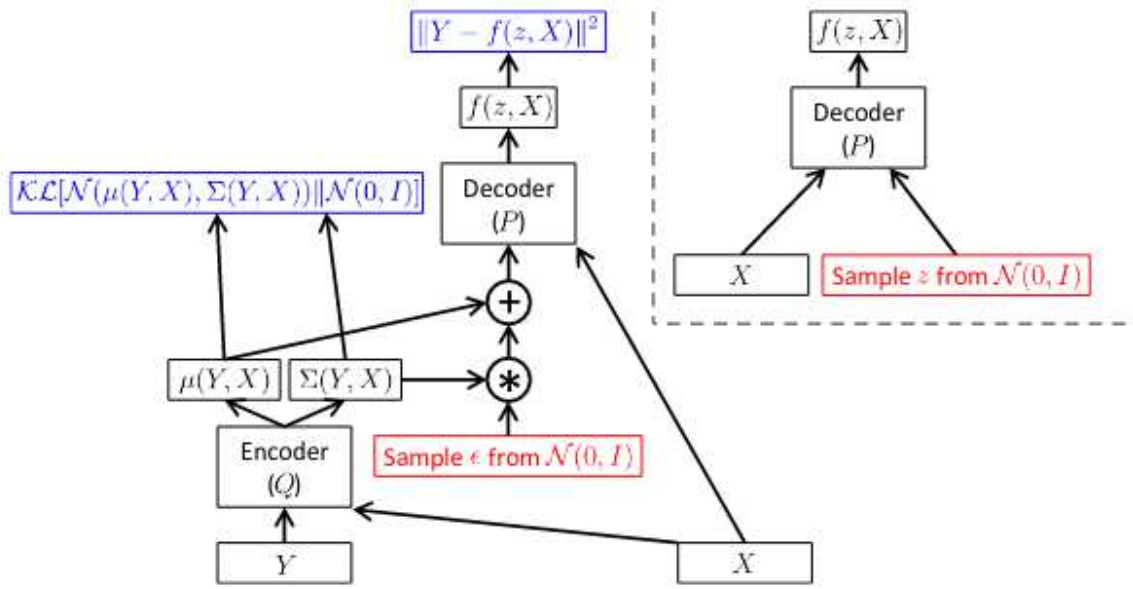


Figure 2.4: Conditional VAE Training (Left) and Inference (Top Right) Processes [18]

2.2 Variational Autoencoders (VAE)

This section provides background on VAEs which are a class of generative machine learning models. VAEs [40] are generative models that learn to model a distribution $P(Y)$ where Y are high-dimensional data points which can represent a variety of inputs, but typically images. They do this by learning a latent representation characterised by latent variables (z) that can be sampled to generate data points close to $P(Y)$.

Consider a set of data points Y and corresponding class labels X. The non-standard notation for inputs and labels used here is to keep in-line with that used by [18] in Fig.2.4. The figure shows the training and inference processes of an extension to VAEs known as Conditional VAEs (CVAEs). CVAEs have the benefit of allowing the generation of data points (Y) conditioned on a particular class (X). The encoder learns the distribution $P(z|Y,X) \sim \mathcal{N}(\mu,\Sigma)$ where a separate function is learnt for the mean μ and variance Σ. The decoder learns the distribution $P(Y|z,X)$.

The loss terms when training the encoder and decoder are those shown in blue boxes in Fig.2.4. The KL-divergence term $\mathcal{KL}[\mathcal{N}(\mu(Y,X),\Sigma(Y,X))||\mathcal{N}(0,I)]$ works both as a regularizer and forces the distribution of the latent space variables to be as close as possible to $\mathcal{N}(0,I)$. The

second term $||Y - f(z, X)||^2$ is the reconstruction error and pushes the decoder to learn a function that can map random variables $z \sim \mathcal{N}(0, I)$ to a representative Y given a class X. This way during the inference process (top-right in Fig.2.4), a class X and a variable randomly sampled from $\mathcal{N}(0, I)$ can be passed to the decoder to generate data points corresponding to the desired class X.

2.3 On-device CNN weights and topology adaptation

The goal of this PhD is to explore ways to adapt both a CNN's weights (through retraining) as well as topology. Sec.2.1 introduced the topology of CNNs and described the manner in which the weights are adapted (training process). This section describes the state-of-the-art in various approaches to adapt a CNN's topology in order to reduce its memory and latency footprints during both inference and training to achieve the constraints specified by the problem definition in Sec.1.2.2. In doing so, this section will produce a landscape of the various approaches available to perform on-device topology adaptation and the trade-offs between costs of inference, training and accuracy that exists between these approaches. Towards this, three different approaches; pruning (Sec.2.3.1), Once-For-All networks [9] (Sec.2.3.2) and PersEPhonEE [47] (Sec.2.3.3) are considered for the target network of ResNet50 [28].

In order to evaluate both the costs of inference and training in a device agnostic manner, this section uses the FLOPs (number of floating point operations performed) of both the inference and training stages as a comparison metric. For ResNet50 and the training configuration used by the works discussed below, i.e. training batch size of 128 and training dataset of ILSVRC'12 (224×224 input image size), 1 mini-batch of training (1 iteration) requires 1.87 TFLOPs of computation to be performed. An epoch is defined as the number of iterations required to complete one pass through of all images in the training dataset. Table.2.2 presents a summary of the results discussed below.

2.3.1 Pruning Algorithms

The field of pruning can be broadly split into two categories; structured and unstructured pruning. Unstructured pruning [86, 26] removes individual neurons in a CNN which result in sparse structures that require specialised hardware [88] to accelerate. Although able to achieve extremely high sparsity ratios of 90% with minimal accuracy loss [15], they do not lend to reducing training or inference costs on off-the-shelf devices and frameworks.

Similarly, quantisation of network parameters and feature maps, another technique used to reduce the memory and compute costs of the inference stage requires access to specialised hardware accelerators such as [13]. Furthermore, most existing works in quantisation explore performing inference, not training, with quantised weights and feature maps. [63], one of the early works to explore quantised training, only emulates the quantisation during training as none of the existing execution frameworks such as PyTorch support quantised training natively.

Consequently, for the scope of this PhD, structured pruning techniques are considered in order to be able to leverage the significant available compute of existing edge devices such as the NVIDIA Jetson Tx2.

The typical method of reducing the computational cost of training and inference through structured pruning is to first perform pruning [48] where entire convolution filters are removed. This creates smaller rather than sparser structures which can be accelerated using off-the-shelf libraries and hardware. Here, pruning methodologies vary based on the choice of metric used to choose which convolutional filters to remove. Following this, the pruned network is retrained in order to regain the lost accuracy. Typically, this pruning and retraining is done in an iterative manner where small portions of the network are pruned in each iteration until the desired pruning level is achieved. Majority of the existing literature focuses on performing pruning in order to reduce the inference (not training) stage memory and latency footprints of a CNN. Additionally, none of the existing methodologies explore performing the pruning directly on-device. Hence, the pruning is performed offline on server grade devices and the computational costs of neither the process used to select filters to prune nor the retraining stage are considered when

Topology Adaptation Strategy	**Accuracy Loss** (pp)	γ **Reduction** (%)	ρ **Reduction** (-x)	**Topology Adaptation Cost** (TFLOPs)	**Finetuning Required**
L1-Norm	4.31	51.5	2.41	2.34×10^{-5}	Yes
ThiNet	4	66	3.5	3744	Yes
Taylor First Order	4	70	3.0	57.97	Yes
Variational	0	unknown	unknown	unknown	No
OFA	3	40	3.8	1.77×10^{-2}	No
PersEPhonEE	0	-40	3.1	3.32	Yes

Table 2.2: Comparison of state-of-the-art topology adaptation strategies for ResNet50 on the ILSVRC'12 dataset. **Accuracy Loss** refers to the loss in accuracy from utilising a smaller network (adapted topology). It is measured in percentage points (pp) compared to the classification accuracy reported in Table.2.1 for ResNet50. γ **Reduction** refers to the percentage of the network's weights that were removed during adaptation. ρ **Reduction** refers to the reported reduction in inference stage FLOPs (in -x times) compared to the original network (ρ reported in Table.2.1). **Pruning Cost** refers to the number of computations required to perform a single round of topology adaptation. It does not take into account the finetuning performed after all the adaptation is completed, but it does account for any iterative retraining performed for each round of pruning for instance. **Finetuning Required** refers to whether training of the shrunk network needs to be performed after the topology adaptation process is completed.

developing these techniques.

To position these works in the context of this thesis the remainder of this section will discuss various state-of-the-art pruning algorithms while also discussing the computational costs of the pruning process itself in order to evaluate the feasibility to perform these directly on edge devices. For the rest of this section, $X\%$ pruning will refer to $X\%$ of the network's parameters being removed corresponding to a reduction in static memory consumption (γ).

L1-norm pruning One of the earliest approaches to pruning, this methodology uses the L1-norm of each convolutional filter as the metric to choose which filters to prune. This methodology has a low computational cost and has shown some empirical [48] and theoretical [80] backing in its ability to select redundant filters. [48] showed that by ranking filters globally (across all layers) based on their L1-norm and then pruning and retraining the network, they were able to achieve 10% pruning without accuracy loss for ResNet34 on the ILSVRC'12 dataset using 20 epochs of retraining after pruning. [2] demonstrate that for ResNet50, L1-norm pruning can achieve 51.5% pruning and 2.41x reduction in inference FLOPs with 4.31pp loss in classification accuracy on ILSVRC'12. The cost of evaluating the L1-norm metric for all convolutional filters in ResNet50 is 23.4 MFLOPs. This is detailed in Table 2.2 under the L1-Norm row. As the

process is not iterative, the *Topology Adaptation Cost* in the table is just that of evaluating the L1-norm of all filters. As after the filters are removed, the networks is retrained (finetuned) for some number of epochs, the *Finetuning Required* column is labelled as Yes.

ThiNet [54] The authors propose a methodology that prunes the network in a layer-by-layer manner while greedily selecting channels to prune from layer i that minimise the reconstruction error of the inputs to layer $i+1$. The reconstruction error is measured as the difference in value of the sum of all input pixels to layer $i+1$ across all images in the training set with and without a particular filter in layer i. Then the filter which minimises this difference is greedily selected until a certain sparsity ratio per layer (uniform across network) is achieved. After each layer is pruned, 1-2 epochs of retraining is performed followed by another 10-12 epochs of retraining after the desired overall percentage sparsity has been achieved. ThiNet achieves up to 66% pruning and a 3.5× reduction in inference FLOPs with only a 4pp accuracy loss for ResNet50 on the ILSVRC'12 dataset. The computational cost of calculating the reconstruction error for all convolutional filters in ResNet50 is 4.25 GFLOPs for a single image. This value is then scaled to the number of images used for pruning (in their case 1,281,167 images, the number of images in the entire ILSVRC'12 training dataset). This followed by 1-2 epochs of retraining for each layer of the network would result in 3744 TFLOPs of computation per round of pruning. This is a result of 1 epoch of training requiring $\frac{1,281,167}{128}$ iterations where 128 is the batch size.

Taylor First Order pruning [58] The authors propose to use both gradient and weight information to measure channel importance and achieve up to 70% pruning and a 3.0× reduction in inference FLOPs with only a 4pp accuracy drop for ResNet50 on ILSVRC'12. The approach uses a first-order Taylor series expansion to estimate the effect of removing a single filter on the overall network classification loss. The pruning strategy removes 2% of filters every 30 iterations until the desired pruning level is achieved and then performs 25 epochs of finetuning at the end. Furthermore, the proposed metric achieves high correlation scores (0.925 with the Pearson correlation coefficient) between the filters selected using this metric and the filters that would be selected if the effect on the loss of removing each filter was measured by performing inference

without that filter (oracle). Computing this metric requires performing a single forward and backward pass and each pruning round requires 30 iterations of training which results in 57.97 TFLOPs of computations.

Variational Pruning [89] Notably one of the only works to target removing the requirement to perform finetuning after pruning, [89] propose a methodology to remove the requirement for finetuning by learning a distribution across the dataset of each channel's importance while using a sparse Gaussian prior to force the distribution to have a mean close to 0 for as many channels as possible. The pruning is done in an iterative manner for 120 epochs, but does not require any more finetuning after this. For ResNet50 on the ImageNet dataset, the approach manages to prune 40% of channels without any accuracy loss, however the memory or FLOPs reduction by doing so is not reported. Furthermore, as there was no open-source code available for this work, none of the results or computational costs of performing the pruning could be evaluated.

For a similar loss in classification accuracy, the L1-norm metric achieves 14.5pp less pruning and 1.5x less reduction in FLOPs. However, it is important to note the significantly reduced computational cost of performing this pruning as compared to other methodologies; 1.59×10^8x compared to ThiNet and 2.47×10^6x compared to Taylor First Order (TFO) pruning. Between ThiNet and TFO pruning, both methodologies achieve similar accuracy trade-off in classification accuracy vs model size reduction. In terms of computational cost per iteration of pruning, the TFO approach outperforms ThiNet by 65x. On an NVIDIA Jetson Tx2 which quotes a peak performance of 1.33 TFLOP/s (floating point operations per second), each iteration of pruning would take at least 47min. and 44s for ThiNet and TFO respectively. Consequently, moving forward only the L1-norm and TFO pruning strategies are considered.

However, pruning on-device raises an additional challenge of having to iteratively perform retraining of a network and a large amount of data to be stored in order to perform this pruning. Outside of the time cost of performing this retraining, as discussed in Sec.1.1, the memory consumption of training is a significant limiting factor to performing training on memory constrained edge devices. Additionally, as discussed in [53], pruning on limited data with a small

number of classes (observed upon deployment) results in worse performance compared to fine-tuning (retraining) a network that was pruned and retrained on the original training dataset. Consequently, work in this thesis will initially explore the costs of performing pruning directly on an edge device, however will progress to alternatives which focus on reducing retraining costs online while performing the pruning itself offline with access to the entire training dataset and server grade devices.

2.3.2 Pruning Alternative : Once-for-all (OFA) networks

[9] propose an algorithm which trains a single once-for-all (OFA) super network and uses weight sharing to progressively train smaller sub networks within it by changing the depth (number of layers) and width (number of filters per layer) of the network. The sub-networks are each trained using the weights of the trained super network as starting points. Following this training process, an evolutionary algorithm based search is performed, by using a trained accuracy predictor to evaluate the accuracy of each sub-network sampled from the super-network, in order to select networks which fit within the target inference stage latency constraints for the given device. This search samples $\sim 50,000$ sub-networks which due to weight sharing between these sub-networks, can all be fit into a single super-network with a memory consumption of only 192MB. The evolutionary algorithm then results in a sub-network with smaller memory and compute footprint than the original network (eg. unpruned ResNet50) with little loss in classification accuracy.

[9] achieve 40% pruning and a 3.8× reduction in inference FLOPs for only a 3pp reduction in accuracy for a network that is a modified version of ResNet50 on the ILSVRC'12 dataset. The cost of sampling a single sub-network is 0.345 MFLOPs, hence the total cost of pruning for this methodology is 0.0177 TFLOPs. As seen in Table.2.2, the OFA approach to pruning reduces the network's inference FLOPs to a similar extent to the other pruning methods introduced, while incurring a similar accuracy loss but leaving more the network's weights unpruned (less γ reduction). However, it reduces the cost per iteration of pruning by 3275x compared to TFO pruning, while also not requiring any finetuning after the pruning has been performed.

These properties make it a great candidate for on-device topology adaptation as it removes the requirement for costly finetuning after pruning as well as access to a memory intensive training dataset. These properties alleviate the important issues that were raised with the methodologies described in Sec.2.3.1.

2.3.3 Pruning Alternative : PersEPhonEE early-exit network

The previous section described an alternative methodology to pruning that could be used for topology adaptation to be performed directly on an edge device. This section provides the details of another such methodology that could be used for on-device topology adaptation based on early-exit networks.

Early-exit networks place multiple classifiers throughout the network with the aim of classifying "easier" inputs, i.e. inputs which the original network classifies with high certainty of classification, from earlier exits and thus on average across a dataset, reducing the inference FLOPs. [47] propose PersEPhonEE, a methodology that uses early-exit networks not just to reduce inference cost but also to reduce the training cost to adapt a network to the data-distribution observed. By placing an early exit architecture, consisting of two convolutional layers and a classifier, after various residual blocks along the ResNet50 architecture, the authors propose to improve retraining time by only retraining the desired early-exit block to the data observed while also resulting in inference latency gains. On ResNet50, the approach reduces training costs compared to training the entire network by 2× - 22× depending on the number of channels present in the early-exit layers. Furthermore by training the early exits, the accuracy of predictions made by earlier exits for the specific data-distribution observed can be significantly increased and in some cases by more than 20pp for ResNet50 on a subset of ILSVRC'12 classes. Consequently by being able to classify with more confidence on earlier exits, [47] achieved up to 3.1× reduction in inference FLOPs without sacrificing accuracy.

With this approach, the topology adaptation occurs through the process of retraining an early exit. [47] propose a 6-exit (exits placed at inference FLOPs normalised locations along the

network) architecture. As the choice of exit used is data-dependent, the mean cost of retraining across all exits is provided in Table.2.2 for fair comparison. Furthermore, the training is assumed to be performed with batch size 128 and across all data points in the ILSVRC'12 training set (for fair comparison with other approaches). The mean cost of training the 6 exits is 3.32 TFLOPs. Note Table.2.2 provides an entry of -40% for γ *Reduction* in order to represent the fact that the approach actually increases the memory consumption of the deployed network as it needs to store the entire early-exit augmented network.

Compared to TFO and OFA methodologies, PersEPhonEE performs worse for γ *Reduction*, but can achieve no loss in classification accuracy for a comparable reduction in ρ, while also reducing the topology adaptation cost by 17.5x compared to the TFO methodology. Consequently, this methodology is also evaluated against when evaluating proposed solutions.

2.4 Performance Modelling

The previous section introduced various methodologies that could be applied to adapt a network's topology on-device. Regardless of the method chosen to adapt the topology however, another goal of this PhD is to be able to retrain the adapted network to the data observed upon deployment. In order to do so, it is important to be able to predict both the memory and latency requirements of training for the adapted network so as to choose appropriate networks to retrain that fit within the device and application's memory and latency constraints. As discussed in Sec.2.4, there is significant variation in the choice of algorithm used for training a network on a GPU depending on the layer characteristics. This in-turn affects the memory and latency footprints of training on a per device and execution framework basis.

This section describes various methodologies that model the performance of either the inference or training stage of CNN execution. Such models can then aid in the choice of adapted network to retrain. The analysis of performance modelling works will be split into both inference and training stage performance modelling as well as embedded and cloud GPU performance modelling. Each scenario presents a unique application to model. The training stage has 3

different types of convolutions that are executed (Sec.2.1.3) compared to just 1 in the inference stage. Edge devices, for instance, tend to have a unified CPU-GPU memory, where the CPU and GPU share a common global memory. Cloud systems on the other hand tend to have multiple GPUs each with their own independent memory.

2.4.1 Inference Stage Performance Modelling

Currently, the focus of performance modelling for GPUs has been on the inference stage rather than the training stage. State of the art approaches for inference performance modelling, Augur [52] and Neural Power [8], explore the topic of predicting CNN inference memory consumption, runtime and energy on GPUs. Augur focuses its analysis on edge GPUs while Neural Power focuses on server grade GPUs. [52] approximate the forward pass as matrix multiplications and profile random matrix multiplication sizes on the NVIDIA TX1 device to train a model to make layer-wise predictions of memory, latency and energy. The developed model uses linear regression on a layer-wise basis based on the sizes of the matrix multiplications, and sums the predictions of each layer to obtain a final value. The developed models achieve on average across two embedded GPUs (NVIDIA TK1 and TX1) and two networks (NIN [49], VGG19M [71]), 37.7% and 26.1% inference stage memory and latency prediction errors respectively.

[8] models the latency and power consumption of the inference stage of various CNNs on the NVIDIA GeForce GTX Titan X GPU (cloud system). The authors develop a layer wise polynomial regression model and achieve average prediction errors of 12.1%, 11.7% and 2.79% in runtime, power and energy respectively across three different networks. They however do not model the memory consumption of inference.

The two approaches differ on the target setting of edge vs server grade devices. Furthermore although both use layer-wise regression models, [52] uses a linear model parameterised on the size of the expected matrix multiplications for each layer. On the other hand, [8] uses a polynomial model on layer hyper-parameters such as stride, kernel size etc. Table 2.3 provides a breakdown of the inference stage memory and latency prediction errors of the two works for

Network	NVIDIA GPU	Model	Memory Error (%)	Latency Error (%)
NIN	TK1	Augur	40.7	26.0
	TX1	Augur	50.9	29.2
	GTX Titan X	Neural Power	-	23.6
VGG19 [71]	TK1	Augur	28.0	21.4
	TX1	Augur	31.1	27.8
AlexNet	GTX Titan X	Neural Power	-	11.3
OverFeat [68]	GTX Titan X	Neural Power	-	1.4

Table 2.3: Comparison of state-of-the-art inference stage performance modelling techniques Augur [52] and Neural Power [8] on the metrics of prediction error of memory consumption of inference (**Memory Error** column) and prediction error of latency of inference (**Latency Error** column).

	TensorFlow	PyTorch	MXNet
VGG16	2.8	9.8	0.6
ResNet50	2.3	17.4	2.5
Inception V3 [77]	13.8	23.0	10.0

Table 2.4: Training stage memory consumption prediction error (%) for various network and framework combination for state-of-the-art CNN training memory prediction work [19].

various networks they evaluated on.

A direct comparison of the results between Augur and Neural Power is not trivial as even for the same network (NIN) they evaluate the latency prediction error on two different devices. However, a possible reason for Neural Power's lower prediction error when predicting the latency of inference for NIN could be that it bases its prediction on layer hyper-parameters which does not make any assumption on the type of algorithm (Sec.2.1.3) used to execute the network on device. Augur on the other hand always assumes matrix multiplications are utilised, which could mean that if the chosen execution framework (version of cuDNN) uses either the FFT or Winograd methods for a particular layer, this is not captured well in Augur's modelling methodology.

2.4.2 Training Stage Performance Modelling

Due to the limited investigation of efficient CNN training on edge devices, there has been little focus on modelling the latency and memory consumption of the training process on edge devices. On a server or distributed GPU system where training is commonly performed, the state of the art is [19] which proposes a model to predict GPU memory consumption (but not latency) of training. [19] develop an analytical model that captures both network and framework specific contributions to memory consumption during training and achieve memory prediction errors between 0.6% and 23.0% across different frameworks (TensorFlow, PyTorch and MXNet) and networks (VGG16, ResNet50, Inception V3). Their results are presented in Table 2.4.

They break down the training stage into the three different convolutions, as described in Sec.2.1.3; and for each, model the memory of the three different methods of performing a convolution. However, in addition to this, they also model the framework specific contributions to memory consumption by hand-crafting features that arise from expert-level knowledge on how the framework executes these operations. This poses a limitation as it prevents easy extensibility of this modelling technique to new devices and frameworks as it requires expert-level understanding of their execution semantics in order to create a model.

The results in Table 2.4 demonstrate on average across networks, 10.4pp and 12.4pp worse error rates when predicting the memory consumption of the PyTorch execution framework compared to the Tensorflow and MXNet frameworks respectively. A possible reason for this increased error rate for PyTorch could be due to the fact that PyTorch utilises a dynamic execution graph that is optimised at runtime while both Tensorflow and MXNet rely on static execution graphs that can be analysed before execution. Consequently, [19]'s approach of hand-crafting framework specific features might work better with Tensorflow and MXNet over PyTorch. Nonetheless, the error rates presented here set a state-of-the-art to compare against when evaluating proposed training stage memory modelling methodologies.

2.5 Related domains of research

The previous sections have provided the relevant background required for understanding the contributions of the PhD. This section now positions the research question answered with respect to other domains and discusses the similarities and differences between them.

The research questions answered by this PhD explore methods to perform the training of image classification CNNs on resource constrained edge devices in order to adapt the network's weights and topology to the observed data distribution. It aims to do so in the absence of ground-truth labels for the images that are observed. Additionally, it performs this adaptation under the assumption that the observed distribution differs from the training distribution only in the classes seen and not in the distribution of data points within a class. Furthermore, the classes of images observed upon deployment are assumed to be a subset of those used during training.

2.5.1 Partial Domain Adaptation (PDA)

In the context of image classification, the field of domain adaptation aims to adapt a network that was trained on a source domain but needs to classify images of a new target domain that differs in some manner from the source domain. This difference can be in the statistical properties of the data itself where for instance one domain contains images taken using cameras while the other domain contains images of the same set of classes, but sketched by an artist. In such cases, it is assumed that the label space of the two domains remain the same. Furthermore, the majority of works in the field of domain adaptation aim to perform this adaptation in a manner that is unsupervised, i.e. the labels of target domain images are unknown. The primary focus of unsupervised domain adaptation (UDA) is on learning domain-invariant feature representations without access to target domain labels, so that the same feature extractor and classifier combination can be utilised in both the source and target domains [66, 23].

However, as argued in Sec.1.1 and introduced in [11], with the increase in the sizes and number of classes present in training dataset such as ILSVRC'12, it is increasingly likely that both the

source domain and the target domain do not share the same label space, but rather the target domain contains only a subset of the classes present in the source domain. Consequently, the field of partial domain adaptation aims to develop methodologies that adapt a network trained on some source domain to be able to successfully classify images in a target domain that differs from the source domain in both the distribution of data within a class as well as the classes observed. Specifically, the label space of the target domain is assumed to be a subspace of that of the source domain. In this thesis, the three state-of-the-art works considered in the field of PDA are PADA [11], IWAN [87] and ETN [12].

These methodologies train a feature extractor that generates similar features, for source and target domain images of the same class, when the class of image is common to both domains and prevents such alignment of features when the image is expected to be from a class present only in the source domain. All three works split the problem into two parts, namely, a feature extractor and domain classifier. The domain classifier aims to identify in an unsupervised manner the difference between the source and target domains, while the feature extractor is trained to reduce the discrepancy between the domains. The domain classifier outputs weights that identify classes as source-only or shared; and the three works vary in their approaches to utilising these weights in training the feature extractor.

The research topic of this PhD is similar to the domain of PDA in that it aims to domain adapt a network in an unsupervised manner where the source and target domains differ in their label space. Furthermore, it is assumed that the target domain label space is a subspace of that of the source domain. The research topic however differs from PDA in that it makes a further assumption that only the label space differs between the two domains and not the distribution of the data itself. This assumption stems primarily from the hypothesised use cases of the proposed solution as introduced in Sec.1.1. For instance, it is unlikely that a CNN deployed on a robotic vacuum cleaner or an autonomous vehicle would be trained on images obtained from the device's camera, but upon deployment be exposed to images that are processed to look like they are sketches instead of real-world images.

Additionally, the topic of this PhD focuses on performing this adaptation on an edge device,

		No-PDA	ETN	IWAN	PADA
ILSVRC'12 → C84		69.69	83.23	78.06	75.03
OFFICE31	A → W	76.03	94.52	89.15	86.54
	A → D	91.12	95.03	90.45	82.17
	D → W	94.06	100.0	99.32	99.32
	D → A	83.62	96.21	95.62	92.69
	W → A	88.42	94.64	94.26	95.41
	W → D	98.69	100.0	99.36	100.0

Table 2.5: Comparison of domain adapted accuracy ($\bar{\alpha}$) of state-of-the-art Partial Domain Adaptation (PDA) approaches for the ResNet50 network. The domains are Caltech-84 (C84), Amazon (A), DSLR (D) and Webcam (W). The No-PDA column corresponds to a ResNet50 network trained on the source domain and not specialised in anyway.

which places memory consumption and latency restrictions on the process. Existing PDA approaches do not take into account such restrictions and hence when executed on GPUs, consume close to 8GB of memory. This would not be feasible to execute on devices such as the NVIDIA Jetson TX2 which has 8GB of memory that is shared between both GPU and CPU processes.

The rest of this section describes commonly used datasets to evaluate PDA methodologies and details the performance of PADA, IWAN and ETN on these datasets. Two commonly used adaptation tasks for evaluating PDA methodologies are the Caltech-84 and the Office-31 datasets. The Caltech-84 dataset corresponds to 84 classes that are common between the ILSVRC'12 and Caltech-256 [24] datasets. Office-31 dataset consists of three categories, each with 31 classes, corresponding to images from the website *Amazon* (A), a *DSLR* (D) and a *Webcam* (W). For the adaptation task, all pairwise combinations of these categories are considered where the source domain in the pair has all 31 classes, but the target domain only has a subset of 10 of the 31 classes. The performance of these works for ResNet50 [3] is provided in Table 2.5.

The results in Table 2.5 show that on average across tasks, ETN outperforms IWAN and PADA by 2.49pp and 4.64pp respectively. As discussed in [12], ETN improves on IWAN and PADA

[3]This is the only network evaluated as it is the only common network evaluated between the three works. Additionally, other network architectures evaluated in some of the papers are out-dated architectures such as VGG and AlexNet and hence are not separately considered.

by not just training the domain classifier using unlabelled feature maps, but also incorporating class discriminatory information obtained from the labeled source domain data. This improves the accuracy of the domain classifier used by ETN which in turn allows the adaptation process to focus better on target domain only classes.

2.5.2 Edge Device CNN Training

This section explores the related field of methodologies that aim to reduce the cost of training a CNN such that the process can be executed on resource constraint devices. To the best of our knowledge, the only two works which also explore methods of reducing the cost of performing training of CNNs with the goal of executing training on resource constraint devices are TinyTL [10] and PersEPhonEE [47]. Other works such as [64] also explore adapting CNNs on resource constrained devices such as Microcontrollers, but focus on the different problem setting of incremental on-device learning. Different to the problem setting introduced here, incremental learning focuses on enabling a deployed network to identify new classes that have not been observed before and improving a network's ability to identify such novel classes. PersEPhonEE was introduced in Sec.2.3.3 and so this section will focus on TinyTL.

The authors of [10] note that the main bottleneck of performing training on edge devices is the significant memory footprint of storing intermediate feature maps of the network for backpropagation. Consequently, they argue that instead of reducing the number of trainable parameters, it is important to reduce the memory footprint of feature maps that are saved during the forward pass for use in the backward pass (Sec.2.1.3).

To achieve this, the authors propose to fix the weights of the convolution layers of the original network, and instead update only biases and the weights of a smaller, newly proposed *Lite-Residual* layer. The *lite-residual* layer downsamples the feature maps generated by the original convolution layer, performs grouped and 1x1 convolutions on them, upsamples the output and adds it to the output of the original convolution layer. This reduces the main memory bottleneck of training which is the storing of intermediate feature maps, as only the downsampled feature

maps are stored. By doing so, TinyTL achieves up to 13.1× reduction in training memory consumption when compared to training the entire network.

However, a key distinction between TinyTL and the focus of the research in this PhD is that TinyTL solely focuses on reducing the training memory consumption. Whereas, as described in Sec.1.2.2, the work in this PhD also aims for the resulting adaptated network to have a smaller inference memory and latency footprint as compared to the originally deployed network. The proposed TinyTL approach actually increases the memory consumption and latency of inference of the resulting network as it adds an additional *lite-residual* layer per convolution layer.

Furthermore, the authors evaluate their methodology on a supervised transfer-learning scenario where the source and target domains do not share the same label space and there is access to ground-truth labels of the target domain. This is outside the scope of the work in this PhD and hence is not compared to.

This chapter introduced terminology for describing the topology of a CNN that will be utilised throughout the thesis as well as the Top1 classification accuracy on the ILSVRC'12 dataset, memory consumption for storing the model and FLOPs for performing inference for various commonly used CNN architectures. The results demonstrated that ResNet50 is the highest performing network in terms of accuracy and also the most widely prototyped network across various related works. Consequently, future chapters will evaluate the proposed methodologies on this architecture in order to perform comparisons with other state-of-the-art methodologies. However, both MobileNetV2 and MnasNet demonstrated strong performance when considering the trade-off between classification accuracy and inference stage execution costs. Hence, where possible one of these two networks are also considered when evaluating proposed methodologies.

The chapter also introduced various concepts such as VAEs and the algorithms used to execute training of CNNs on GPUs which are important to understand the contributions of the works that follow. Furthermore, the chapter identified L1-norm and TFO [58] as state-of-the-art pruning methodologies when considering for the trade-off between the accuracy loss from pruning and the cost of performing pruning itself. Furthermore, both OFA [9] and PersEPhonEE were identified as low cost alternatives to pruning that also reduce the execution cost of a

CNN while maintaining classification accuracy. Consequently, future chapters explore utilising these methodologies when reducing both inference and training stage costs of deployed CNN networks.

Finally, the chapter also provides the state-of-the-art performance for various performance modelling methodologies that define the error rates that works that follow aim to achieve and improve upon. The following chapters introduce the major contributions of this PhD and evaluate the manner in which they further the state-of-the-art.

Chapter 3

Motivating Edge Retraining

The goal of this chapter is to demonstrate the gains that can be achieved by performing network weights (through retraining) and topology adaptation on-device. Amongst the methodologies for topology adaptation discussed in Sec.2.3, this chapter focuses on utilising structured pruning to perform topology adaptation in a manner that is completely on-device. By exploring the gains and costs of doing so, the chapter acts as a feasibility study that motivates the need for adapting a network to the data that is observed while exploring the limitations that arise from performing this adaptation completely on-device. The novel contributions of this chapter are as follows.

Feasibility study of Data Aware Pruning and Retraining (DaPR) A quantitative analysis of the accuracy gains of adapting a network to the data it is processing as well as the cost of achieving these gains on an edge device. For the purpose of this analysis, the target device is the NVIDIA Jetson TX2 and the target dataset is the CIFAR-100 [43] image classification dataset. In doing so, the chapter motivates the chosen topic of research and guides future design decisions.

Autoprune An open-source framework [1] that automates the process of pruning a network for a wide range of network architectures. To the best of our knowledge, this is the only open-source tool that fully automates the process of identifying filters to prune, including those which are dependent on each other due to network architecture constraints, and shrinking the network to obtain memory and performance gains. In doing so, it enables automated deployment of pruning and retraining based solutions on any edge device.

The rest of the chapter is organised as follows. Section 3.1 outlines the proposed algorithm for data aware pruning and retraining that is utilised to perform this study. Section 3.2 describes the key features of the open-source automated pruning tool *Autoprune*. Section 3.3 evaluates the gains and costs of performing DaPR on an NVIDIA Jetson TX2.

3.1 On-device DaPR Methodology

This section proposes a DaPR algorithm that is then executed on the NVIDIA Jetson TX2 and profiled in order to quantify the gains and costs of performing on-device DaPR. The problem setting, as discussed in Sec.1.2 is as follows. Consider a network $\mathcal{M}_{\mathcal{D}}$ that has been trained offline on a large, extensive dataset with images from the set of classes $\mathcal{C}$. Upon deployment on an edge device, the model observes images of the same distribution as those used for training, but only from the set of classes $\mathcal{C}'$ where $\mathcal{C}' \subseteq \mathcal{C}$.

The goal of the proposed algorithm is to obtain a network $\mathcal{M}'_{\mathcal{D}}$ by pruning and retraining $\mathcal{M}_{\mathcal{D}}$ on-device such that the classification accuracy ($\bar{\alpha}$) of the adapted model on the observed data distribution is at least as good as that of $\mathcal{M}_{\mathcal{D}}$. Additionally, the algorithm must perform DaPR under the constraints that the static memory consumption (γ) and the latency of inference (ϕ) of the adapted model are less than the corresponding attributes of the original model, i.e. $\gamma(\mathcal{M}'_{\mathcal{D}}) < \gamma(\mathcal{M}_{\mathcal{D}})$ and $\phi(\mathcal{M}'_{\mathcal{D}}) < \phi(\mathcal{M}_{\mathcal{D}})$.

The proposed approach is shown in Algorithm 1. It performs a binary search over a range of predefined pruning levels in order to identify the smallest pruning level that results in a model

[1] https://github.com/ICIdsl/autoprune

Algorithm 1: Proposed DaPR Methodology

Inputs: Initial model $\mathcal{M}_{\mathcal{D}}$,
Subset $\mathcal{D}'$,
First pruning level to search p_0,
Upper (p_u) and Lower (p_l) bounds of pruning levels to search,
Set of pruning levels to search $\mathcal{P} = \{ip_i | i \in [\frac{p_l}{p_i}, \frac{p_u}{p_i}]\}$,
Target Top1 Test Accuracy $\bar{\alpha}_{target}$

Output: $\mathcal{M}_{\mathcal{D}'}$ with test accuracy $\bar{\alpha}_{max}$ and pruning level $p_0 \in \mathcal{P}$ that is the highest pruning level s.t. $\bar{\alpha}_{max} \geq \bar{\alpha}_{target}$

Finetune $\mathcal{M}_{\mathcal{D}}$ for n_f epochs on $\mathcal{D}'$ to obtain $\mathcal{M}^f_{\mathcal{D}}$
while p_0 *changes across iterations* **do**
  Prune $\mathcal{M}^f_{\mathcal{D}}$ to pruning level p_0 according to chosen pruning strategy
  Retrain the pruned model for n_r epochs on $\mathcal{D}'$ and record model $\mathcal{M}_{\mathcal{D}'}$ and corresponding test accuracy a_{max} for the model with highest validation accuracy
  if $a_{max} < a_{target}$ **then**
    $p_u = p_0$
    // Reduce pruning level to the nearest multiple of p_i to the midpoint between p_l and p_0
    $p_0 = p_i \times \lceil \frac{\lfloor \frac{p_l + p_0}{2} \rfloor}{p_i} \rceil$
  else
    $p_l = p_0$
    // Increase pruning level to the nearest multiple of p_i to the midpoint between p_u and p_0
    $p_0 = p_i \times \lceil \frac{\lfloor \frac{p_u + p_0}{2} \rfloor}{p_i} \rceil$

that satisfies the above constraints. The algorithm takes as input the initially deployed model, the observed data distribution upon deployment ($\mathcal{D}'$), a set of pruning levels to be considered and the target classification accuracy that it must at least achieve. The algorithm utilises the L1-norm pruning metric [48] to perform structured pruning of the model as this metric has the lowest costs associated with performing pruning (Sec.2.3.1) and the goal is to execute the process on an edge device with limited memory and compute capabilities.

The process starts at some initial pruning level p_0 and retrains the pruned network, via back-propagation and stochastic gradient descent (SGD), for n_r epochs after pruning. The retraining is performed using data collected on the edge, and the accuracy of the retrained pruned network is evaluated on a validation dataset. The training and validation datasets are created from the data collected during deployment. It is important to note that in order to construct these datasets, the proposed methodology here makes the ideal case assumption of availability of ground-truth labels of the data observed on the edge device. Making this assumption allows

to remove any errors in classifying the data observed on the edge device and focus solely on the accuracy gains of retraining a network to adapt it to the data it is processing. Such an assumption is not guaranteed to hold true in a real deployment scenario.

If the validation accuracy is greater than that of the original model $\mathcal{M}^f_{\mathcal{D}}$, the pruning level is increased (heavier pruning) and vice versa if the validation accuracy is lower. As the initial pruning level p_0 does not necessarily correspond to 0% pruning, it is possible that each iteration of pruning searches models with pruning levels $< p_0$ (larger models) if the achieved validation accuracy never exceeds that of the original model. However, if progressively larger pruning levels are searched, the time to perform this search and the memory footprint of searched models decreases as progressively smaller models are used. The algorithm converges when the pruning level utilised does not change over iterations of the binary search. The result of performing this search is identifying the smallest model that can perform at least as well as the original deployed model in terms of classification accuracy on the observed data distribution.

In order to successfully deploy this algorithm, it is necessary to be able to automate the process of generating a new, smaller pruned network after removing the filters to be pruned. The challenge with doing this is that depending on the type of network being pruned, the building blocks used (Sec.2.1.2) create dependencies between filters such that pruning one of the dependent filters requires pruning all of them. However existing frameworks that are used to experiment with pruning methodologies, Distiller [92] and Mayo [90], do not provide automation capabilities for identifying these dependencies for a wide range of networks. For instance, Distiller provides dependency identification functionality just for ResNets, while Mayo does not provide any such functionality. Additionally, the output networks from both these frameworks do not actually shrink the network itself but rather just mask the pruned filters with zeros such that the effect of pruning on accuracy can be evaluated, but not that on memory consumption and latency. The following section presents a tool that was developed to overcome these limitations.

Building Block	Networks	Dependencies
Sequential	AlexNet[44], VGG[71]	No
Residuals	ResNet[28], MobileNetV2[67], ResNeXt[84], DenseNet[31]	Yes
Depth-wise Separable	MobileNetV2[67], Xception[16], EfficientNet[79]	Yes
Expand&Concatenate	SqueezeNet[33], GoogLeNet[75]	No

Table 3.1: Summary of architectural building blocks, networks that contain them and whether or not they have dependent convolutions.

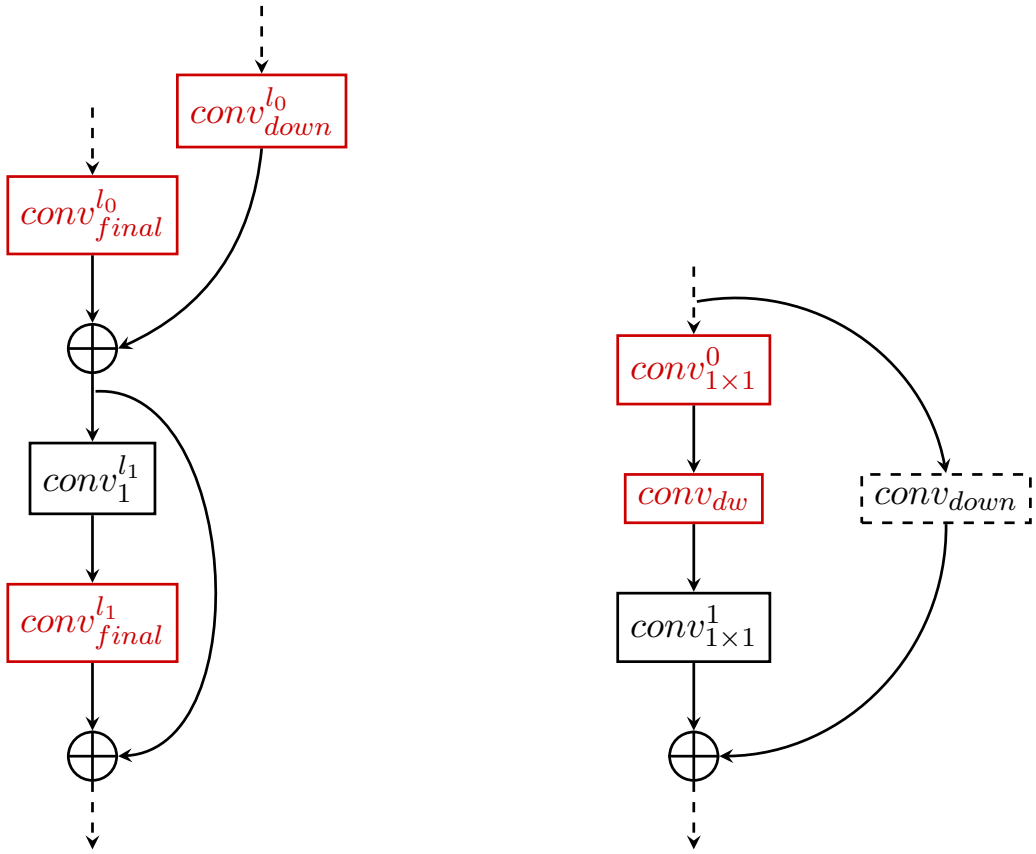
3.2 Autoprune

Autoprune is an open-source tool that enables automated dependency detection and network shrinking for a wide range of network topologies. It is proposed as a solution to build and prototype fully automated DaPR systems. Sec.3.2.1 first introduces what is meant by dependencies between filters within CNN networks and how the tool automates detecting these dependencies. Sec.3.2.2 then describes the process by which the tool shrinks the network.

3.2.1 Pruning Dependency Calculation (PDC)

Modern CNNs are usually constructed by combining a set of basic structural modules that contain multiple convolutions connected in a specific way. Most of the networks are based on the modules found in AlexNet [44], ResNets [28], MobileNetV2 [67], and SqueezeNet [33]. These four networks incorporate amongst them the most commonly used structural modules - *Sequential connectivity* in AlexNet, *Residuals* in ResNets and MobileNetV2, *Depth-wise Separable* convolutions in MobileNetV2 and *Expand & Concatenate* patterns such as in the Fire module (Fig.2.2) in SqueezeNet. Moreover, certain connectivity patterns result in pruning dependencies that necessitate the pruning of identical filters across dependent layers. The PDC automates the process of recognising such dependencies within a network. Table 3.1 summarises the networks that use various building blocks and if that particular building block contains dependent convolutions.

Networks with only *Sequential connectivity* patterns such as AlexNet and VGG [71] do not have any dependent filters. The same applies to *Expand & Concatenate* patterns in networks such



(a) Residual Blocks **(b)** MBConv

Figure 3.1: Structural blocks that require consideration of dependencies when pruning. Red highlighted convolutions are dependent convolutions.

as SqueezeNet. In both these cases, each filter in the network can be pruned independently without affecting the network's structure.

ResNets are made of residual blocks as shown in Fig.3.1a. The figure shows two residual blocks connected to each other labelled as layers l_0 and l_1. The circle with the plus represents an addition operator where it performs a channel-wise addition of the inputs. Some residual blocks contain downsampling layers ($conv^{l_0}_{down}$). These are 1×1 convolutions that ensure that the number of channels output from the final convolution of a residual block match the number of channels output from the residual connection. This ensures that the same number of channels are present in both inputs to the addition operation. Due to the addition operator requiring the same number of channels in both inputs, the highlighted blocks in red are all dependent on

each other, i.e. pruning a single channel in anyone of them requires pruning the corresponding channel in all of them.

Note that this dependency chain of $conv^{*}_{final}$ layers occurs in groups which are split by residual blocks which have downsampling layers. When a residual block with a downsampling layer is encountered, we can reset the dependency chain as the $conv_{final}$ for the layer with the downsampling block can be pruned arbitrarily and the respective $conv_{down}$ can be pruned correspondingly to ensure the number of channels match. The PDC automatically detects these dependencies and creates groups of dependent convolutions that need to be pruned together in order to maintain network topology integrity. Due to the lack of automation in the past, existing literature varies in the handling of this dependency where works such as [50] choose to not prune the last convolutional layer while others such as [48] choose to enforce this dependency manually.

MobileNetV2 is made of MBConv blocks which contain depth-wise separable convolutions as shown in Fig.3.1b. Note that the MBConv block also contains a residual connection with some blocks having the downsampling convolution ($conv_{down}$) while others do not. However, this is handled by the PDC in the same way as in the discussion of residuals above and does not require any further considerations for the MBConv block.

The MBConv building block first has a 1×1 convolution ($conv^{0}_{1\times1}$), followed by a 3×3 depth-wise convolution ($conv_{dw}$) followed by another 1×1 convolution ($conv^{1}_{1\times1}$). The depth-wise convolution is different from regular convolutions in that each filter only acts on one channel of the IFM. This means that the number of filters in $conv_{dw}$ must always match the number of IFM channels into that layer. Consequently the same filters need to be pruned in $conv_{dw}$ and the layer(s) feeding it; in this case, $conv^{0}_{1\times1}$. This dependency is also enforced by the PDC when MBConv modules are present in the model.

The PDC is implemented using PyTorch's TorchScript functionality which returns the PyTorch execution graph. This graph specifies the hardware operators called by PyTorch for each type of CNN layer and also the connectivity between the different operators. For instance, PyTorch's execution backend library is called *aten* and a convolution operation is represented by the

aten::convolution node in the execution graph. Similarly there are nodes for addition operators and concatenation operators. The PDC uses this graph to extract information on the connectivity between various convolutions and enforces the required dependencies across convolutions. This way of developing the PDC means that once a dependency type is implemented, such as the dependency caused by addition nodes, dependencies in any network architecture which contains addition nodes will automatically be recognised by the PDC. For instance, within the networks specified here, apart from specifying a new dependency for a depth-wise convolution, no changes to the PDC were required in order to extend from ResNets to MobileNetV2 as the dependency caused by residuals was already catered for. This automation makes *Autoprune* extensible and quick to use as it removes the necessity for users to manually specify each dependent convolution, which other tools such as Distiller [92] require.

3.2.2 Network Shrinking

The PDC identifies groups of groups of convolutions that are dependent and passes these details on to the next stage of *Autoprune* which prunes the filters and shrinks the size of network. This stage takes as input the network, the groups of dependent convolutions, a metric that is to be used to identify which filters to prune and the desired pruning level. The metric could be something such as the L1-norm [48] or the Taylor First Order pruning metric [58].

Pruning the network is performed by ranking all the filters based on the chosen metric, and removing filters until the desired percentage of pruning has been achieved. Removing a filter from one of the dependent layers removes the corresponding filter from all of the layers in that dependency chain. Furthermore, the effect of removing a filter is propagated through the network as each filter corresponds to an IFM channel for the next layer. After identifying which filters need to be pruned, the tool shrinks the network and copies across the weights from the original network that were left unpruned.

The shrinking stage allows the user to specify their own metrics for selecting filters to prune. Combining this with the extensiblity of the PDC in detecting dependencies, *Autoprune* allows

for new architectures and different pruning techniques to be experimented on with ease.

3.3 Evaluation

The chapter so far has introduced the algorithm that is proposed for the purpose of evaluating data-aware pruning and retraining of a CNN on edge devices. It has also introduced the tool *Autoprune* that enables this algorithm to be deployed in an automated manner. The problem setting used in the evaluation is characterised as follows. A model $\mathcal{M}_{\mathcal{D}}$ is trained offline on a dataset $\mathcal{D}$ with images from the set of classes $\mathcal{C}$. Upon deploying this model on an edge device, the observed images and their corresponding classes are characterised by a dataset $\mathcal{D}'$ with set of classes $\mathcal{C}'$. It is assumed that $\mathcal{C}' \subseteq \mathcal{C}$. The results in this section are produced by applying Algorithm 1 to $\mathcal{M}_{\mathcal{D}}$ in order to obtain the smallest model $\mathcal{M}'_{\mathcal{D}}$ such that the accuracy of $\mathcal{M}'_{\mathcal{D}}$ on $\mathcal{D}'$ is at least as good as that of $\mathcal{M}_{\mathcal{D}}$ on $\mathcal{D}'$.

This section will evaluate the performance of this on-device adaptation methodology in terms of the following metrics.

1. The improvements in classification accuracy that can be achieved when compared to models deployed after **Subset Agnostic Pruning**. Subset Agnostic Pruning refers to pruning and retraining a model using the entire dataset $\mathcal{D}$, without any on-device adaptation. This approach serves as baselines to compare the proposed DaPR methodology as it does not finetune the network being deployed to the observed data distribution ($\mathcal{D}'$) in any way.

2. The reduction in inference time (ϕ) and model static memory consumption (γ) that can be achieved by performing on-device DaPR.

3. The memory consumption (Λ) and latency (Θ) of performing such on-device DaPR.

By doing so, the results in this section motivate the feasibility of performing on-device domain adaptation, the focus of the remainder of this thesis.

Subset Name	Coarse Classes	# Fine Classes
Aquatic	aquatic_mammals, fish	10
Indoor	food_containers, household_electrical_devices, household_furniture	15
Outdoor	large_man-made_outdoor_things, large_natural_outdoor_scenes, vehicles_1, vehicles_2, trees, small_mammals, people	35
Natural	flowers, fruit_and_vegetables, insects, large_omnivores_and_herbivores, medium_mammals, non-insect_invertebrates, small_mammals, reptiles	40
Random	aquatic_mammals, fish, flowers, fruit_and_vegetables, household_furniture, large_man-made_outdoor_things, large_omnivores_and_herbivores, medium_mammals, non-insect_invertebrates, people, reptiles, trees, vehicles_2	65

Table 3.2: Subsets tested along with the coarse classes that were included in the subset and the number of fine classes per subset

3.3.1 Setup and Hyperparameters

The target edge device is the NVIDIA Jetson TX2. Through cross-validation, the values chosen for n_f and n_r from Algorithm 1 are 5 and 25 respectively to ensure that the accuracy plateaus before retraining ends. The range of pruning levels searched are from $p_l = 5\%$ to $p_u = 95\%$ in increments of $p_i = 5pp$. The first pruning level searched $p_0 = 50\%$.

In order to explore the problem setup discussed in Section 1.2, it is necessary to create subsets $\mathcal{D}' \subseteq \mathcal{D}$. In this evaluation, $\mathcal{D}$ is the entire CIFAR-100 dataset, and five different subsets $\mathcal{D}'$ are tested. The CIFAR-100 dataset is hierarchically categorised into 20 "coarse-classes" which contain 5 "fine-classes" each. In order to create the subsets, groups of course classes were selected. The subsets created and the classes within them are described in Table 3.2. The first four subsets are hand selected to have coherent semantic meaning and across the four of them span the entire CIFAR-100 dataset. The fifth subset is randomly generated. The networks tested on are AlexNet, ResNet20, MobiletNetV2, and SqueezeNet for all the subsets listed in Table 3.2. As discussed in Sec. 3.2.1, these four networks cover all the commonly found building blocks. Furthermore, when training these networks on the CIFAR-100 dataset, the memory consumed is less than the available 8GB of memory on the NVIDIA Jetson TX2 allowing the DaPR methodology to be prototyped.

The chosen metric to rank filters was the L1-norm as it has been established to show competitive performance even against data-aware metrics [57] despite being computationally inexpensive.

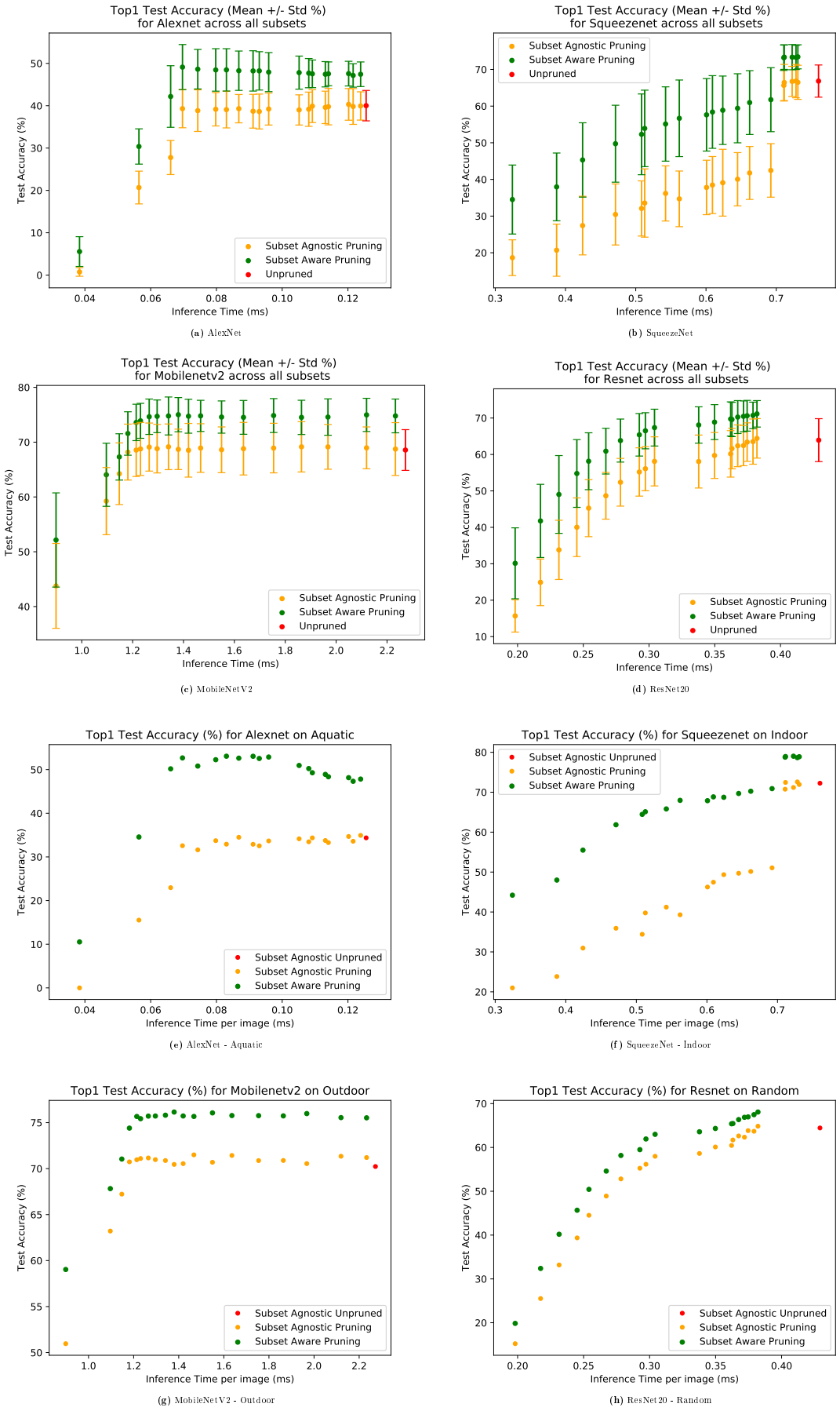
The learning rate schedule is set to start at the final learning rate employed when $\mathcal{M}_{\mathcal{D}}$ is trained on $\mathcal{D}$. After pruning, the learning rate is increased to the second highest learning rate that is employed when $\mathcal{M}_{\mathcal{D}}$ is trained on $\mathcal{D}$. Following this, the learning rate is decayed at epochs 15 and 25 by the same decay rate that is used when $\mathcal{M}_{\mathcal{D}}$ is trained on $\mathcal{D}$.

The batch size used is 128, and the training dataset is split into a training and validation set in the 80:20 ratio. The accuracy values reported in this section are the test set accuracy corresponding to the model with the best validation accuracy. Standard data augmentation techniques for CIFAR-100 are used for the training set such as random crop, rotation and flip. All experiments are repeated 5 times and the mean, and where applicable, the standard deviation of these repetitions is reported.

3.3.2 Classification Accuracy ($\bar{\alpha}$) and Inference Latency (ϕ)

The graphs in Fig. 3.2 shows the trade-off between the achieved test accuracy ($\bar{\alpha}$) and inference time (ϕ) for all pruning levels between 5% and 95% on various subsets and networks. The red dot in each of the figures (**Unpruned**) display the performance of $\mathcal{M}_{\mathcal{D}}$ on $\mathcal{D}'$. The orange dots in the figures (**Subset Agnostic Pruning**) display the performance of a model that was pruned without any finetuning on $\mathcal{D}'$ and retrained on the entirety of $\mathcal{D}$ for $n_f + n_r$ epochs. The green points (**Subset Aware Pruning**) display the performance of a model that was finetuned for n_f epochs on the subset $\mathcal{D}'$, then pruned and subsequently retrained on $\mathcal{D}'$ for n_r epochs. As the cost of performing the adaptation on-device is not being evaluated here, instead of performing a binary search as proposed in Section 3.1, the results in Fig.3.2 are obtained by sweeping all pruning levels between p_l and p_u in increments of p_i. It acts as an *oracle* that for each pruning level compares the performance of the data-aware method (green points) with the data agnostic methods (orange and red points).

Figs.3.2a-3.2d use error-bars to display the mean and standard deviation of the test accuracy across all 5 subsets for a given pruning level and network. For many cases, the worst performing subset aware strategy performs better than the best performing subset agnostic strategy with an



(a) AlexNet (b) SqueezeNet (c) MobileNetV2 (d) ResNet20

(e) AlexNet - Aquatic (f) SqueezeNet - Indoor (g) MobileNetV2 - Outdoor (h) ResNet20 - Random

Figure 3.2: Trade-off of Test Accuracy vs Inference Time for various levels of pruning. The error bars show the mean and standard deviation of test accuracy obtained per level of pruning across all the 5 subsets. The graphs in Fig.3.2e-3.2h only show select combinations of networks and subsets in the interest of space. Results for all combinations of networks and subsets are provided in Appendix A.1.1

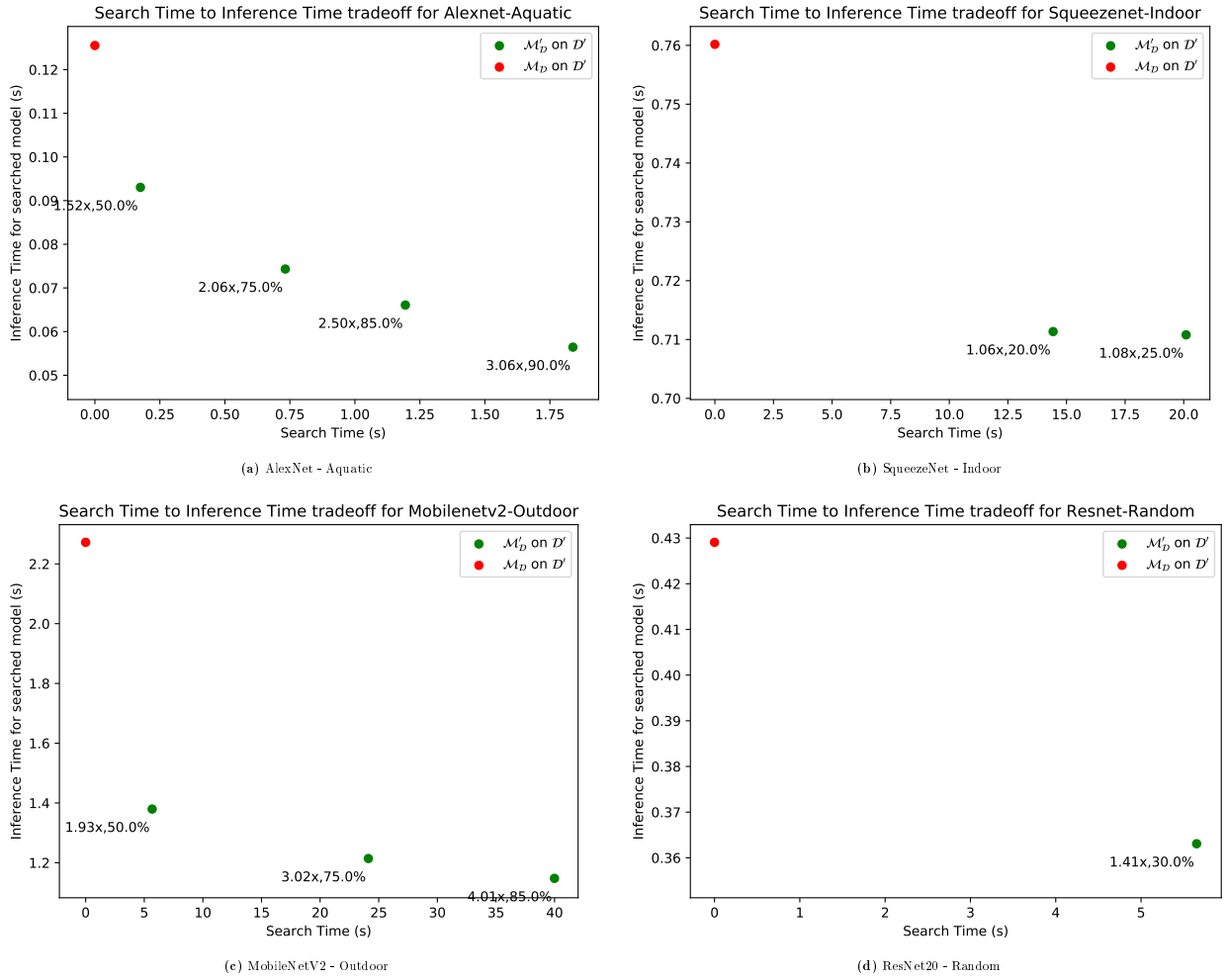
average improvement in test accuracy of 10.2pp and maximum improvement of 42.0pp over all pruning levels, networks and datasets. Furthermore, this improvement in test accuracy tends to increase as more of the network is pruned (lower inference time) thus further motivating the need to perform data-aware pruning and retraining as better accuracy can be achieved for smaller models.

Figs.3.2e-3.2h show the same trade-off but for specific combinations of network and subset. From top left to bottom right, the size of the subset increases and intuitively the gap between subset-aware and subset-agnostic pruning and retraining decreases as the subset $\mathcal{D}'$ converges towards $\mathcal{D}$. Nonetheless for all pruning levels, the performance of the subset-aware strategy outperforms subset-agnostic pruning and finetuning.

The *oracle* results discussed above show the gains that can be obtained for a wide range of pruning levels, however only some of the models perform better than the baseline performance of $\mathcal{M}_{\mathcal{D}}$ on $\mathcal{D}'$ (red points). The goal of the proposed DaPR methodology is to identify the smallest model that does so. The methodology proposed in Section 3.1 is an efficient search strategy to achieve this goal. However, as seen in Fig.3.2, there are multiple pruning levels that result in models that lie at or above the red points. Consequently, there exists a trade-off between the size of the model and the time spent in performing the search. This is represented in Figs.3.3a-3.3d. Each figure forms a pareto-frontier describing the trade-off between the time to search for a smaller model (Θ) and the inference time of this model upon deployment (ϕ). Models found earlier are a result of stopping the binary search process before the pruning levels searched plateaus, but the classification accuracy is higher than that of $\mathcal{M}_{\mathcal{D}}$. The labels next to the points show the relative improvement in GOps (ρ) compared to the unpruned model (-x times) and the pruning level of the model (γ).

3.3.3 Model adaptation latency (Θ) and memory (Λ)

This section evaluates the memory consumption and latency of performing on-device DaPR. The memory utilisation of the GPU (Λ) does not exceed 2GB which lies far below the maximum



(a) AlexNet - Aquatic

(b) SqueezeNet - Indoor

(c) MobileNetV2 - Outdoor

(d) ResNet20 - Random

Figure 3.3: Trade-off between search time per minibatch and inference time performance of searched model. All models shown here have no accuracy loss compared to $\mathcal{M}_{\mathcal{D}}$ infered on $\mathcal{D}'$. The labels next to the points show (improvement in GOps (-x times), pruning level (%)) of that model. Only a few combinations of network and subset are provided here in the interest of space. Appendix A.1.2 provides results for all combinations of network and subset.

Metric	Range of Values
$\bar{\alpha}(\mathcal{M})$	+10.2pp on average across pruning levels, networks and subsets w.r.t $\mathcal{M}_{\mathcal{D}}$
$\rho(\mathcal{M})$	Up to 4.18x reduction w.r.t $\mathcal{M}_{\mathcal{D}}$
$\phi(\mathcal{M})$	Up to 2.22x reduction w.r.t $\mathcal{M}_{\mathcal{D}}$
$\gamma(\mathcal{M})$	Up to 90% reduction w.r.t $\mathcal{M}_{\mathcal{D}}$
$\Delta(\mathcal{P})$	At most the memory consumption of $\mathcal{M}^f_{\mathcal{D}}$, as this is the starting point of Algorithm 1.
$\Theta(\mathcal{P})$	Up to 40min on average across networks and subsets.
$\Lambda(\mathcal{P})$	Less than 2GB

Table 3.3: Summary of metrics of interest for the proposed DaPR algorithm (Algorithm 1).

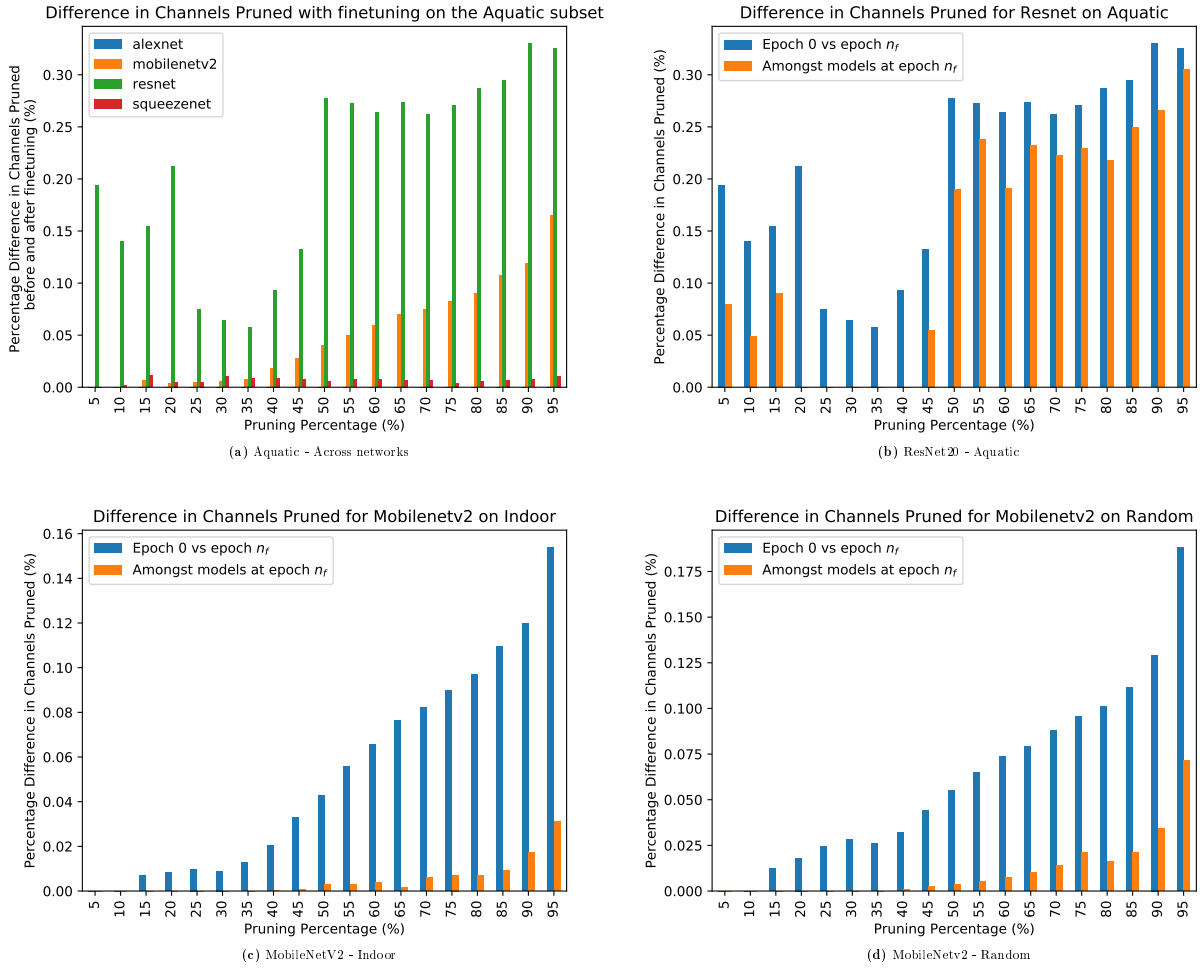
memory availability of the TX2 of 8GB. The number of minibatches searched depends on both the amount of data available and the time budget allocated during deployment to perform finetuning. However, the results presented here show that such DaPR methodologies can be performed on edge devices within a reasonable time budget.

It should be noted that as the time budget allocated for performing the pruning in this work is much shorter than the budget required by the current state-of-the-art pruning methods described in Section 2.3, and as such a direct comparison to those works is not meaningful. The existing methods perform pruning and retraining before deployment, and do not adapt the network once deployed. Furthermore, none of these works except [74] report results on subsets of CIFAR-100. [74] however do not provide details of the classes present in each subset, making a direct comparison infeasible.

Table 3.3 summarises the results discussed above for all the metrics introduced in Table 1.1. Across all subsets of the CIFAR-100 dataset, performing the search described in Section 3.1 results in sub-minute search times per minibatch for up to 2.22x improvement in inference latency, 4.18x improvement in GOps and 90% memory footprint reduction.

3.3.4 Filter Selection

A further investigation was carried out on the relationship between the type of the structural block and the impact of finetuning to its parameters. Fig.3.4a shows the percentage difference between the filters that were selected to prune if pruning were to take place at epoch 0 and at



(a) Aquatic - Across networks

(b) ResNet20 - Aquatic

(c) MobileNetV2 - Indoor

(d) MobileNetv2 - Random

Figure 3.4: Percentage difference in channels pruned at various points of DaPR for different network and subset combinations

epoch n_f after n_f epochs of finetuning on $\mathcal{D}'$. In this case $\mathcal{D}'$ is the *Aquatic* subset. The results show that weights in ResNet20 and MobileNetV2 are highly sensitive to finetuning on a subset, but AlexNet and SqueezeNet show negligible change in the filters selected to prune before and after the finetune stage.

To analyse if these changes in selected filters are due to the finetuning process or random variation of the weights during training, ResNet20 and MobileNetV2 were further explored. The blue bars in Figs.3.4b- 3.4d show the difference in channels pruned at epoch 0 versus epoch n_f. The n_f epochs of finetuning was performed 5 times per network and subset from the same starting point, thus resulting in 5 models at epoch n_f. The orange bars show the results for the average difference in channels pruned across all $\binom{5}{2}$ combinations of models at epoch n_f. Fig.3.4b shows that with ResNet architectures, there is a large random variation due to training (high percentage orange bars), and comparable percentages between the orange and blue bars suggest that the effect of the finetuning process in tuning the topology of the network to the data processed is limited. However for the MobiletNetV2 architecture, Figs.3.4c -3.4d show that the finetuning process effectively tunes the topology to the data processed.

These results suggest that Residual structural blocks (common feature between ResNet and MobileNetV2) make the weights sensitive to finetuning, but the depth-wise convolution (unique to MobileNetV2) allows for data-aware discrimination between filters based on just their L1-norm. Consequently, the metric that needs to be used for data dependent tuning of architectures may vary depending on the structural blocks present.

3.4 Conclusion

Acknowledging the shift in paradigm from server based compute to increased edge processing, this chapter provides a solution that performs on-device DaPR, an analysis of the accuracy gains achievable by performing such data-aware retraining, the costs of performing this process on an embedded device, and a tool that allows for rapid deployment and research of on-device DaPR methodologies. The results show that the gains in accuracy obtained by retraining

to the subset are significant and on average 10.2pp across a wide range of subsets, networks and pruning levels. In terms of costs, the search for a pruned model that achieves a given target accuracy can be performed on an NVIDIA Jetson TX2 with sub-minute search times per minibatch for up to a 2.22x improvement in inference latency and 90% reduction in memory footprint when utilising the CIFAR-100 dataset.

Furthermore, analysis of the selected filters show that for the MobileNetV2 architecture, the L1-norm is an effective yet computationally inexpensive metric to tune a network's topology to the data being processed and suggests that different architectures may require different metrics to make pruning of the network dependent on the data it is processing.

Additionally, the extensible and customisable tool (*Autoprune*) developed to perform this on-device pruning and retraining allows users to prune a wide range of CNN architectures and realise the memory and runtime gains immediately on any platform of their choice in a fully automated manner. To the best of our knowledge, this is the only open-source tool that that allows the user to automatically shrink (through pruning) and deploy a network for such a wide variety of network architectures as well as target hardware. There are commercial tools [25] that can perform this function, however tend to focus deployment on specific target hardware architectures such as FPGAs. These combination of features allow for rapid deployment of pruned models on any device without user intervention and thus makes it a unique open-source tool for pruning research.

However, it is important to keep in mind that the work in this chapter focuses on validating the feasibility and motivating further research into novel methods for on-device domain adaptation. The results in this chapter are only "best case" and not completely representative of real-world scenarios for the following reasons:

1. The dataset used is CIFAR-100 which is made up of 32x32 sized images. This is significantly smaller than images observed in practical scenarios. For instance, the most widely used image classification dataset, ILSVRC'12 [17], is made up of 224x224 sized images and hence networks that classify ILSVRC'12 images requires significantly larger memory

and compute resources to train.

2. An important assumption made here is that ground-truth labels of images observed after deployment are available. This is unlikely in real-world scenarios, but the results in this chapter provide an upper bound on the gains that can be made by adapting a network to the data it is processing.

Consequently, the chapters that follow provide solutions that tackle both these issues and hence can be deployed in more real-world scenarios.

Chapter 4

perf4sight : Using performance modelling to enable on-device DaPR

Chapter 3 introduces an algorithm for on-device data aware pruning and retraining as a solution to adapt a network's architecture and weights to the observed data distribution. The pruning strategy used is the L1-norm of the filters and target device evaluated on is the NVIDIA Jetson TX2. However as demonstrated in Table 2.2, the L1-norm metric, although providing a much lower cost of pruning, performs 14.5pp worse in terms of model size reduction and 1.5x worse in terms of FLOPs reduction than TFO pruning [58] for the same loss in classification accuracy. On the other hand, the resource constraints of edge devices, and the memory and latency intensive nature of TFO pruning means that this cannot be feasibly applied directly on an edge device.

Consequently, there is a need to explore alternative methodologies to pruning in order to perform on-device topology adaptation of CNNs. One such methodology, which is the focus of this chapter, is utilising a Once-For-All (OFA) network [9] (Sec.2.3.2) for this purpose. The OFA network is a large super-network from which sub-networks with the desired memory consumption and latency (either of inference or training) can be sampled. This is done by sharing weights between the sub-networks such that more than 50,000 sub-networks can be sampled from a single super network of size 192 MB. During the training stage, which is performed

once offline, the entire super network is trained first. Following this, progressively smaller sub-networks are randomly sampled for training at some interval which is a hyperparameter of the process.

As many sub-networks are already trained on the original training dataset offline, OFA enables quick access to a large number of high-performing pruned topologies, with significant variation in memory footprint and latency. This creates the possibility for a system that can vary its network architecture over time (by sampling different sub-networks) in line with varying memory and latency requirements. Furthermore, as the searched sub-networks have a high initial classification accuracy, very few iterations of on-device retraining are required to tune the weights of the network to the observed data distribution.

However, the variation in the memory consumed and latency of training for the sampled sub-networks can be significant. For instance, for 100 sampled sub-networks of similar inference FLOPs, the memory consumed during training can vary by up to 25%. Thus, in order to utilise OFA for on-device topology adaptation, an estimate of the training memory and latency of each sampled sub-network is necessary in order to select networks that can be retrained on-device within the available time and memory budgets.

One approach to perform this estimation is to profile each architecture on the device. Not only is this approach time consuming, but edge devices such as the NVIDIA Jetson TX2 tend to have shared CPU and GPU memory systems. This means that a process running on one can prevent processes from starting up on the other due to a lack of available memory. Such an event can be catastrophic in safety critical applications such as autonomous driving. For instance, performing retraining of a CNN using the GPU could prevent the start of another safety critical process either on the CPU or GPU. Hence there is a requirement to accurately predict the memory consumption and latency of CNN inference and training without running the process itself.

Motivated by the goal of enabling on-device adaptation of a model to the observed data distribution using an OFA network, this chapter provides a novel solution for predicting the memory consumption and latency of training a CNN on an edge GPU. The novel contributions are as

follows:

1. The development of performance models that utilise decision trees to predict for a given training batch size, network architecture, device and training framework; the memory consumption and latency of the training process.

2. An extensible open-source PyTorch tool [1] (perf4sight) that automates both the profiling and modelling processes.

3. A methodology that utilises an OFA network along with the developed performance models on an edge device to perform on-device DaPR.

The rest of the chapter is organised as follows. Sec.4.1 introduces the performance modelling problem. Sec.4.2 describes the proposed modelling methodology. Sec.4.3 evaluates the performance of the developed models and provides a case study that performs DaPR using a combination of an OFA network and the developed performance models.

4.1 Problem Description

Problem Statement Given a target device and machine learning framework (PyTorch, Tensorflow etc.), construct models that take as input a CNN network architecture and mini-batch size and accurately predict the memory consumption and latency of training the CNN.

The main challenge associated with this problem is that the libraries used to perform CNN training on off-the-shelf hardware use proprietary heuristics to decide between the three convolution algorithms discussed in Sec.2.1.3. Furthermore the algorithms used vary between networks, target devices and machine learning frameworks, while also being significantly different in their resource consumptions. Consequently, the developed models need to be able to account for this variability in execution and black-box behaviour of the libraries. This chapter develops models to predict the following two attributes of CNN training.

[1] https://github.com/ICIdsl/performance_modelling.git

Training memory consumption (Γ) The total memory consumed by the training process. In unified memory systems where the CPU and GPU share a memory space, this attribute also captures the memory allocated by CPU operations such as pre-processing of the input data.

Mini-batch training latency (Φ) The time taken to perform the forward and backward passes for one mini-batch of size bs. Frameworks such as PyTorch can overlap the tasks of pre-processing data (eg. normalisation) for the following mini-batch with the execution of the current mini-batch. Hence, only the time taken for execution of a mini-batch and not the time taken for data preparation is measured. The time taken to perform the gradient update step in stochastic gradient descent (SGD) is also included in the measurement. The total training time for a system can be estimated by multiplying Φ with the number of mini-batches of training.

4.2 Model Construction

The values of Γ and Φ described in Sec.4.1 depend on the network, training framework and target device. Network architecture related information can be obtained by analytically modelling the expected memory consumption and operations of training on a per layer basis. However, information related to the training framework and target device such as layer wise choice of convolution algorithm (Sec.2.1.3) can only be obtained by profiling networks on the target device.

A layer-wise profiling system (as seen in [52, 8]) obtains memory, latency and power predictions per layer and a prediction for the entire network is constructed from the layer estimates. This approach suits the case of modelling inference as there is only one operation performed per layer (Eq.2.1), allowing for the modelled attributes of a single layer to be treated in isolation. However in the case of training, three operations are performed at different times i.e. one on the forward pass and two on the backward pass. Even if only a single layer is being profiled, frameworks such as PyTorch speculatively allocate more memory than is required for just that layer as it is expected that an entire network rather than a single layer will be executed when performing

training. This makes profiling the memory and latency data for a single layer during training irrelevant. Alternatively, approximating all operations as matrix multiplications and profiling random sizes as in [52] would lose information regarding cuDNN's heuristics that choose between the three implementations of a convolution on a layer and operation-wise (Eq.2.1,2.2,2.3) basis.

As a solution, this work proposes a network-wise profiling strategy where each datapoint corresponds to the training of an entire network, instead of a single layer. Although this approach limits the number of data points that can be collected, the variation in memory and latency between batch sizes as well as network topologies can be utilised as a way to collect a sufficient number of datapoints required by the target modelling methodology. Furthermore, in contrast to [52] and [8] the proposed work focuses on modelling the memory consumption and latency of the training stage instead of that of the inference stage.

Similar to [19], this work uses an analytical model to capture network specific terms, i.e. weights, activations and temporary memory, required by the various convolution algorithms described in Sec.2.1.3. These are static features that are unlikely to change as both the training process and methods of performing convolutions on GPUs are well established fields. However unlike [19], this work complements the analytical modelling with a profiling and learning methodology to account for the constantly changing framework and device specific terms. For instance, terms that arise from the choice of cuDNN [35] version are proprietary and cannot be accurately modelled through analytical approaches. Furthermore, building an analytical model for targeting different frameworks (PyTorch, Tensorflow etc.) requires expert-level understanding of framework specifics and significant handcrafting of features which is both more time consuming and at risk of quickly becoming out-dated as frameworks evolve.

The work in this chapter proposes to use decision trees to construct training stage memory and latency prediction models that further the state-or-the-art set by [19] both in terms of prediction errors detailed in Table 2.4 as well as the ease of extending the developed performance model to new frameworks and devices. At the core of the proposed approach, analytical modelling is used to produce a set of features capturing memory footprint and computational cost of the various operations. These features along with profiled system attributes (Sec.4.1) are used to

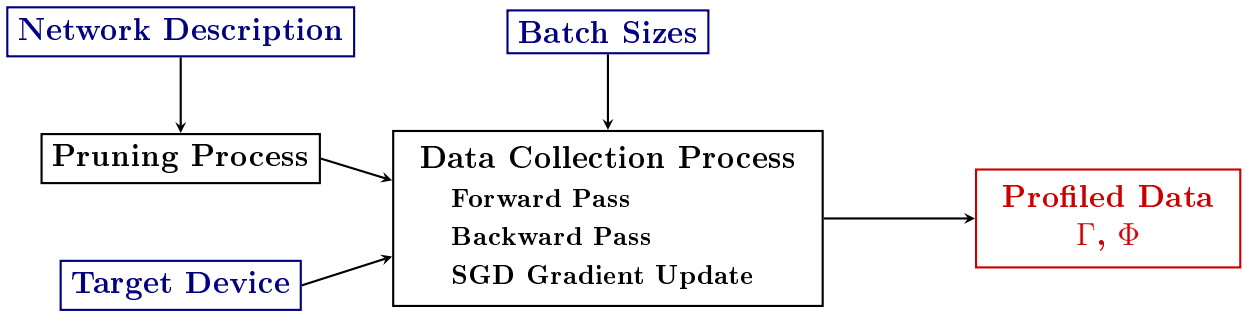


Figure 4.1: The proposed network-wise profiling strategy. Blue boxes are user provided inputs to the system, black boxes are processes and the red box is the output of the profiling system.

train decision tree based prediction models. This combination of handcrafting static features and profiling for device and framework specific optimisations provides a strong argument for future generalisation over [19].

4.2.1 Network-wise profiling strategy

This section outlines the proposed methodology for profiling the desired attributes and provides details on the targeted framework and device. In order to encode network architecture information into the decision tree training process, structured pruning (removing entire convolution filters) is utilised as a methodology to vary the topology of a network and hence produce the data points used to train the decision trees. Fig.4.1 shows the proposed profiling process. The *pruning process* takes as input the network description, the desired pruning level and strategy (S) and performs structured pruning.

The pruned network along with n_{bs} batch sizes in the range $[bs_{low}, bs_{high}]$ are passed to the *data collection process*. For each datapoint, this process profiles the Forward Pass, Backward Pass and SGD gradient update step on the target device. This profiling strategy limits the number of datapoints that can be collected as the degrees of freedom are the pruning levels, pruning strategies and batch size. However it benefits from the fact that the information captured does not treat each layer as an isolated element but rather as part of a larger architecture where the building blocks and their connectivity affects the performance.

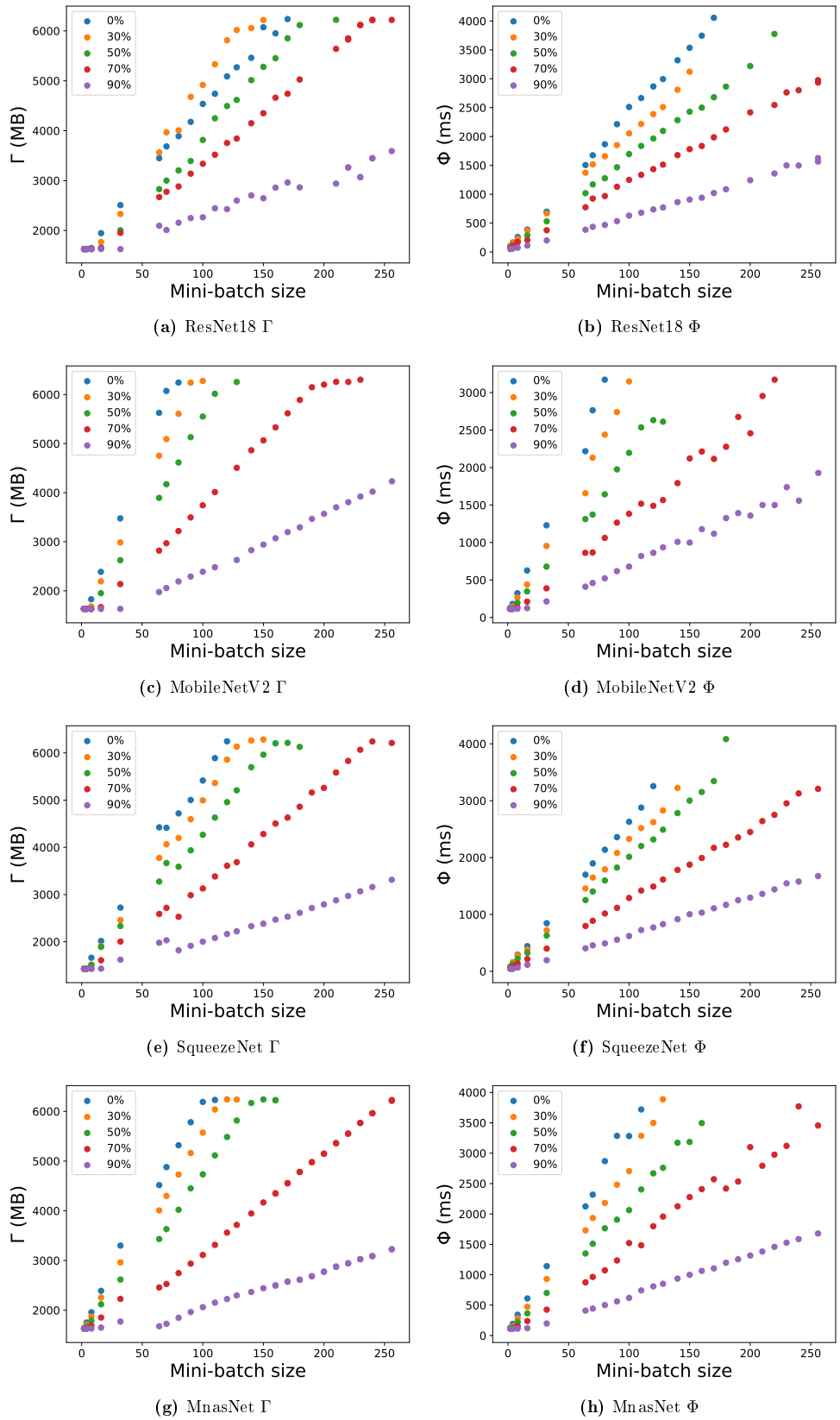
4.2.1.1 Profiling Hyperparameters

Sec.4.2.1 introduced the hyperparameters of S, n_{bs}, bs_{low} and bs_{high}, the values of which along with the targeted framework and device are detailed here. The training framework targeted is PyTorch v1.6 which uses CUDA 10.2 and cuDNN 8.0. The target device is the NVIDIA Jetson TX2. The filters to be pruned are randomly chosen (S), and the pruning is performed using the *Autoprune* [60] tool. $n_{bs} = 25$, $bs_{low} = 2$ and $bs_{high} = 256$; this range and granularity of batch sizes covers the most commonly used training batch sizes (powers of 2 up till 256) while providing sufficient information in the regions in-between. Framework and device specific details on profiling can be found in Appendix B.1.

4.2.2 Decision Tree Models

Four state of the art networks (ResNet18[28], MobileNetV2 [67], SqueezeNet[33] and MnasNet[78]) were profiled for the attributes defined in Sec.4.1 using the profiling strategy described in Sec.4.2.1 for pruning levels {0,30,50,70,90}%. As seen in Fig.4.2, for all networks, both attributes display linearity with batch size, but varying linear fit dependent on the network architecture (pruning level).

To model this behaviour, a decision tree based approach is adopted A decision tree selects terms that best partition the space into regions of low entropy. Regression predictions are made by classifying new data points into these regions and predicting the mean value of that region that was learnt in the training stage [59]. This data dependent partitioning of the space inherently fits the modelling problem at hand well. Hence, random forests [6] are employed to model both the memory and latency of training. To do so, the modelling algorithm needs to be provided with a list of features that define the axes of the space in which the data is fit. These features are constructed through analytical modelling of a network architecture and are described in the following section.



(a) ResNet18 Γ (b) ResNet18 Φ

(c) MobileNetV2 Γ (d) MobileNetV2 Φ

(e) SqueezeNet Γ (f) SqueezeNet Φ

(g) MnasNet Γ (h) MnasNet Φ

Figure 4.2: Training memory consumption (Γ) and minibatch latency (Φ) for 4 different networks that were profiled when training 3x224x224 sized inputs on the NVIDIA Jetson TX2. x-axis is mini-batch size and y-axis is the attribute value.

4.2.2.1 Analytical Modelling

The models receive as input a network description and training batch size. The modelling algorithm is provided with a list of terms, generated from this input, that define the axes of the space in which the data is fit. As discussed in Sec.2.1.3 cuDNN uses proprietary heuristics on a per layer basis to select between the Matrix Multiplication, FFT, and Winograd convolution algorithms. In order to model this black-box behaviour, features corresponding to the expected memory consumption and the operations performed for all three algorithms are generated from the network description, as layer-wise algorithm choice prediction before deployment is not possible. For each algorithm, the following computations are modelled.

$$y = x * w \qquad \text{[Forward Pass]}$$

$$\frac{\delta L}{\delta x} = \frac{\delta L}{\delta y} * rot180(w) \qquad \text{[Gradient w.r.t inputs]}$$

$$\frac{\delta L}{\delta w} = x * \frac{\delta L}{\delta y} \qquad \text{[Gradient w.r.t weights]}$$

where x is an IFM, y is an OFM and w is the weight matrix of a convolution layer; and L is the computed loss.

As discussed in [52], the convolution layers in CNNs take more than 98% of the execution time for most modern CNNs. Additionally, in terms of memory consumption, the only other types of layers which contain learnable parameters are Batch Normalisation layers. However, the memory consumption of these layers is less than 1% of that of the entire network. The inputs or outputs to other layers such as Batch Normalisation and Pooling need not be separately considered as in most networks, these feed into or are fed directly by convolution layers. Hence, the input of one is the output of the other. Non-linearities such as the ReLU [85] layer, perform computations inplace, i.e. on the input tensor directly without creating a copy. Consequently, when creating models for CNN execution and memory consumption, it is sufficient to consider the memory consumption and computations of just the convolution layers. The rest of this section describes the construction of analytically modelled features that are provided to the decision tree modelling methodology.

Consider a CNN where each convolution layer $l \in \mathcal{L}$ has n_l filters of size $m_l \times k_l \times k_l$. Let layer l have stride s_l, padding p_l and groups g_l. Let the IFM to this layer have dimensions $bs \times m_l \times ip_l \times ip_l$, the weights $n_l \times \frac{m_l}{g_l} \times k_l \times k_l$ and the OFM $bs \times n_l \times op_l \times op_l$ where bs is the batch size of training. The OFM spatial dimensions op_l can be calculated using the equation $op_l = 1 + \lfloor \frac{ip_l + 2p_l - k_l}{s_l} \rfloor$.

Independent of the choice of algorithm, the following features are allocated in memory.

$$mem_w = n_l \cdot \frac{m_l}{g_l} \cdot k_l^2 \qquad \text{[Weights]}$$

$$mem_{w_{grad}} = bs \cdot n_l \cdot \frac{m_l}{g_l} \cdot k_l^2 \qquad \text{[Gradient w.r.t weights]}$$

$$mem_{ifm_{grad}} = mem_{ifm} = bs \cdot m_l \cdot ip_l^2 \qquad \text{[Gradient w.r.t. inputs]}$$

$$mem_{ofm_{grad}} = mem_{ofm} = bs \cdot n_l \cdot op_l^2 \qquad \text{[Gradient w.r.t outputs]}$$

The remaining features described below all vary with the choice of algorithm.

Matrix Multiplication based convolution As discussed in Sec.2.1.3, a convolution is converted to a matrix multiplication by performing the *im2col* operation on the IFM (LHS of the convolution). There are two variants of the cuDNN operator, one which stores the entire *im2col* matrix in memory and one that only computes indices that allows the compute unit to read repeated values from the IFM without actually storing these duplicated values [14]. The proprietary nature of cuDNN means that which indices are calculated is not public. However the *im2col* operation reads repeated values due to the overlap between the $k_l \times k_l$ sliding windows during a convolution, where each window corresponds to 1 output pixel in op_l. Within a sliding window, the offsets of the k_l^2 values from the top-left element of the window ($win[0]$) are fixed regardless of which window (op_l) is being computed. However, s_l and p_l decide the pattern of $win[0]$. A reasonable assumption is that the indices computed to reduce the memory overhead of the *im2col* operation are the location of all $win[0]$ for each image in the IFM. The operations do not change between these two variants.

As an example, the forward pass (Eq.2.1) involves a multiplication between the *im2col* IFM

matrix of size $((bs \times op_l^2) \times (k_l^2 \times m_l))$ and a reshaped weights matrix of size $((k_l^2 \times m_l) \times (n_l))$. Using $k_l^2 \times m_l$ as the common dimension and calculating the memory and operations for a matrix multiplication gives the features $mem\_i2c^{mm}_{fwd^{total}}$ and $ops^{mm}_{bwd_x}$. As discussed, the indices of each of the op_l^2 $win[0]$ locations need to be stored, thus giving the $mem\_i2c^{mm}_{fwd^{index}}$ feature. Similar arguments applied to Eq.2.2 and 2.3 give the following parameter count and operations for a convolution performed as a matrix multiplication.

Variant: *im2col* performed with redundant data storage

$mem\_i2c^{mm}_{fwd^{total}} = bs \cdot op_l^2 \cdot k_l^2 \cdot m_l$ [Forward Pass]

$mem\_i2c^{mm}_{bwd_w^{total}} = bs \cdot op_l^2 \cdot k_l^2 \cdot \frac{m_l}{g_l}$ [Gradient w.r.t weights]

$mem\_i2c^{mm}_{bwd_x^{total}} = bs \cdot ip_l^2 \cdot k_l^2 \cdot m_l$ [Gradient w.r.t inputs]

Variant: *im2col* performed with index storage only

$mem\_i2c^{mm}_{fwd^{index}} = mem\_i2c^{mm}_{bwd_w^{index}} = bs \cdot op_l^2$ [Forward Pass] and [Gradient w.r.t weights]

$mem\_i2c^{mm}_{bwd_x^{index}} = bs \cdot ip_l^2$ [Gradient w.r.t inputs]

Common: Operations for convolution as matrix multiplication

$ops^{mm}_{fwd} = ops^{mm}_{bwd_w} = bs \cdot n_l \cdot op_l^2 \cdot k_l^2 \cdot \frac{m_l}{g_l}$ [Forward Pass] and [Gradient w.r.t weights]

$ops^{mm}_{bwd_x} = bs \cdot m_l \cdot ip_l^2 \cdot k_l^2 \cdot n_l$ [Gradient w.r.t inputs]

FFT based convolution As provided in [55], the features required to model a convolution performed as an FFT are:

$mem\_w^{fft}_{fwd} = n_l \cdot \frac{m_l}{g_l} \cdot ip_l \cdot (1 + ip_l)$ [Weights memory for Forward Pass]

$mem\_ifm^{fft}_{fwd} = mem\_ifm^{fft}_{bwd_w} = bs \cdot m_l \cdot ip_l \cdot (1 + ip_l)$ [IFM memory for Forward Pass and ... ...Gradient w.r.t weights]

$mem\_ofm^{fft}_{bwd_w} = bs \cdot n_l \cdot ip_l \cdot (1 + ip_l)$ [OFM memory for Gradient w.r.t weights]

$mem\_w^{fft}_{bwd_x} = n_l \cdot \frac{m_l}{g_l} \cdot op_l \cdot (1 + op_l)$ [Weights memory for Gradient w.r.t inputs]

$mem\_ofm^{fft}_{bwd_x} = bs \cdot n_l \cdot op_l \cdot (1 + op_l)$ [OFM memory for Gradient w.r.t inputs]

$ops^{fft}_{fwd} = ip_l^2 \cdot log(ip_l) \cdot (bs \cdot (m_l + n_l) + n_l \cdot \frac{m_l}{g_l}) + bs \cdot n_l \cdot m_l \cdot ip_l^2$ [Operations for Forward Pass]

$ops^{fft}_{bwd_x} = op_l^2 \cdot log(op_l) \cdot (bs \cdot (m_l + n_l) + n_l \cdot \frac{m_l}{g_l}) + bs \cdot n_l \cdot m_l \cdot op_l^2$ [Operations for Gradient w.r.t inputs]

$ops^{fft}_{bwd_w} = ip_l \cdot log(ip_l^2) \cdot (bs \cdot (m_l + n_l) + n_l \cdot \frac{m_l}{g_l}) + bs \cdot n_l \cdot m_l \cdot ip_l^2$ [Operations for Gradient w.r.t weights]

Winograd based convolution The Winograd minimal filtering algorithm [82] reduces the number of operations required to calculate q outputs using an r-tap FIR filter. This algorithm can be extended to 2-D and applied to a convolution instead of a 1-D FIR filter. As provided in [45], for the computation of $q \times q$ outputs with an $r \times r$ convolutional filter, the 2-D Winograd minimal filtering algorithm is

$$Y = A^T\Big[[GgG^T] \odot [B^T dB]\Big]A \tag{4.1}$$

In Eq.4.1, Y has shape $(q \times q)$, g is an $(r \times r)$ convolutional filter, and d is a tile of the input matrix of size $(q+r-1 \times q+r-1)$. A, G and B are transform matrices that only depend on the values of q and r and have sizes $(q \times q+r-1)$, $(q+r-1 \times r)$ and $(q+r-1 \times q+r-1)$ respectively. The algorithm takes $(q+r-1)^2$ multiplications to produce $q \times q$ output values.

In Eq.2.1, each channel of the IFM x is split into $\lceil \frac{ip_l}{q} \rceil^2$ tiles and each channel of the filter w is split into $\lceil \frac{k}{r} \rceil^2$ tiles. Each IFM and filter tile corresponds to d and g respectively in Eq.4.1. The memory consumed by A, G and B is fixed and does not scale with any of the layer parameters. The Hadamard product contains $3 \cdot (q+r-1)^2$ parameters (one for the LHS, RHS and result). The memory and operations scale with layer parameters depending on the number of these products that can be done in parallel. Each of the $\lceil \frac{ip_l}{q} \rceil^2$ tiles, n_l filters and bs images can be computed in parallel as they constitute independent operations. Accumulation needs to occur across $\lceil \frac{k}{r} \rceil^2$ filter tiles and m_l IFM channels and hence do not lend themselves to parallelisation.

As a further example, consider the case of Eq.2.2. $\frac{\delta L}{\delta y}$ is a matrix of size $(bs \times n_l \times op_l \times op_l)$ and w is a matrix of size $(n_l \times m_l \times k_l \times k_l)$. The term d in Eq.4.1, is one of the $\lceil \frac{op_l}{q} \rceil^2$ tiles of $\frac{\delta L}{\delta y}$ and g is one of the $\lceil \frac{k_l}{r} \rceil^2$ tiles of w. Assuming parallelism over bs, m_l and $\lceil \frac{op_l}{q} \rceil^2$ gives the feature $mem^{wino}_{bwd_x}$ and accounting for $n_l \cdot \lceil \frac{k_l}{r} \rceil^2$ accumulations gives the feature ops^{wino}_{fwd}.

By making similar arguments for Eq.2.3, the workspace memory and parameter counts for

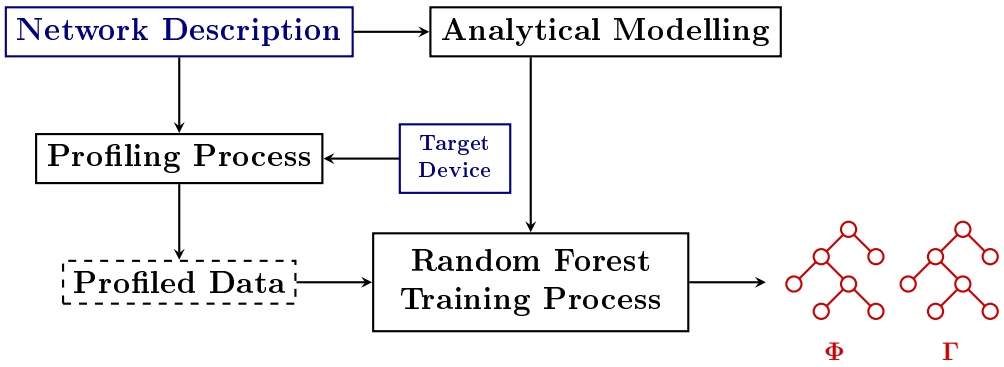


Figure 4.3: High-level description of the entire perf4sight toolflow. Blue boxes are user provided input, black boxes are processes, dashed boxes are data within the modelling system and red boxes are outputs from the system.

performing a convolution using the Winograd algorithm are given by:

$$mem_{fwd}^{wino} = bs \cdot n_l \cdot \lceil \frac{ip_l}{q} \rceil^2 \cdot 3 \cdot (q+r-1)^2 \quad \text{[Forward pass memory]}$$

$$mem_{bwd_x}^{wino} = bs \cdot m_l \cdot \lceil \frac{op_l}{q} \rceil^2 \cdot 3 \cdot (q+r-1)^2 \quad \text{[Gradient w.r.t inputs memory]}$$

$$mem_{bwd_w}^{wino} = bs \cdot n_l \cdot \frac{m_l}{g_l} \cdot \lceil \frac{ip_l}{q} \rceil^2 \cdot 3 \cdot (q+r-1)^2 \quad \text{[Gradient w.r.t weights memory]}$$

$$ops_{fwd}^{wino} = bs \cdot n_l \cdot \frac{m_l}{g_l} \cdot \lceil \frac{ip_l}{q} \rceil^2 \cdot \lceil \frac{k}{r} \rceil^2 \cdot (q+r-1)^2 \quad \text{[Forward pass operations]}$$

$$ops_{bwd_x}^{wino} = bs \cdot m_l \cdot n_l \cdot \lceil \frac{op_l}{q} \rceil^2 \cdot \lceil \frac{k}{r} \rceil^2 \cdot (q+r-1)^2 \quad \text{[Gradient w.r.t inputs operations]}$$

$$ops_{bwd_w}^{wino} = bs \cdot n_l \cdot \frac{m_l}{g_l} \cdot \frac{m_l}{g_l} \cdot \lceil \frac{ip_l}{q} \rceil^2 \cdot \lceil \frac{op_l}{r} \rceil^2 \cdot (q+r-1)^2 \quad \text{[Gradient w.r.t weights operations]}$$

4.2.3 Training Memory and Latency Models

For a given batch size, network architecture and device, the *mem* and *ops* features detailed above along with various summations of these features (shown below) are calculated per-layer and summed across all layers to obtain an estimate for the network. These set of 42 features are provided to the random forest trainer along with training data consisting of profiled Γ and Φ values for a range of batch-sizes and pruning levels (detailed in Sec.4.2.1.1). A separate random forest model is developed for each of the target attributes (Γ, Φ). This process is shown in Fig.4.3. The entire list of 42 features that are calculated per layer is as follows.

4.2.3.1 Tensor Allocations

The features in this section describe the sizes of tensors that constitute the IFM, weights and OFM; and each of their gradients on a per-layer basis.

1. $mem_w = n_l \cdot \frac{m_l}{g_l} \cdot k_l^2$
2. $mem_{w_{grad}} = bs \cdot n_l \cdot \frac{m_l}{g_l} \cdot k_l^2$
3. $mem_{ifm_{grad}} = mem_{ifm} = bs \cdot m_l \cdot ip_l^2$
4. $mem_{ofm_{grad}} = mem_{ofm} = bs \cdot n_l \cdot op_l^2$
5. $mem_w + mem_{w_{grad}} + mem_{ifm_{grad}} + mem_{ofm_{grad}}$

The following sections model the memory consumption and operations for the Matrix Multiplication, FFT and Winograd based convolution algorithms on a per layer basis. Each feature models one of Eq.2.1 (fwd), the forward pass; Eq.2.2 (bwd_x), the computation of the gradient w.r.t input; or Eq.2.3, the computation of the gradient w.r.t weights.

4.2.3.2 Matrix Multiplication based convolution

6. $mem\_i2c^{mm}_{fwd^{total}} = bs \cdot op_l^2 \cdot k_l^2 \cdot m_l$
7. $mem\_i2c^{mm}_{bwd_w^{total}} = bs \cdot op_l^2 \cdot k_l^2 \cdot \frac{m_l}{g_l}$
8. $mem\_i2c^{mm}_{fwd^{index}} = mem\_i2c^{mm}_{bwd_w^{index}} = bs \cdot op_l^2$
9. $mem\_i2c^{mm}_{bwd_x^{total}} = bs \cdot ip_l^2 \cdot k_l^2 \cdot m_l$
10. $mem\_i2c^{mm}_{bwd_x^{index}} = bs \cdot ip_l^2$
11. $mem\_i2c^{mm}_{fwd^{total}} + mem\_i2c^{mm}_{bwd_w^{total}} + mem\_i2c^{mm}_{bwd_x^{total}}$
12. $mem\_i2c^{mm}_{fwd^{index}} mem\_i2c^{mm}_{bwd_w^{index}} + mem\_i2c^{mm}_{bwd_x^{index}}$
13. $ops^{mm}_{fwd} = ops^{mm}_{bwd_w} = bs \cdot n_l \cdot op_l^2 \cdot k_l^2 \cdot \frac{m_l}{g_l}$

14. $ops^{mm}_{bwd_x} = bs \cdot m_l \cdot ip_l^2 \cdot k_l^2 \cdot n_l$

15. $ops^{mm}_{fwd} + ops^{mm}_{bwd_w} + ops^{mm}_{bwd_x}$

4.2.3.3 FFT based convolution

16. $mem\_w^{fft}_{fwd} = n_l \cdot \frac{m_l}{g_l} \cdot ip_l \cdot (1 + ip_l)$

17. $mem\_ifm^{fft}_{fwd} = mem\_ifm^{fft}_{bwd_w} = bs \cdot m_l \cdot ip_l \cdot (1 + ip_l)$

18. $mem\_ofm^{fft}_{bwd_w} = bs \cdot n_l \cdot ip_l \cdot (1 + ip_l)$

19. $mem\_w^{fft}_{bwd_x} = n_l \cdot \frac{m_l}{g_l} \cdot op_l \cdot (1 + op_l)$

20. $mem\_ofm^{fft}_{bwd_x} = bs \cdot n_l \cdot op_l \cdot (1 + op_l)$

21. $mem\_w^{fft}_{fwd} + mem\_ifm^{fft}_{fwd}$

22. $mem\_ofm^{fft}_{bwd_x} + mem\_ofm^{fft}_{bwd_w}$

23. $mem\_ofm^{fft}_{bwd_w} + mem\_ifm^{fft}_{fwd}$

24. Eq.21 + Eq.22 + Eq.23

25. $ops^{fft}_{fwd} = ip_l^2 \cdot log(ip_l) \cdot (bs \cdot (m_l + n_l) + n_l \cdot \frac{m_l}{g_l})$
$+ bs \cdot n_l \cdot m_l \cdot ip_l^2$

26. $ops^{fft}_{bwd_x} = op_l^2 \cdot log(op_l) \cdot (bs \cdot (m_l + n_l) + n_l \cdot \frac{m_l}{g_l})$
$+ bs \cdot n_l \cdot m_l \cdot op_l^2$

27. $ops^{fft}_{bwd_w} = ip_l \cdot log(ip_l^2) \cdot (bs \cdot (m_l + n_l) + n_l \cdot \frac{m_l}{g_l})$
$+ bs \cdot n_l \cdot m_l \cdot ip_l^2$

28. $ops^{fft}_{fwd} + ops^{fft}_{bwd_x} + ops^{fft}_{bwd_w}$

4.2.3.4 Winograd convolution

The following features are applied twice for $(q \times r)$ of (4×3) and (3×2) which both profiling and [35] showed to be the most commonly used sizes of Winograd convolutions by CuDNN·

29. $mem_{fwd}^{wino} = bs \cdot n_l \cdot \lceil \frac{ip_l}{q} \rceil^2 \cdot 3 \cdot (q + r - 1)^2$
30. $mem_{bwd_x}^{wino} = bs \cdot m_l \cdot \lceil \frac{op_l}{q} \rceil^2 \cdot 3 \cdot (q + r - 1)^2$
31. $mem_{bwd_w}^{wino} = bs \cdot n_l \cdot \frac{m_l}{g_l} \cdot \lceil \frac{ip_l}{q} \rceil^2 \cdot 3 \cdot (q + r - 1)^2$
32. $mem_{fwd}^{wino} + mem_{bwd_x}^{wino}$
33. $mem_{fwd}^{wino} + mem_{bwd_w}^{wino}$
34. $mem_{bwd_w}^{wino} + mem_{bwd_x}^{wino}$
35. Eq.32 + Eq.33 + Eq.34
36. $ops_{fwd}^{wino} = bs \cdot n_l \cdot \frac{m_l}{g_l} \cdot \lceil \frac{ip_l}{q} \rceil^2 \cdot \lceil \frac{k}{r} \rceil^2 \cdot (q + r - 1)^2$
37. $ops_{bwd_x}^{wino} = bs \cdot m_l \cdot n_l \cdot \lceil \frac{op_l}{q} \rceil^2 \cdot \lceil \frac{k}{r} \rceil^2 \cdot (q + r - 1)^2$
38. $ops_{bwd_w}^{wino} = bs \cdot n_l \cdot \frac{m_l}{g_l} \cdot \frac{m_l}{g_l} \cdot \lceil \frac{ip_l}{q} \rceil^2 \cdot \lceil \frac{op_l}{r} \rceil^2 \cdot (q + r - 1)^2$
39. $ops_{fwd}^{wino} + ops_{bwd_x}^{wino}$
40. $ops_{fwd}^{wino} + ops_{bwd_w}^{wino}$
41. $ops_{bwd_x}^{wino} + ops_{bwd_w}^{wino}$
42. Eq.39 + Eq.40 + Eq.41

4.3 Evaluation

This section describes the construction of the training and test sets for the evaluation and evaluates the performance models in Fig.4.3 on the following criteria. First the models are

evaluated on their ability to predict the two attributes when the training and test sets have the same base network, but different topologies of this network. Here, random structured pruning, i.e. randomly selecting convolution filters to remove, is used as the strategy to vary topology of a given base network. Thus, the training and test sets are then constructed by having different pruning levels in each, but both sets containing pruned versions of the same network. This criterion evaluates whether the decision tree models are capable of generalising between various network topologies generated from the same backbone CNN. The evaluation is presented in Sec.4.3.2.

The evaluation is then extended to allow the network itself to change, exploring the idea of training performance prediction models on data from a "basis" of networks and predicting attribute values for networks not in the basis. In this case, in addition to varying the pruning levels in the training and test sets, the two sets also vary in the backbone network utilised. Given that different backbone networks share similar building blocks (Table 3.1), this criterion evaluates whether the decision tree models can generalise building block specific information and thus predict Γ and Φ for networks not present in the training dataset, but built with the same building blocks. The evaluation is presented in Sec.4.3.3.

Sec.4.3.4 then presents a case study that utilises the developed models to perform on-device DaPR using OFA networks.

4.3.1 Constructing training and test sets

The degrees of freedom when generating configurations to profile are pruning levels, pruning strategies and batch-size. With batch-sizes and pruning strategy fixed to those specified in Sec.4.2.1.1, the training and test sets are developed by varying the pruning levels in each. Let the set of all pruning levels considered, in percentage of model memory pruned, be $\{5x|x \in [0, 18]\}$. This set of pruning levels needs to be split into two sets, one which represents the pruning levels in the training set and the other, pruning levels in the test set. To identify the partitioning of the set of the possible pruning levels, progressively larger training sets are

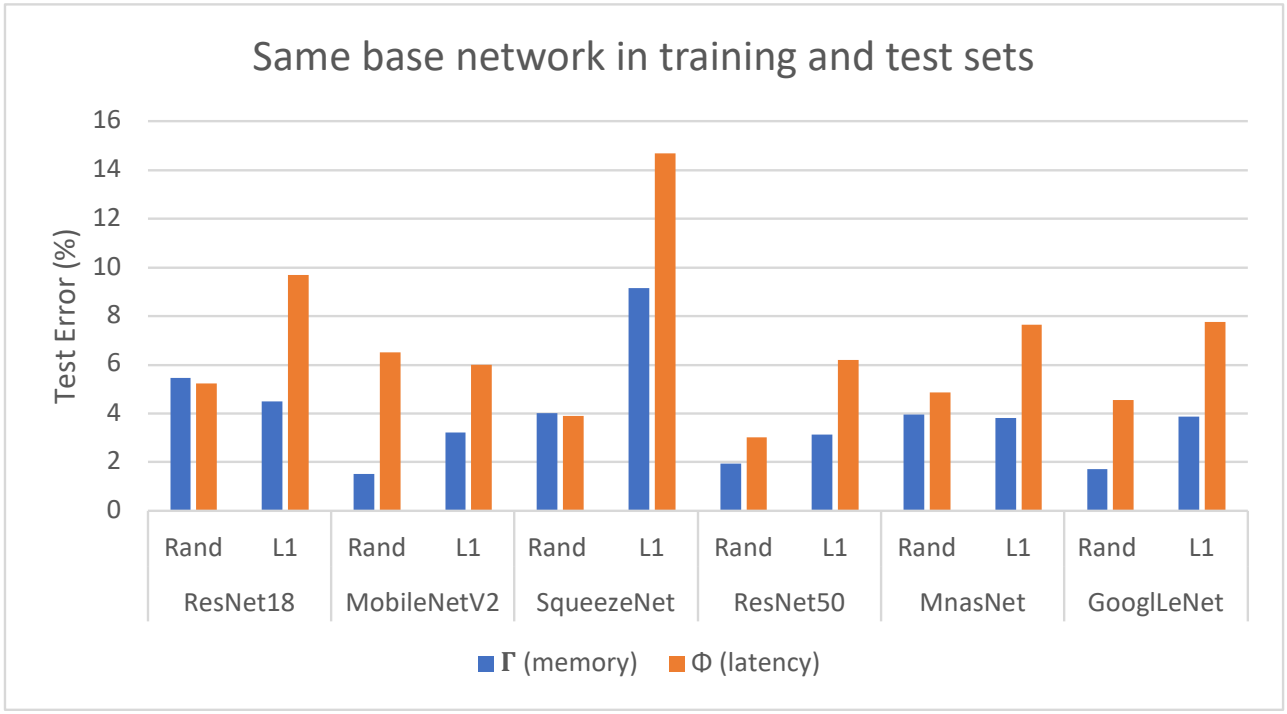


Figure 4.4: Mean test error of attribute prediction for random and L1-norm pruning strategies across 25 batch sizes in the range [2,256] and pruning levels $\{5x|x \in [0, 18], 5x \notin \{0, 30, 50, 70, 90\}\}$

created and for each, the Γ and Φ test set prediction accuracy is evaluated. The smallest training set after which the test set accuracy plateaus is selected as the training set to use in this evaluation.

AlexNet [44] is used to tune this hyperparameter of pruning levels present in the training and test sets. To avoid biasing results, AlexNet is not used in the rest of the evaluation as it is used for hyperparameter tuning. Sizes of training set (T) from 1 ($T = \{0\}$) to 8 ($T = \{0, 10, 20, 30, 50, 60, 70, 90\}$) in increments of 1 were used to train the two models and the resulting accuracy evaluated on a test set $\{5x|x \in [0, 18], 5x \notin T\}$. For $T = \{0\}$, the test error varied between 33%-74% across the attributes and decreased until $T = \{0, 30, 50, 70, 90\}$ after which it plateaued at 3%-6%. Expecting this trend to hold for other networks, the training set chosen for the following experiments is $T = \{0, 30, 50, 70, 90\}$, with the test set containing pruning levels $\{5x|x \in [0, 18], 5x \notin T\}$.

4.3.2 Generalisation within topologies of the same base network

All results presented in this section correspond to the scenario where both the training and test sets are constructed using data from the same base network but differ in topology (pruning levels and strategy). Random pruning strategy refers to randomly pruning filters with equal probability across all layers. In the context of the motivating example for this chapter, this section evaluates the possibility of training a decision tree model based on some random sample of sub-networks from an OFA super-network and using it to predict Γ and Φ for new sub-networks sampled from the same OFA super-network.

Bars labelled *Rand* in Fig.4.4 show the mean attribute prediction error averaged across a wide range of pruning levels and batch-sizes when a random pruning strategy was employed for both training and test sets. Bars labelled *L1* in Fig.4.4 show the mean attribute prediction error when a random pruning strategy was employed for the training set, but an L1-norm pruning strategy for the test set. This strategy prunes filters with the smallest L1-norm first and results in more filters pruned from deeper layers. The results show that the mean prediction error of Γ and Φ does not exceed 9.15% and 14.7% respectively. Furthermore, compared to testing on randomly pruned networks, Γ and Φ demonstrate on average across networks, only 1.51pp and 4.0pp increases in error respectively when testing on L1-norm pruned networks.

To further explore the ability of the modelling to encode network topology information, 100 50% pruned MobileNetV2 topologies were generated by random structured pruning. Instead of only pruning uniformly across all layers as before, additional topology variation was induced by randomly increasing pruning at early, late or middle layers. For batch-size 80, the mean and variation in attribute values Γ and Φ across topologies were 4423$\pm$1597 MB and 1741$\pm$871 ms respectively. The models, trained on just a uniform random pruning strategy, predicted these attributes with a mean error across topologies of 1.32% and 9.90% respectively. Compared to testing on data obtained using a uniform random pruning strategy for MobileNetV2 (Fig.4.4), this scenario demonstrates only a 0.2pp and 3.4pp increase in error rate in Γ and Φ respectively. Thus, the results from this section demonstrate the ability of the proposed methodology to capture network architecture dependent information well.

4.3.2.1 Comparison to state-of-the-art works modelling CNN training memory

DNNMem [19] predicts the memory consumption of training for an NVIDIA Tesla P40 GPU using PyTorch 1.2, CUDA 9.0 and CuDNN 7.0.2. Due to lack of access to a P40 or DNNMem's source code, but to perform as fair a comparison as possible, ResNet50 was profiled on an NVIDIA RTX 2080Ti server GPU using PyTorch 1.6, CUDA 10.2 and CuDNN 7.6 as the framework configurations used in [19] do not support the 2080Ti. Γ (corresponding to DNNMem's memory consumption attribute) had a 2.4% prediction error across batch-sizes and topologies (pruning levels). This significantly outperforms [19] which had a 17.4% error across a range of batch-sizes and input image sizes. Possible reasons for the worse error achieved by [19] are inaccuracies in handcrafting features per training framework or the fact that [19]'s model accounts for a multiple GPU setting which may have lead to inaccuracies on the single GPU setting evaluated here. By demonstrating perf4sight's ability to generalise from a unified memory embedded GPU system to a non-unified memory single GPU server system, these results indicate potential for future generalisation to other devices and training frameworks.

4.3.3 Generalisation across different networks with similar building blocks

This section investigates the performance of the proposed framework in the case where the targeted network is not known beforehand but rather a model is built using data from a "basis" of networks, and predictions are made for the unseen network. The training set was constructed with a combination of data from ResNet18, MobileNetV2 and SqueezeNet (basis) for the same batch-sizes and pruning levels listed in Sec.4.3.1 and a uniform random pruning strategy. This training set represents the "basis" of networks. Testing was performed on data from both random and L1-norm pruned networks for ResNet18, MobileNetV2, SqueezeNet, ResNet50, MnasNet, GoogLeNet. Test set attribute prediction error for each network was obtained individually based on a model trained on aggregated data from the basis networks.

As only a single basis is created, the results for ResNet18, MobileNetV2 and SqueezeNet cor-

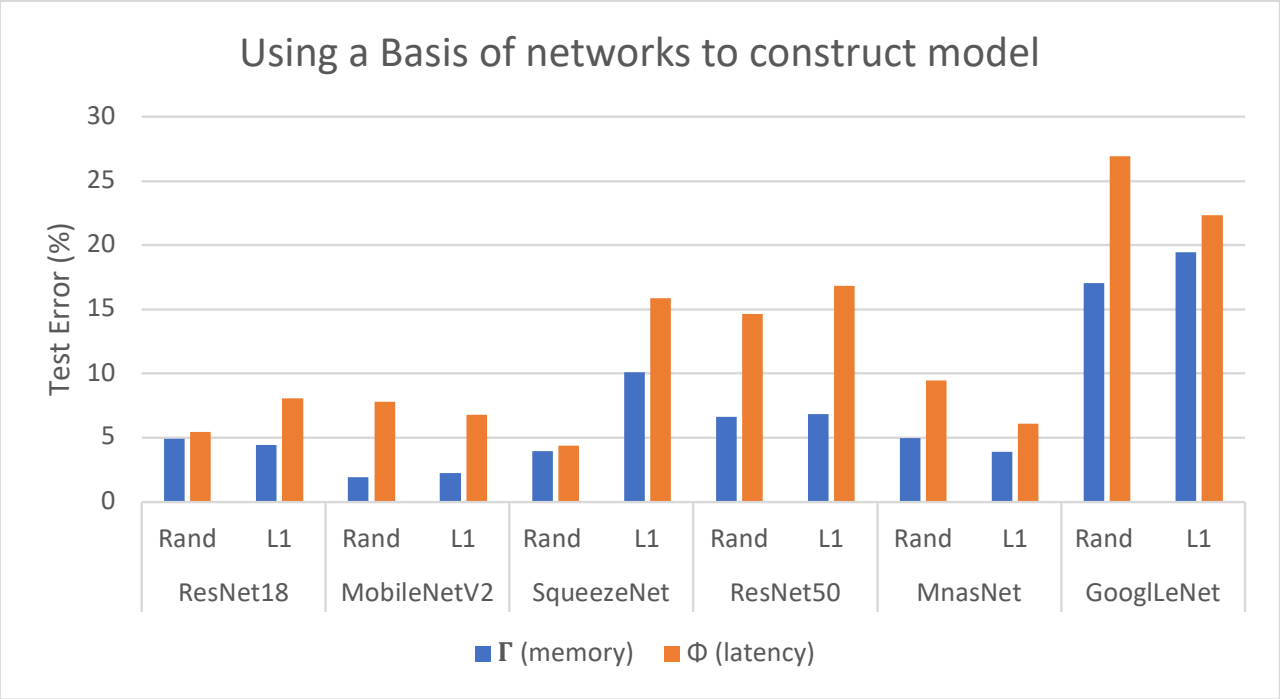


Figure 4.5: Mean test error of attribute prediction across 25 batch sizes in the range [2,256] and pruning levels $\{5x|x \in [0,18]\}$. The random forest models used here are trained using data from a basis of networks containing ResNet18, MobileNetV2, and SqueezeNet.

respond to testing on networks present within the basis. However unlike Sec.4.3.2, where a separate model is created per network, here only a single basis model is trained for testing on all networks. The results for ResNet50, MnasNet and GoogLeNet, correspond to testing on networks that contain similar building blocks to those in the basis.

The attribute prediction errors are shown in Fig.4.5. Across all attributes and pruning strategies, networks present in the basis (ResNet18, MobileNetV2 and SqueezeNet) showed -1pp[2], +4.6pp, +2.7pp mean increase in error respectively and networks not present in the basis (ResNet50, MnasNet and GoogLeNet [76]) showed +5.6pp, +2.55, +16pp mean increase in error respectively compared to the results in Fig.4.4. Across the networks in Fig.4.5, the mean errors for Γ and Φ are 5.53% and 9.37% respectively.

Table 4.1 details various commonly used building blocks and the networks that utilise them. Both the Fire and Inception modules employ a "branch-and-concatenate" computation structure and have 1×1 and 3×3 convolutions. The Fire module has 2 branches, whereas the Inception module has 4 branches and a 5×5 convolution. ResNet18 is made of "Basic-block

[2]pp stands for percentage points

Building Block	Networks
Residual	ResNet18, ResNet50
Depth-wise Separable	MobileNetV2, MnasNet
Fire	SqueezeNet
Inception	GoogLeNet

Table 4.1: Summary of architectural building blocks and networks that utilise them

residuals" which have two 3×3 convolutions, whereas the deeper ResNet50 is made of "Bottleneck residuals" which have 3 convolutions (one 3×3 and two 1×1). MobileNetV2 and MnasNet are made up of the same depth-wise separable inverted residual module, but have different depths. Thus architecture pairs from most to least similar are: MobileNetV2 and MnasNet; ResNet18 and ResNet50; and SqueezeNet and GoogLeNet.

The results of this investigation are inline with these observations where the model trained on a basis of networks predicted MnasNet attributes the best followed by ResNet50 and then GoogLeNet. This supports the idea of using a basis of networks, but suggests that the best performance is achieved when the exact building blocks if not the identical network is present in both training and test sets. This is an expected trend, but the evaluation quantifies the error in attribute prediction for each case.

4.3.4 On-device DaPR using OFA

As discussed earlier, the motivating goal for the performance modelling work in this chapter, was to enable utilisation of OFA as a methodology for on-device topology adaptation. The OFA network [9] is a large super network trained once offline, following which, a large number of smaller sub-networks can be sampled, each with good performance on the original training dataset. [9] proposed the network as an offline methodology to enable rapid search of the space of network topologies in order to tailor a network to a desired target device. This case study explores the option of using the OFA network on an edge device as a way to perform on-device DaPR in order to accommodate a dynamic system where both the observed data distribution and available memory and latency budgets can vary over time. The OFA network used is

OFAResNet50 trained and open-sourced by [9] and has the same building blocks as ResNet50, but a slightly different connectivity.

Dynamic system description Let us consider the scenario of an autonomous car performing image classification on an NVIDIA Jetson TX2. As examples of input data for classification, let us consider 4 subsets of the ILSVRC'12 dataset containing images that could be observed by an autonomous car in *City*, *Motorway*, *Countryside* and *Off-road* settings. The subsets were created to emulate objects, people, buildings and animals that might be observed in different environments that an autonomous vehicle could encounter. The details of the classes present in each subset are as follows [3]:

- **City** (185 classes) : ambulance, trash can, station wagon, tandem bike, taxi, car mirror, car wheel, convertible, crane, electric locomotive, fire engine, fire truck, garbage truck, petrol pump, radiator grille, jeep landrover, rickshaw, lawn mower, limousine, mailbox, manhole cover, minibus, minivan, Model T, moped, scooter, mountain bike, moving van, parking meter, passenger car / coach, pay-phone, pickup truck, police car, race car, school bus, shopping cart, snowplow, sports car, steel arch bridge, tram, suspension bridge, tow truck, trolley bus, street sign, traffic light, palace, mosque, church building, castle, boathouse, tirumphal arch, academic gown/robe, cardigan, fur coat, gown, jersey / t-shirt, suit, sunglasses, sweatshirt, trench coat, umbrella, swan, dogs, red fox, cats

- **Motorway** (26 classes) : station wagon, bullet train, car mirror, car wheel, convertible, electric locomotive, petrol pump, jeep landrover, minibus, minivan, mobile home, Model T, moving van, passenger car / coach, pay-phone, pickup truck, police car, race car, recreational vehicle, snowplow, sports car, tow truck, trailer truck, street sign, water tower

- **Countryside** (204 classes) : wheelbarrow, station wagon, bullet train, taxi, car mirror, car wheel, convertible, electric locomotive, freight car, garbage truck, petrol pump, radi-

[3] There are fewer written categories than the number of classes stated as some categories such as "dogs" have many ILSVRC'12 classes within them. There are also overlap of classes between the subsets.

ator grille, half track, horse cart, jeep landrover, lawn mower, mailbox, manhole cover, minibus, minivan, mobile home, Model T, moped, scooter, mountain bike, moving van, oxcart, parking meter, passenger car / coach, pay-phone, picket-fence, pickup truck, plough, police car, race car, recreational vehicle, school bus, snowmobile, snowplow, sports car, steel arch bridge, tank, thatched roof, tile roof, tow truck, tractor, trailer truck, worm fence, street sign, traffic light, hay, palace, mosque, church building, castle, lighthouse, barn, viaduct, water tower, cardigan, fur coat, gown, sarong, jersey / t-shirt, suit, sunglasses, sweatshirt, swimming trunks, trench coat, umbrella, cock, hen, quail, goose, swan, dogs, red fox, cats, rabbits, ram, sheep

- **Off-road** (26 classes) : mobile home, mountain bike, oxcart, pickup truck, plough, snowmobile, tank, tractor, hay, ostrich, iguana, alligator, wallaby, koala, wombat, brown bear, black bear, hog, wild boar, ox, water buffalo, bison, wild deer

Fig.4.6 shows some examples of images in each subset.

Let us consider the case where the observed data distribution changes over time from a city to the countryside and the memory consumption of other, possibly safety critical, processes running on the device also change over time. Assume there is some available time during which the system can retrain a deployed network to account for the shift in the observed data distribution. Let us place hard constraints on the following three attributes: acceptable memory consumption of training (Γ) and inference (δ), and latency (ϕ) of inference, all of which can change over time. These are determined by the memory consumption of other processes running on the device at the time of retraining and inference as well as the required latency of the classification task.

Note that we also evaluate the runtime memory consumption of inference here which is not a common metric to evaluate on as works which optimise for inference aim to improve the latency of inference while assuming sufficient memory to perform inference. However, in the scenario of execution on edge devices, this is a metric of interest as such devices typically have significant memory restrictions and hence the memory of both the inference and training stages

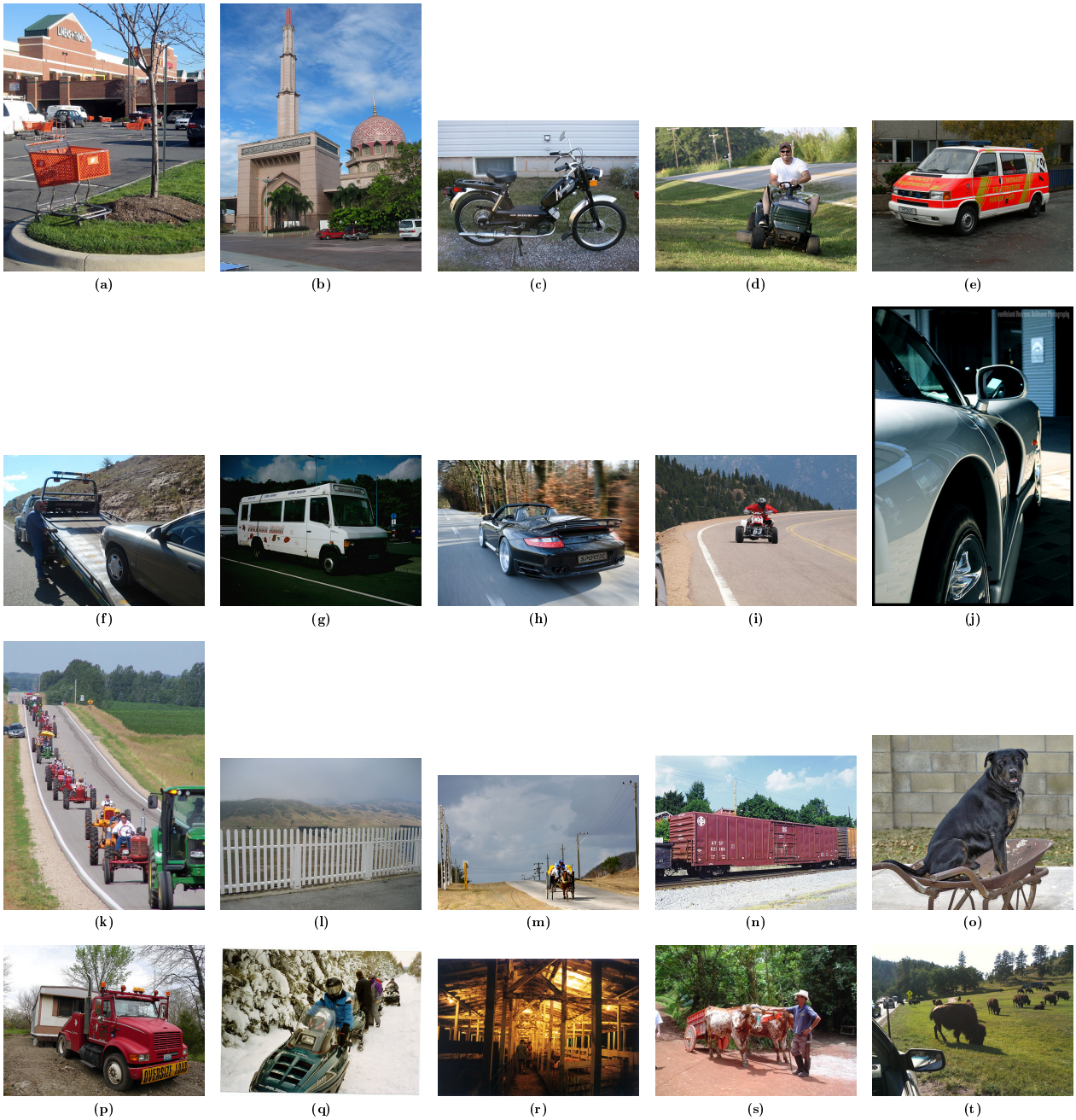


Figure 4.6: Examples of images in each of the 4 subsets of ILSVRC'12. Figs.(a-e) show examples of images in the City subset. Figs.(f-j) show examples of images in the Motorway subset. Figs.(k-o) show examples of images in the Countryside subset. Figs.(p-t) show examples of images in the Off-road subset.

is important to consider.

Given the above scenario, the goal of this case study is to design a system that can safely and quickly identify network architectures on which training and inference can be performed without exceeding the above constraints. The proposed system uses the evolutionary search (ES) algorithm proposed in [9] to search the OFA super network for a sub-network that meets the requirements. The ES algorithm starts with a population of 100 sub-networks and runs 500 iterations of evolutionary search before identifying the best performing sub-network within the constraints. This results in the process sampling at least $50,000$ sub-networks, but often more due to some sub-networks exceeding the hard constraints. Each sub-network sampled requires an estimation of the three constrained attributes. The following analysis compares both the approach of profiling each sub-network on device and utilising the developed memory models.

Naive Approach (Profiling) It is not feasible to profile offline the attribute values for all possible sub-networks as the weight sharing employed by OFA results in too many possible sub-networks ($> 50,000$) to do so. Furthermore, the mean and standard deviation of Γ for a 100 sub-networks sampled from OFAResNet50 profiled on the TX2 was 3332 ± 407 MB for batch size 32 and 5389 ± 527 MB for batch size 64. This large variation means that a single estimate for all possible sub-networks sampled would be a highly uncertain estimate. Hence, this necessitates case-by-case sampling on-device. In the case where there are other safety critical applications to be executed, profiling is not feasible due to the possibility of a profiled sub-network consuming more memory than is acceptable and preventing the start up of another application. Additionally, reliably profiling requires use of the GPU and multiple runs to average across. On the TX2 this process takes on average 20s per data point. With the $50,000$ sub-networks sampled by the ES algorithm, this would result in an infeasible 11 days of run-time.

Utilising performance models Alternatively, utilising the memory prediction model to predict the attribute values only requires 0.1s and 2MB as it simply requires the inference of a random forest model. For the same search space as above, this would result in 1.4 hours of run-

time with the entire process running only on the CPU and hence can even be performed when the GPU is being utilised for network inference. The model for Γ developed in Sec.4.3.2 using data from ResNet50 generalises well to the OFA version and predicts Γ for the 100 sampled sub-networks with a mean error of only 4.28%.

Furthermore, with minor modifications to the approach proposed in Sec.4.2, random forest models were developed to predict both runtime inference stage attributes δ (memory consumption) and ϕ (latency). The proposed methodology is split into two stages, profiling and analytical modelling. The profiling stage was modified for the case of inference as follows. Batched (throughput driven) is less common than single image (latency driven) inference on the edge device setting as there are diminishing throughput gains after batch sizes greater than 32 on the Jetson TX2 [38]. Thus profiling for the batch sizes specified in Sec.4.2.1.1 yields little useful information. However, limiting the batch sizes profiled to between 1 and 32 and utilising only the five pruning levels specified in Sec.4.3.1 results in too few data points to train decision tree models on. Instead, the models developed for δ and ϕ are trained using profiled data for 25 out of the 100 sampled OFA sub-networks for batch sizes 1,2,4,8,16,32 and the reported test errors are for the remaining 75 sampled networks averaged across all batch sizes. Furthermore, the profiling was performed only for the inference stage of execution instead of the whole training stage.

The analytical modelling stage was then modified to only use the features corresponding to the forward pass from Sec.4.2.3.1 to train the random forest models. As the training and test sets consisted of 25 and 75 sampled OFA sub-networks respectively, and for each 6 batch sizes were profiled, there were a total of 150 training data points and 450 test data points. For the test data points, the mean and standard deviation of δ and ϕ were 1811 ± 248MB and 164 ± 165ms. For this test dataset, the developed models predict δ and ϕ with 1.8% and 4.4% errors respectively.

Utilising these performance models to accurately predict Γ (training stage memory consumption), δ (inference stage memory consumption), and ϕ (inference stage latency), allowed for the proposed OFA based methodology for on-device model adaptation and retraining to be

explored. The remainder of this section describes the results of this experiment.

Performance gains Table 4.2 displays the results of retraining 4 different sub-networks on the 4 subsets described above. The sub-networks MAX and MIN correspond to the largest and smallest sub-networks, in terms of static memory consumption (γ), that can be extracted from OFAResNet50 and hence no search was required to obtain them. Sub-networks A and B are obtained using the ES algorithm with progressively stricter constraints on Γ, δ and ϕ. The table details all 6 training and inference stage attributes for for all 4 of these networks. The MAX sub-network is used as a comparison as this is assumed to be the best possible model that can be deployed when maximising for accuracy without retraining. Consequently, all attribute values have a comparison in terms of "-x times less" than the MAX network. Networks MAX and MIN do not have any values for model search time (Θ) as these networks can be identified pre-deployment without any evolutionary-search being performed. The search times provided for models A and B that were found through evolutionary search are provided as (*profiling approach*)/(*performance model based approach*).

The achieved classification accuracy ($\bar{\alpha}$) is displayed per subset. The row corresponding to *Initial* describes the accuracy of the model without any subset specific retraining. The row corresponding to *Retrained* describes the accuracy of the model after retraining on the observed subset for 1 epoch. Note that the model MAX is not retrained on the subset as this is the initial deployed model that is assumed to have a sufficient classification accuracy for the application at hand. Consequently, the percentage point differences in accuracy are all relative to the *Initial* accuracy of MAX. This difference evaluates if a smaller model upon retraining can achieve the same classification accuracy of the original deployed model. In context of the problem setting introduced in Sec.1.2, this evaluates the constraint of $\bar{\alpha}(\mathcal{M}_{\mathcal{D}'}) \geq \bar{\alpha}(\mathcal{M}_{\mathcal{D}})$.

The results show that not performing evolutionary search (MIN) produces models that consistently under perform compared to MAX in terms of classification accuracy on the subset after retraining. On average across subsets of ILSVRC'12, the model MIN after retraining performs 1.3pp worse than MAX. Consequently, there is a need to perform evolutionary search as it is not

			MAX	A	B	MIN
$\boldsymbol{\Theta}$ (hours)			-	382/1.9	519/2.6	-
$\boldsymbol{\rho}$ (GFLOPs)			7.5 (1x)	3.6 (2.1x)	1.9 (4.0x)	0.96 (7.8x)
$\boldsymbol{\gamma}$ (MB)			192 (1x)	153 (1.3x)	76 (2.5x)	26 (7.4x)
$\boldsymbol{\Gamma}$ / $\boldsymbol{\Lambda}$ (MB)			5838 (1x)	3735 (1.6x)	3104 (1.9x)	2768 (2.1x)
$\boldsymbol{\delta}$ (MB)			1958 (1x)	1873 (1.05x)	1733 (1.1x)	1569 (1.2x)
$\boldsymbol{\phi}$ (ms)			69.6 (1x)	39.2 (1.8x)	25.1 (2.8x)	19.1 (3.6x)
$\bar{\boldsymbol{\alpha}}$ (%)	City	Initial	82.0	81.4	79.6	76.4
		Retrained	-	82.8 (+0.8pp)	81.6 (-0.4pp)	78.9 (-3.1pp)
	Off-road	Initial	86.2	83.9	84.1	79.6
		Retrained	-	90.4 (+4.2pp)	89.9 (+3.7pp)	88.1 (+1.9pp)
	Motorway	Initial	78.3	78.0	76.4	70.8
		Retrained	-	81.0 (+2.7pp)	80.2 (+1.9pp)	77.3 (-1.0pp)
	Countryside	Initial	82.4	81.7	80.0	77.0
		Retrained	-	83.6 (+1.2pp)	81.9 (-0.5pp)	79.4 (-3.0pp)

Table 4.2: Performance gains from on-device model selection and retraining. Top1 test accuracy ($\bar{\alpha}$), model search time (Θ), model size (γ), adaptation memory consumption (Λ) which is equivalent to the training memory conusmption (Γ), memory consumption of inference (δ) and latency of inference (ϕ) for 4 subnetworks (MAX, A, B, MIN) of OFAResNet50 on the 4 autonomous driving subsets (City, Off-road, Motorway, Countryside). Γ is reported for batch size 32 and γ and ϕ are reported for batch size 1.

possible to simply select the smallest model (in terms of γ) that can be sampled from the OFA super-net. Furthermore, the networks found through evolutionary search can vary significantly in attribute values from MAX. For instance network B has 2.8x less inference time latency than MAX. This variation in attributes further reinforces the need to utilise performance models. Performing evolutionary search (A and B) results in γ (static memory consumption), Γ (training stage memory consumption), δ (inference stage memory consumption) and ϕ (inference stage latency) improvements up to 2.5x, 1.9x, 1.1x and 2.8x respectively. Furthermore, in most cases, models A and B achieve a better Top1 test accuracy, after retraining, compared to MAX. On average across subsets, the models A and B perform 2.2pp and 1.2pp respectively better than MAX after retraining. Furthermore, performing evolutionary search using the naive approach of profiling results in infeasible search times (Θ) for on-device deployment while utilising the models provides a 200x improvement in search time allowing the process of model selection and retraining to be performed on-device.

4.3.5 Model Adaptation Metrics of Interest

Table 4.2 summarises most of the metrics of interest introduced in Table 1.1. The adaptation process latency (Θ) for this methodology is the time taken to perform evolutionary search on the OFA super network as this step is by a large extent the most time consuming part of the adaptation. Retraining is negligible as we only retrain for 1 epoch due to the high performing networks that can be sampled from the OFA super network. However, the retraining memory consumption Γ is important and corresponds to the adaptation process memory consumption Λ. The evolutionary search using decision trees has a negligible memory cost. The static memory consumption (Δ) corresponds to γ of the OFA super network which is 192 MB.

4.4 Conclusion

This chapter introduced a methodology to model the memory consumption and latency of the training process for a given combination of network, device and machine learning framework. The developed decision tree based models predict memory consumption and latency of the training process with a mean error of 4.1% and 9.4% respectively over a wide range of network topologies and mini-batch sizes for the Jetson TX2 device and PyTorch framework. These error rates are a significant improvement over those obtained by other works when modelling the same attributes on CNN training. Thus, perf4sight enables the quick and accurate identification of favourable training configurations on edge devices. The methodology has been automated and open-sourced to enable researchers working on topics from NAS to resource constrained training of CNNs to benefit from the ability to accurately predict hardware attributes of CNN training.

Furthermore, a use case of the performance models as a way to enable on-device DaPR is also proposed by utilising an OFA network that is deployed onto an edge device. The OFA super-network exposes a large search space of more than 50,000 topologies. In a system where the memory and latency budgets of both inference and training can change over time, there is a requirement to identify network topologies that fit within these budgets on-device. Profiling each of these topologies is not only infeasibly time consuming, but also can be unsafe in

safety critical scenarios where doing so might prevent other safety critical processes from starting up due to limited shared memory available. In such cases, utilising the proposed models enables rapid and accurate prediction of these execution attributes such that search time for the topologies can be accelerated by up to 200x compared to on-device profiling. By doing so, improvements in model size and inference stage latency of up to 1.9x and 2.8x respectively could be achieved on various subsets of the ILSVRC'12 dataset. The pruned networks (found through evolutionary search) performed up to 3.7pp better than the original unpruned network after retraining on the observed data distribution.

However, even though compared to profiling, model search times were improved by up to 200x, as demonstrated in Table 4.2, the search times are still around 2-3 hours. Consequently, the proposed methodology falls under the scenario where the device collects data about the observed environment over time and performs this search and adaptation process when there is available downtime. In the case of an autonomous vehicle, this could be when parked at work or overnight at home.

The methodology proposed in Sec.4.3.4 addresses the first of the two constraints discussed in Sec.3.4 by demonstrating the ability to perform DaPR within reasonable time constraints on an edge device when utilising state-of-the-art networks and datasets with large input image sizes. However, the results presented here still do not account for the uncertainty in image labels that would be present in a real-world scenario when attempting to retrain a network using data observed after deployment. The following chapter proposes a methodology that can also account for this uncertainty, thus addressing both constraints presented in Sec.3.4.

Chapter 5

LoCO-PDA : Low-Cost On-Device Partial Domain Adaptation

Chapter 4 proposed a methodology to perform on-device topology adaptation and retraining by utilising a deployed OFA super-network. The proposed methodology performs a search on-device over more than 50,000 sub-networks of the OFA super-network in order to identify a pruned architecture that fits within target budgets placed on the memory and latency of inference and training. However, profiling each sub-network on an edge device to identify if it fits within the target budgets takes in the order of days when searching more than 50,000 sub-networks. In order to enable this methodology to be performed within a reasonable time budget on an edge device, the chapter proposed a novel decision tree based modelling methodology to predict the inference and training stage attributes of memory consumption and latency. The developed models provide predictions for these inference and training stage attributes 200 times faster than profiling, thus bringing search times down to 2-3 hours. The architecture identified by this search is then retrained on the observed data distribution in order to improve its classification accuracy.

This final stage of the solution assumes availability of ground-truth labels of the observed data upon deployment for retraining and furthermore, the solution as a whole does not aim to reduce the cost of retraining a CNN itself. Consequently, this chapter focuses on reducing the cost of

retraining a CNN on an edge device while accounting for the uncertainty in the image class labels that are obtained upon deployment. By doing so, it addresses both the concerns about practical applicability of on-device DaPR methodologies introduced in Sec.3.4.

As this chapter does not assume availability of ground-truth labels upon deployment, unlike previous chapters the proposed methodology here is unsupervised. Furthermore, as introduced in Sec.1.2, the label space of the observed data distribution is assumed to be a subset of that of the original training distribution. These two features of the solution proposed in this chapter bring it closer to the field of partial domain adaptation (PDA) as introduced in Sec.2.5.1. However an important distinction here from works in the field of PDA is that the work here does not assume a difference between the training and observed data distributions in the distribution of the data points within a class but rather only in the label spaces. Nonetheless, the work in this chapter is also compared against PDA methodologies as they provide a good comparison point in-terms of the achieved post-adaptation classification accuracy in a closely related domain of research.

Consider the following deployment scenario. The inputs to the system are a model $\mathcal{M}^0$ and a large training data distribution $\mathcal{D}$. Let $\mathcal{M}^0$ be the model that has been trained on $\mathcal{D}$ and prepared for initial deployment on an edge device. Upon deployment, the system observes a data distribution $\mathcal{D}'$ that is assumed to be able to change over time. Furthermore, the label space of $\mathcal{D}'$ is assumed to be a subset of that of $\mathcal{D}$. Due to this assumption, it is expected that a model ($\mathcal{M}^p$), with a significantly smaller compute and memory budget than $\mathcal{M}^0$, can be adapted to the observed $\mathcal{D}'$ after deployment as a smaller number of classes are being classified ($\mathcal{D}' \subset \mathcal{D}$). This results in an adapted network that also has significant gains in inference memory and latency.

The goal of this chapter is to create a system that enables the training of $\mathcal{M}^p$ on an edge device such that the accuracy of $\mathcal{M}^p$ on $\mathcal{D}'$ is at least as good as that of $\mathcal{M}^0$ on $\mathcal{D}'$. The main difference between the problem setting here and in previous chapters is that here we do not assume availability of ground-truth labels for images in $\mathcal{D}'$ when performing this training. We will focus on obtaining $\mathcal{M}^p$ from $\mathcal{M}^0$ by pruning away $p\%$ of the memory footprint of

$\mathcal{M}^0$ using the Taylor-First order (TFO) and OFA pruning methodologies discussed in Sec.2.3. As discussed in Chapter 4, the TFO methodology is too memory intensive for pruning to be performed directly on an edge device and even with the profiling methodology proposed earlier, performing OFA on the edge takes around 2-3 hours. Consequently, the work in this chapter performs all pruning offline before deployment. Here, $\mathcal{M}^0$ and $\mathcal{M}^p$ correspond to $\mathcal{M}_{\mathcal{D}}$ and $\mathcal{M}_{\mathcal{D}'}$ respectively as introduced in Sec.1.2. However, in order to indicate the pruning levels of the models in a succinct manner, this new more concise notation is utilised for the rest of this chapter. The novel contributions of this chapter are as follows:

- LoCO-PDA: A methodology that utilises Variational Autoencoders (VAEs) to perform the low-cost retraining of networks to the observed data distribution on an edge device.
- An open-source tool [1] that trains and deploys the VAEs required to perform the on-device network adaptation.
- Identification of the fact that modern networks with linear classifiers can be adapted to changes in the label space of domains by solely focusing on retraining the model's fully-connected (FC) layer. This provides insight into the fact that for modern networks, the feature extractor is able to generate linearly separable activations in the feature space.

The rest of this chapter is organised as follows Sec.5.1 motivates the proposed solution. Sec. 5.2 describes the proposed methodology. Sec. 5.3 and 5.4 evaluate the methodology and its limitations.

5.1 Low-cost Training

The greyed out path in Fig.5.1 is an abstraction of a CNN network architecture. The architecture is split into a feature extractor ($\mathcal{M}^p_{FE}$) which takes as input an image and outputs activations that are then passed to the classifier ($\mathcal{M}^p_{FC}$) to predict the class of the image. The

[1] https://github.com/adityarajagopal/locopda

	ResNet18	MobileNetV2	ResNet50	MnasNet	GoogLeNet
Feature Extractor Training (GFLOPs)	815	1033	1869	1792	621
Classifier Training (MFLOPs)	262	655	1050	655	525
Network Inference, ρ (GFLOPs)	232	40	524	40	192

Table 5.1: Floating point operations performed (FLOPs) for the training stage of the feature extractor and classifier as well as inference stage of the entire network for various networks. The data is for input images size 224×224 and batch size of training is 128. Note feature extractor data is provided in GFLOPs while classifier data is provided in MFLOPs.

cost of training the feature extractor is significantly larger than that of the classifier. As stated in [10], the limiting factor of training CNNs on edge devices is the memory required to store intermediate activations used by backpropagation. Consequently, the primary cost of retraining a CNN lies in the feature extractor stage and bypassing this would significantly reduce the cost of retraining a network. This is demonstrated in Table 5.1 which shows the number of floating point operations performed (FLOPs) for training the feature extractor only and training the classifier only (no execution of the feature extractor) for various networks. On average across the networks, training the classifier alone requires 2.1×10^3 times fewer FLOPs than training the feature extractor.

One way to bypass the training of the feature extractor stage is to only train the classifier. Doing so would only require inference (not training) of the feature extractor to be performed. However, doing so also requires sufficient storage to store images with which retraining can be performed. This can be infeasible on edge devices due to the limited available memory. To perform retraining in this manner requires storage of at least one mini-batch worth of images. The standard mini-batch size for training networks on ILSVRC'12 is 128 to obtain good gradient estimates when performing SGD. For 224×224 sized images, this results in a minimum of 26MB of storage required which is comparable to the size of storing a network (γ) as seen from Table 2.1. Depending on the edge device being utilised, this could be an infeasible memory requirement.

Furthermore, Table 5.1 also presents the values of ρ (FLOPs for the inference stage of the network). Comparing these to the FLOPs for training the feature extractor provided in the table show that the approximately 2000x FLOPs benefit from classifier-only training mentioned earlier would be significantly reduced to only about 16x if inference were to also be performed.

Network	α_0 (%)	Type of classifier
AlexNet [44]	63.3	Non-linear
ResNet18 [28]	69.8	Linear
ResNet50 [28]	76.2	Linear
MobileNetV2 [67]	71.9	Linear
MnasNet [78]	73.5	Linear
SqueezeNet [33]	58.1	Non-linear
GoogLeNet [75]	69.8	Linear
EfficientNet [79]	82.9	Linear

Table 5.2: Top-1 ILSVRC'12 validation set classification accuracy for various networks (α_0) and a summary of the type of classifier used for each network.

Consequently, LoCO-PDA aims to retrain just the classifier but also bypass the execution of the feature extractor, thereby significantly reducing the costs of the retraining process.

5.2 Methodology

All networks in Table 5.2 except AlexNet and SqueezeNet have linear classifiers. Furthermore, the two networks with non-linear classification layers are the lowest performing in terms of classification accuracy on ILSVRC'12. Covering a wide range of modern CNNs the results in the table suggest a trend that most modern networks for image classification utilise linear classification layers with highly expressive feature extractors that create representations of images (activations) that can be linearly separated in feature space.

LoCO-PDA exploits this fact and uses light-weight VAEs to generate activations just before the classifier layer instead of performing inference through the feature extractor of $\mathcal{M}^p$. In the case of ResNet50, the VAE used to generate activations has a model memory consumption (γ) of 6.29 MB and inference stage operations (ρ) of 9.67 MFLOPs. These are 16x and 423x improvements in memory consumption and FLOPs respectively compared to performing inference using the feature extractor [2].

[2] γ and ρ of ResNet50 obtained from Table 2.1

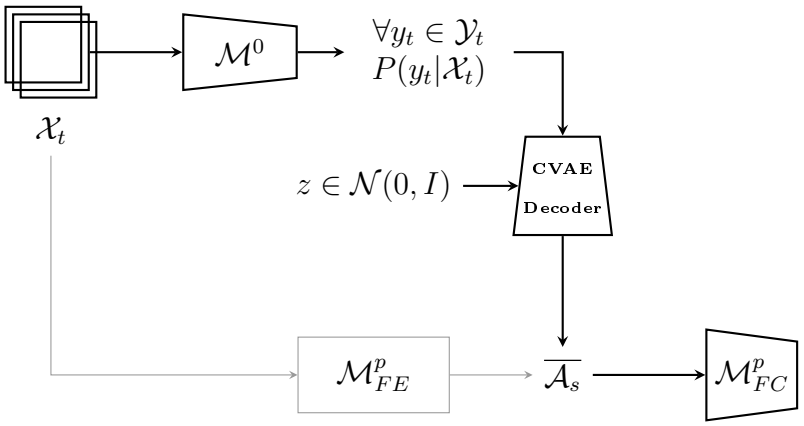


Figure 5.1: Proposed methdology for low-cost adaptation of the pruned network's classifier ($\mathcal{M}^p_{FC}$).

These generated activations, which capture the activation distribution on the original training dataset, can then be used to train the classifier. This process is described by the black path in Fig.5.1. $\mathcal{M}^0$ produces a set of predicted labels ($\mathcal{Y}_t$), for target domain images $\mathcal{X}_t$. The predicted classes along with their frequency of occurrence ($P(y_t|\mathcal{X}_t)$) are provided to the trained decoder. The decoder then generates a set of estimated activations $\overline{\mathcal{A}_s}$ which can be used to train $\mathcal{M}^p$'s classifier. Utilising $\mathcal{M}^p$ achieves gains in inference memory consumption and latency over $\mathcal{M}^0$, while retraining the classifier allows $\mathcal{M}^p$ to be optimised for classification of images from the observed data distribution. There is no added overhead to performing inference through $\mathcal{M}^0$ as this is the originally deployed model which is being used for inference and thus statistics of $P(y_t|\mathcal{X}_t)$ can be collected over time before initiating retraining.

5.2.1 VAE Training Process

LoCO-PDA uses an extension of VAEs known as Conditional-VAEs (CVAEs). VAEs [40] are generative models that learn to model a distribution $P(t)$ where t are high-dimensional data points which can represent a variety of inputs. They do this by learning a latent representation that can then be sampled to generate data points close to $P(t)$. In this chapter, t corresponds to the activations of $\mathcal{M}^p$ at the output of the feature extractor. CVAEs [73] allow for the sampling of the latent space to be conditioned on a desired class of t. One CVAE needs to be trained per source domain dataset $\mathcal{D}$ and model $\mathcal{M}^p$ and they are trained on $\mathcal{M}^p$'s activations for all images in $\mathcal{D}$ before deployment.

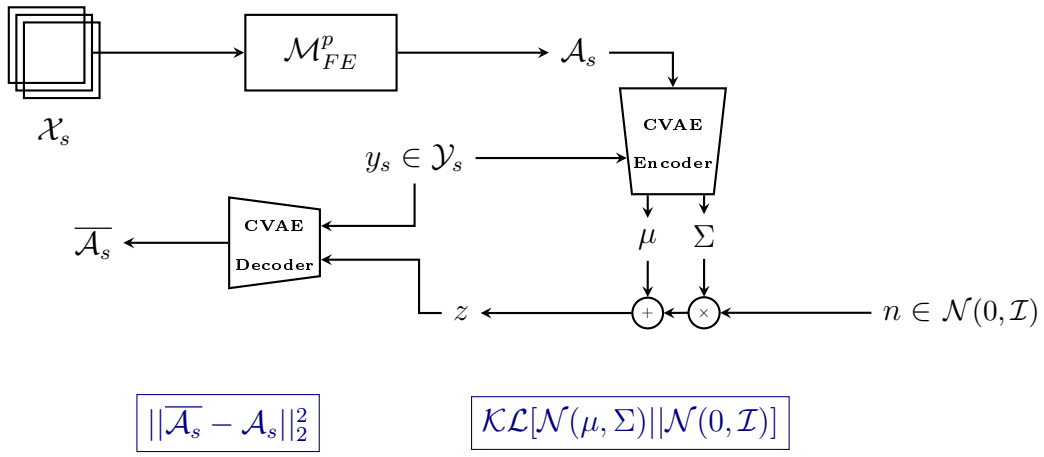


Figure 5.2: VAE training process used to train the decoder that is used in Fig.5.1.

Fig.5.2 shows the process of training the CVAEs to generate these activations. The whole process is performed offline using source domain images ($\mathcal{X}_s$) and available ground truth source domain labels ($\mathcal{Y}_s$). The CVAEs are trained to reproduce source domain activations ($\mathcal{A}_s$) when provided with the desired class y_s and a random vector z sampled from a multivariate standard normal distribution. The blue boxes show the loss functions used to train the CVAEs, where the left box is the reconstruction mean-squared-error (MSE) between the estimated activations $\overline{\mathcal{A}_s}$ and the true activations $\mathcal{A}_s$ and the right box is the KL-divergence between the learnt distribution of the latent space $\mathcal{N}(\mu, \Sigma)$ and $\mathcal{N}(0, \mathcal{I})$. The encoder learns the function $q_\phi(z|\mathcal{A}_s, y_s)$ which maps activations to latent space variables. The decoder then learns the function $p_\theta(\mathcal{A}_s|z, y_s)$ which generates an activation corresponding to the provided class. It learns to generate activations for all source domain classes ($\mathcal{Y}_s$) in order to be able to adapt to varying target domains.

The loss function used is that proposed in [30] for conditional VAEs and is given by:

$$\mathcal{L}(\theta, \phi; \mathcal{A}_s, z, y_s, \beta) = \mathbb{E}_{q_\phi(z|\mathcal{A}_s, y_s)}[\log p_\theta(\mathcal{A}_s|z, y_s)] - \beta\ \mathcal{KL}(q_\phi(z|\mathcal{A}_s, y_s)||p(\boldsymbol{z})) \tag{5.1}$$

The weights of the encoder and decoder that are learnt are given by ϕ and θ respectively. β is a parameter that balances the trade-off between reconstruction error and latent space regularisation. $p(\boldsymbol{z})$ is the prior distribution and is set to $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$ in order to obtain a closed form solution to the regularisation term.

After training, the conditional decoding process samples $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, augments the variable with y_s corresponding to the desired class and provides this as input to the trained decoder as shown in Fig.5.1. This process is facilitated by learning a latent space that closely follows $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$ (KL-divergence loss term), thus training a decoder that can map a randomly sampled variable in this distribution to the desired network activation based on the conditioning information.

As described in [5], the value of β changes over the course of training and is initialised at 0 (favouring reconstruction only) and gradually increased in steps of δ_β every ep_β epochs of training, until it reaches a value of 1. Furthermore, the training process has the hyperparameters of bs^{vae} (training batch size), lr_0^{vae} (inital learning rate), γ^{vae} (learning rate multiplier), ep_{lr}^{vae} (learning rate step size).

5.2.1.1 Training Hyperparameters

For all experiments in the following sections unless mentioned otherwise, the following values are used for the introduced hyperparameters:

- $|z| = 16$ (size of the latent space), $s = 1000$ (number of classes in ILSVRC'12)
- $q_\phi(z|\mathcal{A}_s, y_s)$ is estimated using a four layer fully connected neural network with layer sizes $[|\mathcal{A}_s| + s,\ 1024]$, [1024, 128], [128, 64], [64, $|z|$] with ReLU non-linearities.
- $p_\theta(\mathcal{A}_s|z, y_s)$ is estimated using a two layer fully connected neural network with layer sizes $[|z| + s,\ 512]$, [512, $|\mathcal{A}_s|$] with ReLU non-linearities.
- $\delta_\beta = 0.2$, $ep_\beta = 5$ - This is followed until $\beta = 1$ after which β remains unchanged.
- Adam [39] optimiser with $bs^{vae} = 512$, $lr_0^{vae} = 1e^{-3}$, $\gamma^{vae} = 0.1$, $ep_{lr}^{vae} = 30$ - This is set such that the learning rate only changes after the value of β has reached 1.
- 90 epochs of training are performed in total.

5.2.2 Identifying $\mathcal{D}'$

Once the VAE is trained, the decoder is deployed along with the initial deployment model $\mathcal{M}^0$ and the pruned model $\mathcal{M}^p$ that will be adapted to the observed data distribution. Before retraining $\mathcal{M}^p$ on $\mathcal{D}'$, it is necessary to identify the labels observed. In this work, we propose to identify these labels from the distribution predicted by the currently deployed network $\mathcal{M}^0$ (Fig.5.1).

5.2.3 Retraining $\mathcal{M}^p$

The retraining process gets as input a distribution $P(y_t|\mathcal{X}_t)$ which assigns a probability per observed class based on the frequency with which that class was observed. The tunable parameter R decides how many total activations to use when performing the classifier retraining. The R activations that are generated follow the class distribution $P(y_t|\mathcal{X}_t)$, thus a sufficiently small probability could result in no activation being generated for that class. Under the assumption that $\mathcal{M}^0$ predicts the correct class with a sufficiently high probability, this approach can be robust to noisy class distributions generated by $\mathcal{M}^0$ that may contain incorrect class predictions. These activations are then used to train $\mathcal{M}^p$'s classifier.

In summary, the proposed methodology utilises VAEs to skip execution of the feature extractor by generating activations, similar to those generated by the feature extractor, that follow the distribution of the classes observed. These activations can then be used to train the classifier of the network under adaptation in order to tune it to the observed data distribution. By avoiding execution of the feature extractor and only retraining the classifier, the proposed methodology significantly reduces the cost of adapting a network to the observed data distribution. Note as the VAEs are trained on source domain activations only, this methodology is tailored to the proposed problem setting where the distribution of the data points of the source and target domains are the same, and only the label space changes between the two domains.

5.2.3.1 Retraining hyperparameters

For all experiments in the following sections, the following values of hyperparameters are used:

- $R = 3000$
- Stochastic Gradient Descent (SGD) optimiser with 50 epochs of retraining, batch size 32, initial learning rate of $1e^{-6}$, learning rate changes every 15 epochs with $\gamma = 0.1$, momentum of 0.9 and no weight decay.

5.3 Evaluation

This section first evaluates LoCO-PDA under the constraints of a practical deployment scenario. Then Sec.5.3.7 compares LoCO-PDA to other state-of-the-art PDA methodologies and Sec.5.4 discusses its limitations.

Consider the following scenario, as proposed in Sec. 4.3.4, of an autonomous vehicle running an image classification CNN on an NVIDIA Jetson TX2 edge device. As the environment the car observes varies over time, the weights of a pruned version of the deployed CNN can be adapted to specialise this network to the data observed. Apart from the image classification network, there are other safety critical applications running on the device.

This scenario introduces the following considerations:

- As the device utilises a unified memory system, memory intensive processes such as retraining started on either processor can affect processes from executing on the other. In safety critical scenarios, this could have catastrophic consequences.
- On-device retraining of CNNs is primarily limited by the memory consumed by the training process and for many CNNs even batch sizes as low as 32 require too much memory.
- The robustness of the methodology to the lack of ground-truth labels of the observed data.

Consequently all approaches in this section are evaluated on the following metrics:

- Static (Δ) and runtime (Λ) memory consumption of the adaptation methodology.
- Classification accuracy of the adapted network on $\mathcal{D}'$ ($\bar{\alpha}$).
- Performance under noisy $\mathcal{D}'$ estimation.
- Wall-clock time to perform on-device adaptation on both the GPU and CPU (Θ).
- CNN model memory consumption (γ), inference stage latency (ϕ) and total inference stage floating-point operations (FLOPs) (ρ) of the adapted network.

5.3.1 Evaluation Datasets and Methodologies

This section details the methodologies that will be evaluated and the datasets that they will be evaluated on.

5.3.1.1 Datasets

For all experiments in this section, the source domain dataset ($\mathcal{D}$) on which model training and pruning are performed is the entire ILSVRC'12 training set. For experiments that evaluate PDA methodologies on the Office-31 dataset, an ILSVRC'12 trained network is further finetuned on the Office-31 source domain in order to obtain the initial deployment model.

For the practical deployment scenario, adaptation to the four example target domain datasets ($\mathcal{D}'$), introduced in Sec.4.3.4 will be explored. They are City (185 classes), Motorway (26 classes), Countryside (204 classes), Off-road (26 classes). Furthermore, for this evaluation, an edge-retraining dataset is created by randomly sampling 20% of the ILSVRC'12 training dataset. For each $\mathcal{D}'$, images from this dataset are utilised to emulate images that a deployed network would classify and are used to retrain the network on the edge. Although the deployed network has already observed these images during training on ILSVRC'12, a different random

seed is used for image preprocessing and augmentation to emulate variation from the training distribution. Doing so allows for all accuracy results to be reported on the full ILSVRC'12 validation dataset (unseen by models) to allow for fair comparison in the future.

As the problem setting addressed in this chapter is close to that of partial domain adaptation (Sec. 2.5.1), LoCO-PDA is also compared against other resource unconstrained PDA approaches. In the field of PDA, common domain adaptation datasets utilised are the Caltech-84 and Office-31 datasets. The Caltech-84 dataset corresponds to the 84 classes that are shared between the ILSVRC'12 and Caltech-256 [24] datasets. The Office-31 dataset consists of three categories corresponding to images from *Amazon*, a *DSLR* and a *Webcam* and each category has images corresponding to 31 classes. The PDA setting considers domain adaptation between all combinations of the 3 categories, where the source domain contains all 31 classes and the target domain contains only 10 classes as in [12]. These target domains are common transfer learning benchmarks and details of the classes present in all target domains are provided below.

The 84 classes shared by the ILSVRC'12 and Caltech-256 datasets which forms the Caltech-84 dataset are: hot-air-balloon, leopards, centipede, mountain-bike, llama, umbrella, microwave, fighter-jet, fire-truck, ak47, computer-keyboard, killer-whale, car-tire, chimp, traffic-light, goose, computer-monitor, mushroom, french-horn, school-bus, house-fly, coffee-mug, photocopier, gorilla, ice-cream-cone, rifle, penguin, greyhound, snail, hamburger, scorpion, tripod, skunk, teapot, laptop, praying-mantis, video-projector, gas-pump, triceratops, spoon, binoculars, syringe, hourglass, camel, mailbox, ostrich, speed-boat , trilobite, grand-piano, refrigerator, cockroach, hot-dog, ipod, canoe, beer-mug, harmonica, screwdriver, conch, hummingbird, backpack, soccer-ball, golden-gate-bridge, grasshopper, golf-ball, wine-bottle, harp, hot-tub, football-helmet, owl, socks, starfish, tennis-ball, cannon, dumb-bell, cowboy-hat, bathtub, goldfish, porcupine, iguana, tricycle, toaster, zebra, electric-guitar, self-propelled-lawn-mower.

The partial domain adaptation scenario ILSVRC'12 -> C84 refers to a source domain containing all 1000 classes of ILSVRC'12 and a target domain containing only these 84 classes from the Caltech-84 dataset.

The 31 classes in the source domain for the *Amazon*, *DSLR* and *Webcam* datasets consist

Figure 5.3: Examples of images in each of the three domains of Office31 (Amazon, DSLR and Webcam) for three different classes (Mug, Ruler, Back Pack).

of classes: back pack, ring binder, laptop computer, file cabinet, mobile phone, bookcase, tape dispenser, desk chair, scissors, punchers, bike helmet, trash can, bike, desktop computer, printer, calculator, paper notebook, monitor, mouse, phone, projector, ruler, headphones, letter tray, bottle, keyboard, pen, mug, desk lamp, stapler, speaker

The 10 classes in the target domain for each of *Amazon*, *DSLR* and *Webcam* datasets consist of classes: mouse, calculator, projector, bike, back pack, headphones, mug, monitor, keyboard, laptop computer.

Therefore, the PDA task A->W for instance corresponds to the 31 classes from the source domain *Amazon* and 10 classes in the target domain *Webcam*. Fig.5.3 provides some examples of images in the three domains for three different classes. Within the same class, the Amazon domain has drastically different images in both resolution and type compared to the other two domains. However the DSLR and Webcam domains primarily just vary in the resolution and quality of colours in the images.

5.3.1.2 Methodologies

As this chapter does not investigate the scenario where architecture selection (model pruning) is performed on the device itself, the methodologies that LoCO-PDA is compared against, store a set of N possible pruned models on-device, each of which can be retrained to the observed data distribution. The choice of model that is retrained depends on the target inference memory and latency at the time of deployment. The methodologies vary in either pruning strategy or the manner in which multiple networks with different budgets are exposed. Additionally, the comparisons are split into two categories, namely resource constrained class domain adaptation approaches and resource unconstrained PDA approaches.

The following two methodologies fall under resource constrained class domain adaptation. For these, the source and target domains differ only in the label space and not the distribution of the data points and the evaluated target domains are the 4 autonomous driving subsets of ILSVRC'12.

On-device Retraining A set of N pruned models are obtained using both TFO and OFA pruning (Sec.2.3). All TFO pruning is performed using the tool *Autoprune*. For both pruning strategies, the following two on-device retraining approaches are considered as baselines:

- All-layers retraining on $\mathcal{D}'$ (both feature extractor and classifier)
- Classifier-only retraining on $\mathcal{D}'$ (by performing inference through the feature extractor)

PersEPhonEE [47] (Early-exit Retraining) This methodology appends early-exit modules which contain two convolution layers and a classifier at various points along the backbone network. After training of all exits and the backbone network offline on $\mathcal{D}$, on-device retraining cost is minimised by only finetuning the early exit that falls within the available memory and compute budget.

To the best of our knowledge, PersEPhonEE is the only other work that explores edge device retraining for the class domain adaptation setting described here. The TFO and OFA approaches are baselines to evaluate the robustness of LoCO-PDA to different pruning strategies.

For all methodologies, both memory bounded and unbounded training approaches are evaluated. This refers to placing a memory limit on the number of images that can be stored in order to perform retraining versus utilising the entire retraining dataset which as discussed in Sec.5.3.1.1 corresponds to 20% of the ILSVRC'12 training dataset. In the memory bounded scenario, a limit of 400MB is set as this is deemed to be a reasonable allocation of the available 8GB on a Jetson TX2 for this purpose. In the memory unbounded scenario, as each $\mathcal{D}'$ has a different number of classes, the storage memory utilised varies and is 9.40 GB, 1.36 GB, 10.3 GB and 1.36 GB for the City, Motorway, Countryside, and Off-road $\mathcal{D}'$ respectively. The memory unbounded scenario evaluates the best case performance of each methodology when not subjected to any memory constraints. All retraining based approaches are evaluated on the NVIDIA Jetson TX2 using PyTorch v1.6, CUDA 10.2, cuDNN 8.0 and batch-size 32.

For the resource unconstrained PDA scenario, LoCO-PDA is compared against state-of-the-art (at time of writing) PDA approaches **ETN** [12], **IWAN** [87] and **PADA** [11] which are

		$\boldsymbol{\rho}$ (MFLOPs)	$\boldsymbol{\gamma}$ (MB)	$\boldsymbol{\phi}$ (ms)	$\boldsymbol{\alpha_0}$ (%)
TFO	$\mathcal{M}^0_{TFO}$	4087	102	45.13	76.12
	$\mathcal{M}^{80}_{TFO}$	1250 (3.27x)	41 (2.48x)	36.66 (1.23x)	63.74 (-12.47pp)
OFA	$\mathcal{M}^0_{OFA}$	3496	126	42.34	83.89
	$\mathcal{M}^{80}_{OFA}$	1197 (2.92x)	40 (3.15x)	20.44 (2.07x)	81.75 (-2.14pp)
PerseEPhonEE	$\mathcal{M}^0_{PE}$	4087	143	45.13	76.12
	$\mathcal{M}^{80}_{PE}$	1246 (3.28x)	143 (1x)	16.65 (2.71x)	16.91 (-59.21pp)

Table 5.3: Summary of inference operations (ρ), model memory footprint (γ), Jetson TX2 single image inference latency (ϕ) and ILSVRC'12 Test Top1 accuracy (α_0) for the target networks.

introduced in Sec.2.5.1. Furthermore, the assumption here is that both the distribution of the data points within a class and the label space vary between the source and target domains.

5.3.1.3 Networks

The $\mathcal{M}^0$ and $\mathcal{M}^p$ networks utilised in this evaluation are shown in Table.5.3. As discussed in Sec.5.3.1.2, N $\mathcal{M}^p$ networks are stored on device, however for simplicity this section only evaluates the case of $N = 1$.

$\mathcal{M}^0_{TFO}$ is an unpruned ResNet50 network and $\mathcal{M}^{80}_{TFO}$ is the same network with 80% of its parameters pruned using the TFO pruning methodology. Table 2.2 demonstrated that ResNet50 could be pruned by 70% with only a 4pp loss in classification accuracy on the entire ILSVRC'12 dataset. $\mathcal{M}^0_{OFA}$ is the largest sub-network that can be sampled from the OFA super network and $\mathcal{M}^{80}_{OFA}$ is an OFA sub-network that has a similar inferece stage FLOPs budget to $\mathcal{M}^{80}_{TFO}$. $\mathcal{M}^0_{PE}$ is an unpruned ResNet50 network augmented with 6 early exits located at inference stage FLOPs normalised locations along the network. Consequently, apart from *model memory*, all other metrics are the same as $\mathcal{M}^0_{TFO}$ in Table.5.3. $\mathcal{M}^{80}_{PE}$ is the second early exit (after layer2.1) as this has a similar FLOPs budget to $\mathcal{M}^{80}_{TFO}$. Note that the accuracy quoted here is α_0, the Top1 classification accuracy on the entire ILSVRC'12 dataset and not just the subset $\mathcal{D}'$. Only versions of ResNet50 are considered in the evaluation here as the state-of-the-art methodologies in the related field of PDA that are compared to (PADA, IWAN and ETN), only provide results

for this network. Sec.5.4 further extends the evaluation of resource constrained class domain adaptation (label space variation only) to MobileNetV2.

The TFO pruning methodology pruned 80% of the network, reducing ρ by 3.27x, γ by 2.48x, ϕ by 1.23x while incurring a 12.47pp classification accuracy loss on the entire ILSVRC'12 dataset. Similar comparisons between $\mathcal{M}^0$ and $\mathcal{M}^{80}$ for all methodologies are provided in Table 5.3.

The rest of the evaluation is organised as follows. Sec.5.3.2 first explores the impact on the memory consumed by the adaptation process when performing All-layer retraining, classifier-only retraining, early-exit retraining and LoCO-PDA. Following this, Sec.5.3.3 explores the achieved post adaptation classification accuracy for both the memory bounded and unbounded scenarios discussed in Sec.5.3.1.2. This evaluation is performed under the assumption of knowledge of ground-truth labels of the observed images. In doing so, this section presents both a baseline for comparison when label-uncertainty is accounted for as well as an idea of the trade-off between adaptation process memory consumption to accuracy achieved of the various methodologies. Sec.5.3.5 then evaluates LoCO-PDA against other methodologies in the scenario where ground-truth labels are not available. Sec.5.3.6 evaluates the wall-clock time to perform this domain adaptation. All results until here assume that the source and target domains only vary in the label space and not the data distribution within a class. Sec.5.3.7 then extends the evaluation of LoCO-PDA to the full PDA problem setting where both the label space and the within class data distribution varies.

5.3.2 Static (Δ) and Runtime (Λ) Memory Footprint

For a given adaptation process $\mathcal{P}$, the static memory footprint (Δ) of a methodology corresponds to the storage memory consumed by the various parts of the methodology required to perform the network adaptation process. Runtime memory footprint (Λ) corresponds to the memory loaded on to the device's RAM during the network adaptation process. The total static and runtime memory consumption of the various methodologies on a Jetson TX2 is displayed in Table.5.4. The network memory consumption corresponds to the weights of the model and

		Δ (MB)			Λ (MB)
		NETWORK	IMAGES	TOTAL	
TFO	Classifier-only (GPU)	143	400	543	1668
	All-Layers (GPU)				3508
	LoCO-PDA (CPU)	153	0	**153**	**13**
	LoCO-PDA (GPU)				1143
OFA	Classifier-only (GPU)	166	400	566	1865
	All-Layers (GPU)				2800
	LoCO-PDA (CPU)	183	0	**183**	**28**
	LoCO-PDA (GPU)				1128
PERSEPHONEE (GPU)		143	400	543	1971

Table 5.4: Static (Δ) and runtime (Λ) memory consumption of the various methodologies. NETWORK refers to the model storage requirements. IMAGES refers to the storage of images for retraining.

the memory consumption of images comes from the memory bounded scenario introduced in Sec.5.3.1.2. *Classifier-only* refers to training just the classifier of a network but performing inference through the feature extractor and *All-layers* refers to training both the feature extractor and classifier. The remainder of this section discusses the results presented in Table 5.4.

5.3.2.1 TFO, OFA: Image based retraining

For each methodology (TFO, OFA), the network storage component consists of the corresponding $\mathcal{M}^0$ to estimate labels and $\mathcal{M}^{80}$ which is adapted through retraining. Note depending on the number (N) of pruned networks ($\mathcal{M}^p$) stored, the static network storage component will increase. As mentioned in Sec.5.3.1.3, for simplicity, the case of $N = 1$ is evaluated here. The static image storage component consists of 400MB of images for memory bounded retraining. The runtime memory consumption is affected by whether classifier-only or all-layer retraining is performed and is quoted for retraining $\mathcal{M}^{80}$ on the GPU of the Jetson TX2.

	TFO		OFA		PersEPhonEE
	All-layers	Classifier-only	All-layers	Classifier-only	
LoCO-PDA (GPU)	3.07x	1.16x	2.48x	1.65x	1.72x

Table 5.5: Summary of results presented in Table 5.4 in terms of reduction (-x times) in runtime adaptation memory consumption of LoCO-PDA compared to on-device retraining approaches when executed on a GPU.

5.3.2.2 PersEPhonEE: Early-exit retraining

The network component of the static memory consumption is from $\mathcal{M}^{80}_{PE}$. The images component of the static memory consumption consists of 400 MB of images stored to perform memory bounded early-exit retraining. The runtime component is from performing on-device retraining of the second early exit (Sec.5.3.1.3) on the GPU.

5.3.2.3 LoCO-PDA

For LoCO-PDA, the additional network storage component is the storage of the CVAE decoder used to generate activations. This component is 9.65 MB for $\mathcal{M}^{80}_{TFO}$ and 16.83 MB for $\mathcal{M}^{80}_{OFA}$. As PersEPhonEE uses $\mathcal{M}^{0}_{TFO}$ as the backbone network, the CVAE trained on activations generated by $\mathcal{M}^{80}_{TFO}$ is used for comparison. No images are used for retraining with this approach, hence the static image component is 0 MB. The memory consumed by the generated activations is incorporated into the runtime memory consumption which is evaluated both on the CPU and GPU of the Jetson TX2. The significantly extra memory consumption of the GPU is due to CUDA initialisation overheads which can be close to 1 GB depending on the device and framework combination. The results demonstrate that on average across methodologies, LoCO-PDA reduces the static memory consumed by the adaptation process by 3.38x.

A summary of the runtime memory consumption results from Table 5.4 is presented in Table 5.5. The table provides the reduction (-x times) in runtime memory consumption of the adaptation process that is achieved by LoCO-PDA compared to other methodologies. Note only GPU results are used here in order to perform a like-to-like comparison of methodologies as running classifier-only or all-layer retraining on a CPU was not considered. On average across all methodologies, LoCO-PDA reduces the runtime memory consumption of the adaptation process

by 2.02x when executed on a GPU.

5.3.3 Accuracy of $\mathcal{M}^p$ on $\mathcal{D}'(\bar{\alpha})$

This section compares the best achieved Top1-Test accuracy ($\bar{\alpha}$) after adapting the network to the four autonomous driving $\mathcal{D}'$s for all the methodologies discussed in Sec.5.3.1.2. In order to evaluate the methodologies under a no-uncertainty situation, all results here assume knowledge of the ground-truth labels for the images in $\mathcal{D}'$. Later sections evaluate the effect of uncertain labels when using $\mathcal{M}^0$ to identify $\mathcal{D}'$.

For the TFO and OFA methodologies, 10 epochs of retraining are performed with an initial learning rate of $1e^{-3}$ which is decayed by 0.1 every 3 epochs. For PersEPhonEE, as discussed in [47], 10 epochs of retraining are performed with an initial learning rate of $1e{-}2$ that is left unchanged. For LoCO-PDA, the CVAE model and hyperparameters discussed in Sec.5.2 are used.

First, memory bounded and unbounded; and all-layers and classifier-only training approaches are evaluated. The rest of this section discusses these results that are presented in Table.5.6.

5.3.3.1 Memory bounded vs. unbounded

As presented in Table 5.6, the memory unbounded approach outperforms the memory bounded approach by 2.64pp on average across $\mathcal{D}'$ and methodologies. However, from Sec.5.3.1.2, the memory bounded scenario uses only 400MB of memory to store images while depending on $\mathcal{D}'$, the memory unbounded scenario can take up to 10.3GB of memory. Considering that for some $\mathcal{D}'$ the memory unbounded approach does not fit on the available device memory (uses more than 8GB of memory), and for those which do it requires 3.4x the amount of memory compared to the memory bounded approach (Sec.5.3.1.2), the comparatively small trade-off in adaptation accuracy (2.64pp) is considered acceptable. Consequently, the rest of the evaluation, unless mentioned otherwise, will utilise the memory bounded approach for baseline comparison.

	All Layers		Classifier Only		
	$\mathcal{M}^{80}_{TFO}$	$\mathcal{M}^{80}_{OFA}$	$\mathcal{M}^{80}_{TFO}$	$\mathcal{M}^{80}_{OFA}$	$\mathcal{M}^{80}_{PE}$
Memory Bounded	76.88 ± 6.14	81.44 ± 4.34	75.15 ± 6.52	82.83 ± 4.59	23.79 ± 10.52
Memory Unbounded	79.59 ± 4.91	83.14 ± 4.46	79.22 ± 5.23	84.63 ± 4.77	26.70 ± 10.87

Table 5.6: Best achieved Top1-Test accuracy ($\bar{\alpha}$) for the various baseline approaches to on-device retraining. Values are presented as $mean \pm std$ over all four autonomous driving $\mathcal{D}'$.

	$\mathcal{M}^{0}_{TFO}$ (%)	$\mathcal{M}^{0}_{OFA}$ (%)	$\mathcal{M}^{80}_{TFO}$ (%)			$\mathcal{M}^{80}_{OFA}$ (%)		
	No Retrain	No Retrain	No Retrain	Retrained	**LoCO-PDA**	No Retrain	Retrained	**LoCO-PDA**
City (185)	78.01	81.64	70.89	71.11	73.84	78.50	80.37	80.73
Motorway (26)	74.54	78.28	68.23	71.28	76.75	74.85	79.91	79.86
Countryside (204)	78.48	81.98	71.36	71.78	74.03	79.07	80.28	81.22
Off-road (26)	82.87	85.99	76.20	86.43	87.30	82.32	90.78	89.94

Table 5.7: On-device domain adaptation accuracy ($\bar{\alpha}$) of $\mathcal{M}^{80}_{TFO}$ and $\mathcal{M}^{80}_{OFA}$ under the scenario where ground truth labels are known in the target domain. The number of classes in each $\mathcal{D}'$ is provided in brackets.

5.3.3.2 All layers vs. Classifier-only

From Table 5.6, retraining all layers performs on average over memory bounded and unbounded approaches 1.05pp better and 1.44pp worse than classifier-only retraining for $\mathcal{M}^{80}_{TFO}$ and $\mathcal{M}^{80}_{OFA}$ respectively. Furthermore, retraining all layers has a runtime memory footprint that is 1.6x (Table 5.4) and training stage FLOPs that is 16x larger (Sec.5.1) than classifier-only training. Hence the following sections use classifier-only retraining as a baseline as the gain in accuracy for TFO networks when retraining all-layers is marginal and does not justify the significantly larger memory footprint.

5.3.3.3 LoCO-PDA

The previous sections identified the trade-offs between accuracy achieved and memory consumed by the various approaches to on-device retraining, namely, memory-bounded and unbounded approaches and all-layers and classifier-only retraining approaches. In doing so, it identified that classifier-only, memory-bounded retraining to provide the best trade-off between the memory consumed and achieved accuracy. This section compares LoCO-PDA to this approach to retraining applied to both the TFO and OFA pruned networks.

Table.5.7 displays the achieved Test Top1 accuracy on each autonomous driving $\mathcal{D}'$ for $\mathcal{M}^{80}_{TFO}$ and $\mathcal{M}^{80}_{OFA}$ when no retraining (No Retrain), memory bounded classifier-only retraining

	City (pp)	**Motorway** (pp)	**Countryside** (pp)	**Off-Road** (pp)
$\mathcal{M}^{80}_{TFO}$	+2.73	+5.47	+2.25	+0.87
$\mathcal{M}^{80}_{OFA}$	+0.36	-0.05	+0.94	-0.84

Table 5.8: Summary of results presented in Table 5.7 in terms of difference (in precentage points) in post adaptation classification accuracy for each autonomous driving subset between LoCO-PDA and memory bounded, classifier-only, on-device retraining approaches.

(Retrained) and LoCO-PDA are performed on-device; all under the no label uncertainty scenario. This evaluation acts as an "oracle" for the performance achieved when retraining a network directly on the target domain. PersEPhonEE was not compared against here due to its poor performance ($\mathcal{M}^{80}_{PE}$ in Table.5.6). To allow for a FLOPs normalised comparison between methodologies (5.3.1.3), the PersEPhonEE methodology utilises an early-exit placed only after the second Botttleneck block of ResNet50. The utilisation of an early-exit so early on in the network is likely the cause of its poor performance.

Table 5.8 provides a summary of the results in Table 5.7 and presents the difference, in percentage points, of the achieved classification accuracy per subset between LoCO-PDA and memory bounded, classifier-only retraining (Retrained). On average across datasets and retraining methodologies, LoCO-PDA outperforms memory bounded retraining of $\mathcal{M}^{80}$ by 1.46pp. However, these accuracy gains are significantly greater when retraining TFO networks as opposed to OFA networks. This is likely due to the fact that the initial pruned OFA network has a much smaller reduction in accuracy compared to $\mathcal{M}^0$ than TFO (Table 5.3). Consequently, there is less accuracy to be gained from retraining and hence even with a memory limited retraining scenario, we can regain most of the accuracy lost by pruning. Nonetheless, LoCO-PDA executed on a GPU has a 1.65x improvement in adaptation memory footprint (Table 5.5). Furthermore, as seen in Table 5.7, on the smaller $\mathcal{D}'$ (Motorway, Off-road), on which domain adaptation is more beneficial, LoCO-PDA also outperforms its corresponding unpruned network ($\mathcal{M}^0$) by 3.04pp on average across subsets and methodologies.

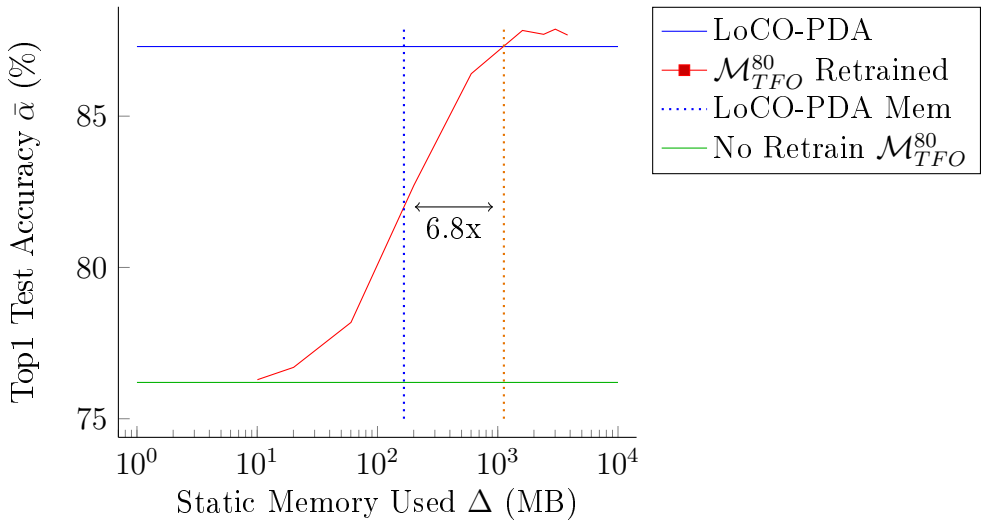


Figure 5.4: Effect of available image storage memory on achieved retraining accuracy for the Off-road $\mathcal{D}'$. Please note the logarithmic scale on the x-axis.

5.3.4 Storage Memory Limitations

The results from the previous sections utilise a storage memory limit of 400MB for image based retraining approaches. However, as discussed in Sec.5.3.3.1 placing this restriction did have a reduction in accuracy when compared to the memory unbounded scenario. This section expands on this to demonstrate the effect of various storage limits on the achieved accuracy when performing image based classifier-only retraining. The red line in Fig.5.4 displays the achieved classifier-only retraining accuracy for $\mathcal{M}^{80}_{TFO}$ on the Off-road $\mathcal{D}'$ as the available memory to store images is varied (x-axis). These values are the average of five retraining runs. The solid green line displays the no retraining accuracy of $\mathcal{M}^{80}_{TFO}$ and the solid blue line displays the accuracy achieved by LoCO-PDA. The dashed blue line is the static memory consumed by LoCO-PDA. The graphs show that as the available memory budget is increased, the achieved retraining accuracy improves and depending on $\mathcal{D}'$ can outperform LoCO-PDA under no label uncertainty while utilising 6.8x times the memory. However, for the same memory budget as LoCO-PDA, the memory-bounded retraining approach performs on average 4.88pp worse than LoCO-PDA.

	$\mathcal{M}^0_{TFO}$ (%)	$\mathcal{M}^0_{OFA}$ (%)	$\mathcal{M}^{80}_{TFO}$ (%)			$\mathcal{M}^{80}_{OFA}$ (%)		
	No Retrain	No Retrain	No Retrain	Retrained	**LoCO-PDA**	No Retrain	Retrained	**LoCO-PDA**
City (185)	78.01	81.64	70.89	66.36	72.33	78.50	76.59	80.67
Motorway (26)	74.54	78.28	68.23	53.93	75.20	74.85	73.49	79.05
Countryside (204)	78.48	81.98	71.36	67.82	73.04	79.07	78.12	80.56
Off-road (26)	82.87	85.99	76.20	72.13	86.88	82.32	85.38	90.55

Table 5.9: On-device domain adaptation accuracy ($\bar{\alpha}$) of $\mathcal{M}^{80}_{TFO}$ and $\mathcal{M}^{80}_{OFA}$ under the scenario where target domain labels are estimated using $\mathcal{M}^0$. The number of classes in each $\mathcal{D}'$ is provided in brackets.

5.3.5 Noisy $\mathcal{D}'$ Estimation

All results presented thus far assume correct knowledge of the label space of $\mathcal{D}'$ which is not representative of real deployment scenarios where ground-truth labels of the observed images are not available. In this chapter we propose to estimate the label space of $\mathcal{D}'$ using the predictions of the initially deployed model $\mathcal{M}^0$. As the accuracy of this model is not 100%, some of the labels obtained from this model are incorrect. This section evaluates the effect on achieved adaptation accuracy under such noisy $\mathcal{D}'$ estimation. The following results are averages across both the OFA and TFO networks.

Table 5.9 shows the achieved test top1 accuracy on $\mathcal{D}'$ for both the memory bounded classifier-only retraining strategy (Retrained) and the LoCO-PDA approach when the target domain labels are estimated using $\mathcal{M}^0$. Note that the No Retrain columns are the same as in Table 5.7. For the image based retraining approaches, memory bounded retraining with uncertain labels causes a 7.27pp degradation in achieved accuracy compared to retraining with availability of ground-truth labels across $\mathcal{D}'$ (Table 5.7). Furthermore in most cases, retraining with noisy labels actually degrades the network's accuracy compared to $\mathcal{M}^{80}$ without any retraining. On the other hand, LoCO-PDA only incurs a 0.67pp reduction in achieved accuracy across $\mathcal{D}'$ when compared to using no-uncertainty class distribution predictions (Table 5.7) and never a degradation compared to $\mathcal{M}^{80}$ without retraining.

With image based retraining, predicting an incorrect label and utilising this combination to retrain the network causes classification accuracy degradation as the network is now being trained to associate incorrect label and image pairs. As LoCO-PDA generates the activations corresponding to the estimated class, only an incorrect ratio of the labels is generated and the retraining stage still associates labels with their correct corresponding activations. Thus,

	TFO		OFA	
	RETRAINED	**LoCO-PDA**	RETRAINED	**LoCO-PDA**
GPU (MIN:SEC)	02:43	00:12	02:02	00:10
CPU (MIN:SEC)	329:53	00:46	344:00	00:54

Table 5.10: The on-device network adaptation time (Θ) for various methodologies averaged across $\mathcal{D}'$. RETRAINED refers to 400MB memory-limited classifier-only training.

the proposed VAE approach is significantly more robust to noisy estimations of $\mathcal{D}'$ than image based retraining approaches.

5.3.6 On-device Adaptation Wall-clock Time (Θ)

This section evaluates the wall-clock time to perform on-device adaptation using each methodology on the Jetson TX2. The values in Table 5.10 are the time taken to retrain the pruned network until the accuracies in Table 5.7 are achieved. As different $\mathcal{D}'$ behave differently under retraining, the quoted values are averaged across the four $\mathcal{D}'$ (City, Off-road, Motorway and Countryside).

GPU times for the time taken to perform on-device adaptation for image based retraining approaches is are utilised here for comparison. The results in Table 5.10 show that LoCO-PDA executed on a CPU outperforms the classifier-only retraining on a CPU by 406x and even outperforms classifier-only retraining executed on a GPU by 2.9x. If LoCO-PDA were to be executed on a GPU instead, the gains in wall-clock runtime would be 12.9x compared to classifier-only retraining executed on a GPU.

By bringing domain adaptation latencies down to less than a minute, while consuming less than 300 MB of memory and being robust to noisy estimations in $\mathcal{D}'$, LoCO-PDA opens up avenues for on-device partial domain adaptation of networks that have not been explored before.

5.3.7 Partial Domain Adaptation

This section focuses on the comparison of LoCO-PDA against state-of-the-art PDA methodologies ETN [12], IWAN [87], and PADA [11]. The setting here differs from previously discussed results in two ways. First, as this explores the complete PDA problem, both the distribution of data within a class as well as the label space change between the source and target domains. Furthermore, the PDA works compared to here have not been optimised for resource constraints. Although a comparison of the resource consumption of these methdologies is presented in this section, their main utility is in providing a comparison to LoCO-PDA in terms of achieved post adaptation classification accuracy as LoCO-PDA is a methodology that is only tailored to the shift in the label space and not the within class data distribution.

For the ILSVRC'12 to Caltech-84 transfer setting, activations corresponding to the 84 classes shared between the ILSVRC'12 and Caltech-256 datasets are generated using a CVAE trained on ILSVRC'12 data that has no knowledge of the target Caltech-84 activation distribution. Furthermore, the target distribution is estimated using the output of classifying the entire Caltech-84 dataset on the unpruned ResNet50 network, hence no ground-truth labels were used. For the Office-31 setting, ResNet50 is first finetuned to each of the three categories using all 31 classes and 3 CVAEs are trained to generate activations for each of these source domain networks. The target domain images are then classified on the source domain finetuned network to obtain a distribution and the corresponding CVAE is used to generate activations to retrain the network.

As discussed in Sec. 2.5.1, ETN, IWAN or PADA do not optimise for the memory and compute footprint of the adaptation process (Λ, Θ) or the resulting adapted network (γ, ϕ, δ) - all of which are important considerations for real-life applications. However, as demonstrated in Fig.5.5, LoCO-PDA aims to optimise for the trade-off between $\bar{\alpha}$, Λ and Θ. In order to normalise across the methodologies for the number of iterations of adaptation performed, Fig.5.5 plots the adaptation latency per iteration for the same batch size of iteration across all methodologies on the x axis. An iteration is defined as 1 mini-batch worth of data being processed by the domain adaptation process. The test accuracy quoted is the average across all Office-31 transfer

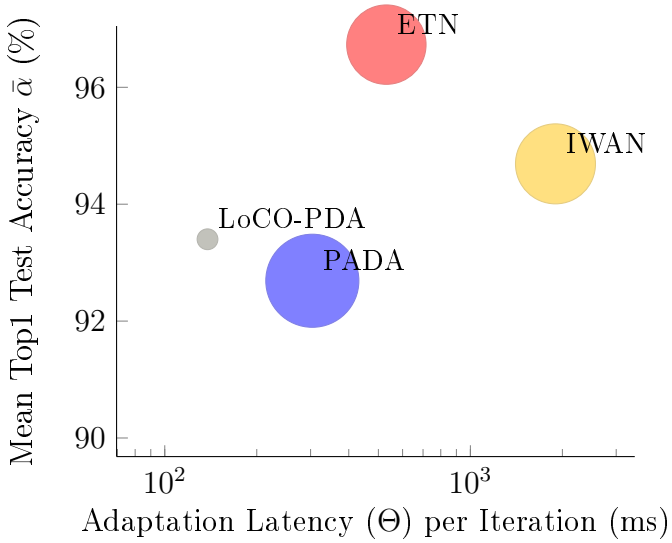


Figure 5.5: Adaptation process latency Θ per iteration (x-axis), memory Λ (dot-area) and mean accuracy $\bar{\alpha}$ (y-axis) trade-off across PDA transfer tasks on the Office-31 dataset for SOTA PDA methodologies and LoCO-PDA. One iteration is defined as 1 mini-batch worth of data being processed by the domain adaptation process. Please note the logarithmic x-axis.

tasks, where a per task breakdown is provided in Table.5.11. Compared to state-of-the-art PDA methodologies, LoCO-PDA achieves a comparable adaptation accuracy while improving Θ and Λ on average by 7x and 4x respectively.

From solely an accuracy perspective, on average across all PDA settings, Table.5.11 shows that LoCO-PDA improves performance compared to no domain adaptation (ResNet50 column) and PADA by 5.42pp and 1.20pp respectively; and performs comparably to IWAN (only 0.95pp worse). Although performing 3.43pp worse compared to ETN, LoCO-PDA has the significant advantage over all three state-of-the-art PDA methodologies that it can rapidly adapt to changes in the target domain, where ETN, IWAN and PADA all consume close to 8GB of memory during adaptation while requiring access to the entire training dataset. Hence, applying such methodologies directly on edge devices to adapt to changes in the target domain would require significant further research to optimise them for the resource constrained setting.

		ResNet50	SOTA			LoCO-PDA
			ETN	IWAN	PADA	
ILSVRC'12 → C84		69.69	83.23	78.06	75.03	79.17
Office31	A → W	76.03	94.52	89.15	86.54	84.64
	A → D	91.12	95.03	90.45	82.17	90.56
	D → W	94.06	100.0	99.32	99.32	97.50
	D → A	83.62	96.21	95.62	92.69	93.74
	W → A	88.42	94.64	94.26	95.41	93.95
	W → D	98.69	100.0	99.36	100.0	100.0

Table 5.11: Comparison of domain adapted accuracy ($\bar{\alpha}$) of LoCO-PDA and SOTA Partial Domain Adaptation approaches. The domains are Caltech-84 (C84), Amazon (A), DSLR (D) and Webcam (W). The ResNet50 column corresponds to a ResNet50 network trained on ths source domain and not specialised in anyway.

5.4 Further Experiments

This section discusses additional experiments performed to identify the limits of LoCO-PDA. We evaluate the generalisability of LoCO-PDA to other network architectures and investigate the difference in class domain adaptation capabilities between conditional and unconditional VAEs.

5.4.1 Other network architectures

The results from Sec.5.3 demonstrate that LoCO-PDA generalises well across different pruning strategies (TFO, OFA) for the same backbone network (ResNet50). This section evaluates LoCO-PDA's generalisability to other networks by evaluating it on MobileNetV2 [67] and VGG11 [71].

5.4.1.1 MobileNetV2

Table.5.12 displays the accuracy regained by performing on-device memory bounded, classifier-only training and LoCO-PDA on the four autonomous driving $\mathcal{D}'$. $\mathcal{M}^0_{TFO}$ is an unpruned MobileNetV2 with 312.3 MFLOPs of inference cost (ρ) and 14.09 MB of model memory (γ).

	$\mathcal{M}^0_{TFO}$ (%)	$\mathcal{M}^{50}_{TFO}$ (%)				
		No Uncertainty			Uncertainty	
	No Retrain	No Retrain	Retrained	**LoCO-PDA**	Retrained	**LoCO-PDA**
City (185)	74.44	65.72	68.80	69.70	60.45	68.09
Motorway (26)	72.03	63.87	71.51	72.55	57.91	69.34
Countryside (204)	74.96	66.61	69.18	70.35	63.57	69.30
Off-road (26)	78.85	70.73	85.78	86.19	77.30	82.27

Table 5.12: On-device domain adaptation accuracy ($\bar{\alpha}$) of a 50% pruned MobileNetV2 network. The table titles are the same as in Table.5.7.

$\mathcal{M}^{50}_{TFO}$ is a 50% TFO pruned MobileNetV2 network with ρ of 166.6 MFLOPs and γ of 7.06 MB.

Across all $\mathcal{D}'$, LoCO-PDA performs 0.88pp better than memory bounded classifier-only retraining. However, although the image based retraining methodology degrades accuracy by 9.01pp in the label uncertain setting compared to label certain setting, LoCO-PDA also degrades accuracy by 2.45pp. This could be due to the lower accuracy of the unpruned MobileNetV2 compared to the networks in Sec.5.3, thus demonstrating the sensitivity of LoCO-PDA to an overly noisy estimation of $\mathcal{D}'$.

5.4.1.2 VGG11

All the networks evaluated thus far and all modern CNN architectures have linear classifiers while AlexNet [44] and VGG11 [71] have non-linear classifiers. Consequently, utilising a CVAE, with significantly lower expressive power than a feature extractor, to estimate the activations of networks such as AlexNet and VGG is challenging. Furthermore, the dimensionality of the input to VGG11's feature extractor is 10,388 neurons, making it challenging to create a CVAE architecture with a reasonable memory footprint to estimate so many neurons. Preliminary results fail to successfully train small VAEs to estimate the activations of VGG11 and further exploration of this limitation is left for future work. However, as discussed in Sec.5.2, there is trend towards modern networks having highly expressive feature extractors that feed into linear classifiers. Hence, the need for exploring methods of adapting the proposed VAE methodology to non-linear classifiers should be evaluated.

	Top1 Test Accurcay (%)				Memory (MB)
	City (185)	Motorway (26)	Countryside (205)	Off-road (26)	
Conditional	73.84	76.75	74.03	87.30	10
Unconditional	73.92	74.92	74.18	88.02	400

Table 5.13: Comparison of retraining accuracy and VAE model memory consumption when using a single CVAE vs 1000 unconditional VAEs. The network is an 80% pruned ResNet50 ($\mathcal{M}^{80}_{TFO}$).

5.4.2 Unconditional vs Conditional VAEs

Instead of training a single Conditional VAE for all 1000 classes of ILSVRC'12, it is possible to train smaller unconditional VAEs per class and depending on the activation desired, the appropriate VAE is sampled. However, this approach increases the memory consumption of the adaptation process as it now requires one VAE per class. This experiment explores if conditioning the VAE on classes causes any loss in representation capabilities and what the trade-off is between the memory consumption savings of having a single VAE versus the increased expressivity obtained by having a VAE per class. The CVAE architecture used is as described in Sec.5.2.1. Each unconditional VAE consumes only 0.4 MB and the architecture used is:

- $|z| = 2$ (dimension of latent space), $s = 1000$ (number of classes in ILSVRC'12)
- The encoder, which estimates $q_\phi(z|\mathcal{A}_s)$, is a three layer fully connected neural network with layer sizes [$|\mathcal{A}_s|$, 128], [128, 64], [64, $|z|$] with ReLU non-linearities.
- The decoder, which estimates $p_\theta(\mathcal{A}_s|z)$, is a two layer fully connected neural network with layer sizes [$|z|$, 64], [64, $|\mathcal{A}_s|$] with ReLU non-linearities.

Table.5.13 shows the results for no label space uncertainty retraining of $\mathcal{M}^{80}_{TFO}$ for various $\mathcal{D}'$ using LoCO-PDA. The training used an SGD optimiser with 5 epochs of retraining, batch size 32, initial learning rate $1e^{-6}$, no learning rate changes, momentum of 0.9 and no weight decay.

The results demonstrate that on average across $\mathcal{D}'$, the unconditional approach performs 0.22pp worse than one CVAE while consuming 40x more memory. Thus, a conditional architecture has significant memory benefits and similar representation capabilities as the unconditional architecture.

5.5 Conclusion

This chapter introduced LoCO-PDA, a methodology to perform low cost on-device domain adaptation of CNNs along with an open-sourced PyTorch implementation of the tool. The proposed pruning and retraining methodology achieves up to 3.15x reduction in model memory footprint (γ) and 2.07x improvement in inference latency (ϕ) while achieving on average 3.04pp improvement in classification accuracy ($\bar{\alpha}$) compared to an unpruned version of the network that is deployed without any domain adaptation. Compared to other image based on-device retraining methodologies, LoCO-PDA achieves up to 3.07x runtime memory consumption (Λ) improvements while also being significantly more robust to noisy estimations of the target domain class distribution. Furthermore, the entire retraining process can be performed on a CPU with sub-minute domain adaptation wall-clock times on a Jetson TX2 device (Θ). LoCO-PDA also performs comparably to other state-of-the-art PDA methodologies while benefiting from the ability to adapt to varying target domains, which the other methodologies cannot.

In doing so, LoCO-PDA opens avenues for on-device domain adaptation, through network pruning and retraining, that have not been explored before. Furthermore, it addresses both the concerns raised in Sec.3.4 by enabling on-device retraining of networks that classify large images and accounting for label uncertainty present on the edge and thus makes a strong case for applicability to real-world scenarios.

However, as the proposed methodology is created for the assumption that the source and target domains vary only in the label space and not the distribution of data within a class, it does not target the full PDA problem. As existing approaches to PDA do not optimise for resource constraint execution scenarios, exploring ways to extend LoCO-PDA to the full PDA scenario could be an interesting future direction of research.

Chapter 6

Conclusion

The thesis began with the goal of exploring methodologies to perform on-device adaptation of the weights (through retraining) and the topology of an image classification CNN network. The goal of doing so was to reduce the inference and training stage latency and memory requirements of the CNN while improving its classification accuracy on the observed data distribution. The entire adaptation process was constrained to be performed completely on-device without any edge server communication. The motivating assumption was that upon deployment of a CNN, the classes observed by the network would be a subset of the classes that were used to train the network. Furthermore the constraint of performing this adaptation completely on an edge device was placed as sending the observed data back to a centralised server for processing raises both data privacy concerns as well as fails in scenarios where there is no reliable edge-server communication. This section summarises the main contributions of the PhD, both from the angle of directly solving this problem as well as new research that have furthered the state-of-the-art in related domains. It also provides interesting future directions of research that could be pursued.

6.1 Summary of contributions

In order to achieve the goal of this PhD, two key challenges needed to be addressed. First, training a CNN is a highly memory intensive process while edge devices tend to have limited memory resources. Furthermore, on-device training is necessary for two purposes. First, in order to be able to improve the network's classification accuracy, the network needs to be retrained on the data observed upon deployment. In order to perform this retraining within the memory budgets of the target edge device, pruning was utilised as a manner of reducing both the network's training memory and latency footprints. State-of-the-art pruning methodologies however require to iteratively prune and retrain a network in order to reduce the loss in classification accuracy as a result of pruning. To solve this problem, Chapter 4 proposes to use a Once-for-all [9] network that enables to rapidly sample from a super-network, small sub-networks that perform well on the original training data distribution while having latency and memory footprints that fit within the device's and application's constraints. These sub-networks can then be retrained on the observed data distribution in order to improve their classification accuracy on this distribution. Thus, the first contribution of Chapter 4 is the identification of a system that removes the need for expensive pruning and retraining to be performed on-device.

However, in order to successfully deploy such a system it was necessary to be able to predict the training stage memory consumption and latency of a sub-network so that appropriate sub-networks, that can be trained within the device's and application's memory and latency budgets, could be selected. Thus, the other novel contribution of Chapter 4 was *perf4sight* - a low-cost, scalable and data-driven CNN training and memory modelling methodology.

The state-of-the-art in memory and latency models for CNN training and inference memory and/or latency prediction resided on two ends of a spectrum. On the one end, they were completely data-driven as with [52, 8], i.e. fitting a regression model on various network and execution related parameters such as batch-size, filter sizes etc. Such methods resulted in reduced accuracy of prediction. On the other end, as with [19], they were completely analytical

(without any data-driven component) where each aspect of a network, execution framework and device were modelled with hand-crafted features. This resulted in much more accurate predictions, but suffered from lack of scalability to new devices, execution frameworks or networks as for each, expert-level knowledge of their execution semantics is necessary.

Combining the best of both worlds, *perf4sight* used analytical modelling for attributes of the inference and training processes that are expected to remain static such as methods of executing convolutions on GPUs while using a data-driven approach to model proprietary and constantly changing attributes such as execution framework specific contributions to memory and latency. By doing so, *perf4sight* achieves training memory and latency prediction error rates of 5.53% and 9.37% on a wide range of networks while furthering the state-of-the-art in training memory prediction accuracy by 15pp. Additionally, the hybrid modelling approach argues strongly for good future scalability to new networks, devices and execution frameworks.

However, the requirement to train the entire network places a hard bound on how much the cost of adapting a network to a new domain can be reduced by. In order to be able to target edge devices even smaller than the Jetson TX2 in the future, methodologies that can either reduce the cost of training or bypass training entirely would need to be explored. Towards this goal, it was observed that modern neural networks have highly expressive feature extractors coupled with linear classifiers. This led to the work described in Chapter 5 which proposes a methodology to completely bypass execution of the feature extractor while only adapting the classifier to the observed data distribution.

That being said, along with reducing the cost of retraining a network, a second important challenge to address was to perform the retraining without access to ground-truth labels of the images observed on an edge device. Existing works in the field of domain adaptation tackle this issue by training, in an unsupervised manner, a separate domain discriminator (learns to differentiate images from the source and target domains) while also jointly training the CNN's feature extractor to reduce the difference between the generated activations for the two domains. Thus the resultant network produces features that are invariant to domain and hence the network's original classifier can provide reliable labels for the new target domain images.

In the context of this thesis, the source domain is the original training dataset while the target domain is the observed data upon deployment whose label space is a subspace of that of the training data. However, existing methodologies are too memory intensive to be performed directly on an edge device due to the requirement for training the feature extractor as well as having access to the entire training dataset in order to do so. Consequently, the work in Chapter 5 proposes a novel methodology that enables retraining the network in an unsupervised manner using unlabelled target domain images.

The proposed methodology relies on two important insights that are also contributions that arise from this PhD. First, in the case that the source and target domains differ only in the label spaces in a manner that the target domain label space is a subspace of that of the source domain; it is sufficient to train just the classifier (while leaving the feature extractor frozen) in order to improve the network's classification accuracy on the new target label space. Note that this assumption has been shown to only hold in the case of networks with linear classifiers. Second, although having uncertain label predictions negatively affects training, the main factor that degrades the performance of training is having incorrectly paired tuples of images and labels. For instance, retraining a network with an image of cat that is labelled as a dog causes significant degradation to the resulting classification accuracy of the network, even if it is only the classifier that is being retrained.

Combining these two insights, the work in Chapter 5, proposes a methodology that uses Variational Auto Encoders (VAE) to generate pre-classifier activations of a network, conditioned on a desired class. As the generated activations correctly correspond to the class label for which they were generated, these activations coupled with their labels, can then used to retrain the network's classifier in order to improve classification accuracy on the target domain. By bypassing even the inference stage of the network during training, this strategy, when executed on a CPU, reduces the latency of training by up to 3.5x compared to classifier-only retraining of a network on a GPU. Here, classifier-only retraining refers to retraining by first performing inference through the feature extractor followed by training of the classifier only.

With this strategy, inference on the originally deployed network (trained only on the source

domain) is utilised to generate a distribution of expected classes in the target domain which is provided as input to the VAE to generate activations. This process makes use of the second insight mentioned where even if the predictions of which classes are observed are uncertain (as they are generated from the original deployed network's classification), as long as the generated activation corresponds to the label provided, these activations can be used to train the deployed network to improve classification accuracy. By doing so, the PhD proposes a second novel methodology for on-device domain adaptation that is low-cost enough to be run directly on a CPU while also accounting for the uncertainty in the labels of the target domain images.

Additionally, work done during the feasibility study carried out in Chapter 3 resulted in a final contribution, *Autoprune* - an open-source tool that automates both the process of identifying dependencies between filters across layers in a CNN and pruning the network in a manner that respects these dependencies. The automation of both dependency identification as well as the pruning of the network itself are significant improvements over existing tools that require considerable manual specification of which filters to prune. In doing so, *Autoprune* enables faster and automated prototyping and deployment of pruning and retraining methodologies that can target a wide range of network connectivity patterns.

6.2 Future Works

Having summarised the contributions of this PhD, this section identifies three future directions of research that stem from the work done here.

First, an immediate extension is to develop a method to combine the methodologies proposed in Chapters 4 and 5. The work in Chapter 4 has its strengths in being able to expose an immense ($> 50,000$) number of topologies to be chosen from in a manner that is completely on-device. However, the proposed methodology here assumes access to ground-truth labels of the images observed in the target domain. The work in Chapter 5 has its strength in being able to cater for this label uncertainty, however as a new VAE needs to be retrained per network that is being adapted, limits the number of topologies that can be chosen from on-device. Combining

the two works would raise the interesting research questions on how to train VAEs that are not just conditioned on the label of the desired activation, but also on the specific sub-network sampled from the OFA super-network. Additionally, the work would also raise the interesting question of when having access to such as a large number of topologies as 50,000 is useful in an edge-device setting.

Another interesting line of research would be to explore expanding the use case of the proposed methodologies to object detection framework rather than image classification ones. The same problem setting and assumption would hold, but it would raise interesting questions on if a LoCO-PDA style approach could be integrated into either multi-stage object detectors such as [22] or single-stage ones such as [21].

Finally, consider the scenario of a translation natural language processing model running on a mobile device. Although trained on a dataset with a wide range of languages (and accents within each language) once deployed in a specific country, the model now would be required to translate far more specific nuances in accents within a much smaller set of languages. Sending details of conversations back to a server, although this happens today, again raises data privacy constraints. Additionally, the network a user is on could have poor connectivity and cannot communicate reliably with a server, in which case again model adaptation is restricted. Consequently, exploring a similar problem setting, but from the perspective of natural language models would also be an interesting future direction of research.

Bibliography

[1] Ecovacs robotics: the ai robotic vacuum cleaner powered by tensorflow. https://blog.tensorflow.org/2020/01/ecovacs-robotics-ai-robotic-vacuum.html. 2020.

[2] A. Alqahtani, X. Xie, M. W. Jones, and E. Essa. Pruning cnn filters via quantifying the importance of deep visual representations. *Computer Vision and Image Understanding*, 208-209:103220, 2021.

[3] M. Bansal, A. Krizhevsky, and A. S. Ogale. Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst. *CoRR*, abs/1812.03079, 2018.

[4] 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*, CCS '17, page 1175–1191, New York, NY, USA, 2017. Association for Computing Machinery.

[5] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Józefowicz, and S. Bengio. Generating sentences from a continuous space. *CoRR*, abs/1511.06349, 2015.

[6] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[7] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and

D. Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[8] E. Cai, D.-C. Juan, D. Stamoulis, and D. Marculescu. Neuralpower: Predict and deploy energy-efficient convolutional neural networks. *arXiv preprint arXiv:1710.05420*, 2017.

[9] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once for all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020.

[10] H. Cai, C. Gan, L. Zhu, and S. Han. Tiny transfer learning: Towards memory-efficient on-device learning. *CoRR*, abs/2007.11622, 2020.

[11] Z. Cao, L. Ma, M. Long, and J. Wang. Partial adversarial domain adaptation. *CoRR*, abs/1808.04205, 2018.

[12] Z. Cao, K. You, M. Long, J. Wang, and Q. Yang. Learning to transfer examples for partial domain adaptation. *CoRR*, abs/1903.12230, 2019.

[13] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, pages 262–263, 2016.

[14] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[15] M. Cho, A. Joshi, and C. Hegde. ESPN: extremely sparse pruned networks. *CoRR*, abs/2006.15741, 2020.

[16] F. Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1800–1807, Honolulu, HI, July 2017. IEEE.

[17] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[18] C. Doersch. Tutorial on variational autoencoders. *CoRR*, abs/1606.05908, 2016.

[19] Y. Gao, Y. Liu, H. Zhang, Z. Li, Y. Zhu, H. Lin, and M. Yang. Estimating gpu memory consumption of deep learning models. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1342–1352, New York, NY, USA, 2020. Association for Computing Machinery.

[20] H. Gholamalinezhad and H. Khosravi. Pooling methods in deep neural networks, a review. *CoRR*, abs/2009.07485, 2020.

[21] R. Girshick. Fast r-cnn. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448, 2015.

[22] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014.

[23] B. Gong, Y. Shi, F. Sha, and K. Grauman. Geodesic flow kernel for unsupervised domain adaptation. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2066–2073, 2012.

[24] G. Griffin, A. Holub, and P. Perona. Caltech 256. *California Institute of Technology*, 2007.

[25] K. Guo, L. Sui, J. Qiu, S. Yao, S. Han, Y. Wang, and H. Yang. From model to FPGA: Software-hardware co-design for efficient neural network acceleration. In *2016 IEEE Hot Chips 28 Symposium (HCS)*, pages 1–27, Aug. 2016. ISSN: null.

[26] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. *Advances in Neural Information Processing Systems*, 2015-January:1135–1143, 6 2015.

[27] C. He, M. Annavaram, and S. Avestimehr. Group knowledge transfer: Collaborative training of large cnns on the edge. *CoRR*, abs/2007.14513, 2020.

[28] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016-December:770–778, 12 2016.

[29] B. Heintz, A. Chandra, and R. K. Sitaraman. Optimizing grouped aggregation in geo-distributed streaming analytics. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 133–144, New York, NY, USA, 2015. Association for Computing Machinery.

[30] I. Higgins, L. Matthey, A. Pal, C. P. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. In *ICLR*, 2017.

[31] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. Densely Connected Convolutional Networks. *arXiv:1608.06993 [cs]*, Jan. 2018. arXiv: 1608.06993.

[32] Q. Huang. Weight-quantized squeezenet for resource-constrained robot vacuums for indoor obstacle classification. *AI*, 3(1):180–193, 2022.

[33] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv:1602.07360 [cs]*, Nov. 2016. arXiv: 1602.07360.

[34] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[35] M. Jorda, P. Valero-Lara, and A. J. Pena. Performance evaluation of cudnn convolution algorithms on nvidia volta gpus. *IEEE Access*, 7:70461–70473, 2019.

[36] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland,

R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[37] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, July 2021.

[38] P. KANG and J. JO. Benchmarking modern edge devices for ai applications. *IEICE Transactions on Information and Systems*, E104.D:394–403, 03 2021.

[39] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[40] D. P. Kingma and M. Welling. Auto-encoding variational bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[41] A. Kouris and C.-S. Bouganis. Learning to Fly by MySelf: A Self-Supervised CNN-Based Approach for Autonomous Navigation. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–9, Madrid, Oct. 2018. IEEE.

[42] A. Kouris, C. Kyrkou, and C.-S. Bouganis. Informed Region Selection for Efficient UAV-based Object Detectors: Altitude-aware Vehicle Detection with CyCAR Dataset. In *2019*

IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 51–58, Nov. 2019. ISSN: 2153-0858.

[43] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. Technical Report 0, University of Toronto, Toronto, Ontario, 2009.

[44] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[45] A. Lavin and S. Gray. Fast algorithms for convolutional neural networks. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016.

[46] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov. 1998. Conference Name: Proceedings of the IEEE.

[47] I. Leontiadis, S. Laskaridis, S. I. Venieris, and N. D. Lane. It's always personal: Using early exits for efficient on-device cnn personalisation. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, HotMobile '21, page 15–21, New York, NY, USA, 2021. Association for Computing Machinery.

[48] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 8 2016.

[49] M. Lin, Q. Chen, and S. Yan. Network in network. *CoRR*, abs/1312.4400, 2014.

[50] S. Lin, R. Ji, Y. Li, C. Deng, and X. Li. Towards Compact ConvNets via Structure-Sparsity Regularized Filter Pruning. *arXiv:1901.07827 [cs]*, Mar. 2019. arXiv: 1901.07827.

[51] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1411.4038, 2014.

[52] Z. Lu, S. Rallapalli, K. Chan, S. Pu, and T. L. Porta. Augur: Modeling the resource requirements of convnets on mobile devices. *IEEE Transactions on Mobile Computing*, 20(2):352–365, 2021.

[53] J. Luo and J. Wu. Neural network pruning with residual-connections and limited-data. *CoRR*, abs/1911.08114, 2019.

[54] J.-H. Luo, J. Wu, and W. Lin. Thinet: A filter level pruning method for deep neural network compression. *Proceedings of the IEEE International Conference on Computer Vision*, 2017-October:5068–5076, 7 2017.

[55] M. Mathieu, M. Henaff, and Y. LeCun. Fast training of convolutional networks through ffts. *CoRR*, abs/1312.5851, 2014.

[56] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. M. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi, J. Pak, A. Tong, K. Srinivasa, W. Hang, E. Tuncer, Q. V. Le, J. Laudon, R. Ho, R. Carpenter, and J. Dean. A graph placement methodology for fast chip design. *Nature*, 594 7862:207–212, 2021.

[57] D. Mittal, S. Bhardwaj, M. M. Khapra, and B. Ravindran. Recovering from Random Pruning: On the Plasticity of Deep Convolutional Neural Networks. *arXiv:1801.10447 [cs]*, Jan. 2018. arXiv: 1801.10447.

[58] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz. Importance estimation for neural network pruning. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 11264–11272. Computer Vision Foundation / IEEE, 2019.

[59] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[60] A. Rajagopal and C.-S. Bouganis. Now that i can see, i can improve: Enabling data-driven finetuning of cnns on the edge. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020.

[61] A. Rajagopal and C.-S. Bouganis. perf4sight: A toolflow to model cnn training performance on edge gpus. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, pages 963–971, October 2021.

[62] A. Rajagopal and C.-S. Bouganis. Low-cost on-device partial domain adaptation (loco-pda): Enabling efficient cnn retraining on edge devices. arXiv, 2022.

[63] A. Rajagopal, D. Vink, S. Venieris, and C.-S. Bouganis. Multi-precision policy enforced training (MuPPET) : A precision-switching strategy for quantised fixed-point training of CNNs. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 7943–7952. PMLR, 13–18 Jul 2020.

[64] H. Ren, D. Anicic, and T. A. Runkler. Tinyol: Tinyml with online-learning on microcontrollers. *CoRR*, abs/2103.08295, 2021.

[65] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[66] K. Saenko, B. Kulis, M. Fritz, and T. Darrell. Adapting visual category models to new domains. In K. Daniilidis, P. Maragos, and N. Paragios, editors, *Computer Vision – ECCV 2010*, pages 213–226, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[67] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 1 2018.

[68] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *CoRR*, abs/1312.6229, 2013.

[69] N. Shazeer, K. Fatahalian, W. R. Mark, and R. T. Mullapudi. Hydranets: Specialized dynamic architectures for efficient inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8080–8089, 2018.

[70] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[71] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[72] V. Smith, C.-K. Chiang, M. Sanjabi, and A. S. Talwalkar. Federated multi-task learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[73] K. Sohn, H. Lee, and X. Yan. Learning structured output representation using deep conditional generative models. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[74] X. Suau, L. Zappella, and N. Apostoloff. Filter Distillation for Network Compression. *arXiv:1807.10585 [cs]*, Dec. 2019. arXiv: 1807.10585.

[75] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going Deeper with Convolutions. *arXiv:1409.4842 [cs]*, Sept. 2014. arXiv: 1409.4842.

[76] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[77] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016.

[78] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2019-June:2815–2823, 7 2018.

[79] M. Tan and Q. Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 09–15 Jun 2019.

[80] H. Tanaka, D. Kunin, D. L. K. Yamins, and S. Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[81] D. A. Vink, A. Rajagopal, S. I. Venieris, and C.-S. Bouganis. Caffe barista: Brewing caffe with fpgas in the training loop. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 317–322, 2020.

[82] S. Winograd. *Arithmetic Complexity of Computations*. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1980.

[83] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, Oct. 2020. Association for Computational Linguistics.

[84] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated Residual Transformations for Deep Neural Networks. *arXiv:1611.05431 [cs]*, Apr. 2017. arXiv: 1611.05431.

[85] B. Xu, N. Wang, T. Chen, and M. Li. Empirical evaluation of rectified activations in convolutional network. *CoRR*, abs/1505.00853, 2015.

[86] R. Yu, A. Li, C.-F. Chen, J.-H. Lai, V. I. Morariu, X. Han, M. Gao, C.-Y. Lin, and L. S. Davis. Nisp: Pruning networks using neuron importance score propagation. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 9194–9203, 2017.

[87] J. Zhang, Z. Ding, W. Li, and P. Ogunbona. Importance weighted adversarial nets for partial domain adaptation. *CoRR*, abs/1803.09210, 2018.

[88] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct. 2016. ISSN: null.

[89] C. Zhao, B. Ni, J. Zhang, Q. Zhao, W. Zhang, and Q. Tian. Variational convolutional neural network pruning. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2775–2784, 2019.

[90] Y. Zhao, X. Gao, R. Mullins, and C. Xu. Mayo: A Framework for Auto-generating Hardware Friendly Deep Neural Networks. In *Proceedings of the 2nd International Workshop on Embedded and Mobile Deep Learning - EMDL'18*, pages 25–30, Munich, Germany, 2018. ACM Press.

[91] 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*, 107:1738–1762, 2019.

[92] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik. Neural Network Distiller: A Python Package For DNN Compression Research. *arXiv:1910.12232 [cs, stat]*, Oct. 2019. arXiv: 1910.12232.

Appendix A

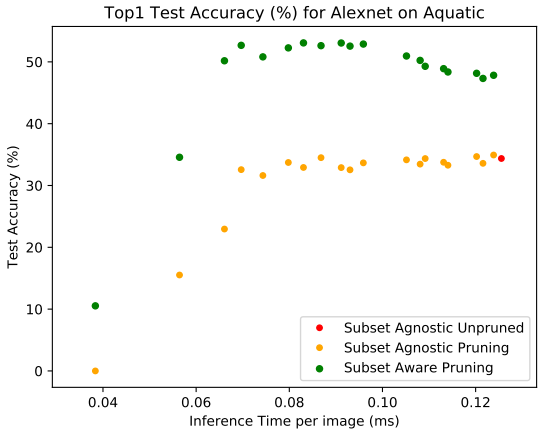
A.1 Results for all combinations of network and subset (Chapter 3)

A.1.1 Accuracy vs. Inference Time trade-off

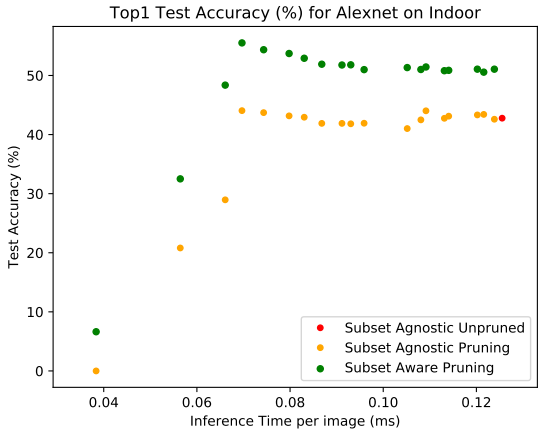
Figs.A.1-A.4 provide results for all combinations of network and subset for the trade-off between achieved classification accuracy and inference time for various pruning levels.

A.1.2 Search time per minibatch vs. Inference Time trade-off

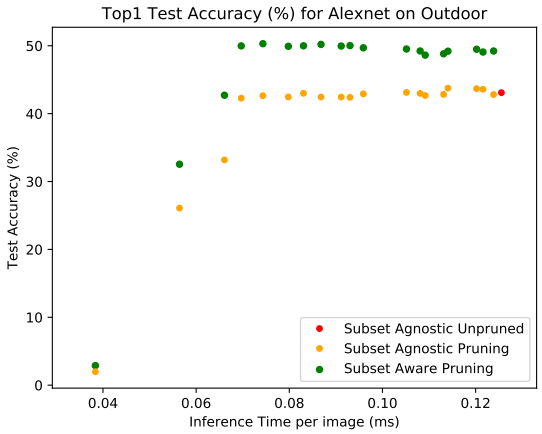
Figs.A.5-A.8 provide results for all combinations of network and subset for the trade-off between the time spent searching for a model with lower compute footprint and the achieved inference time.



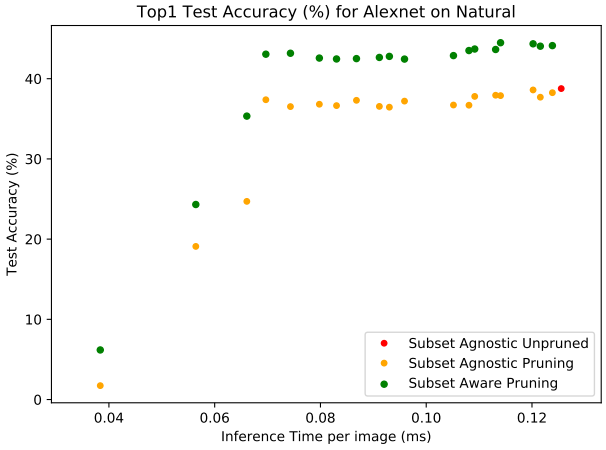
(a) AlexNet - Aquatic



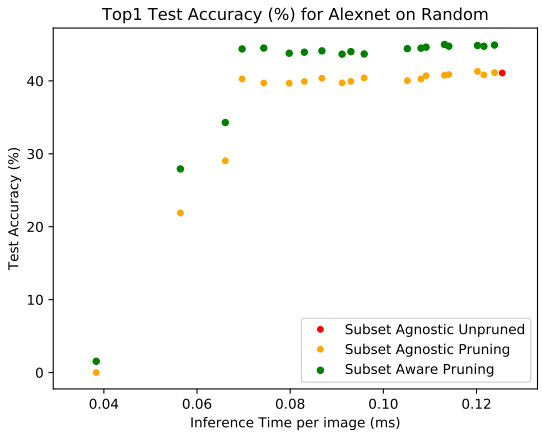
(b) AlexNet - Indoor



(c) AlexNet - Outdoor

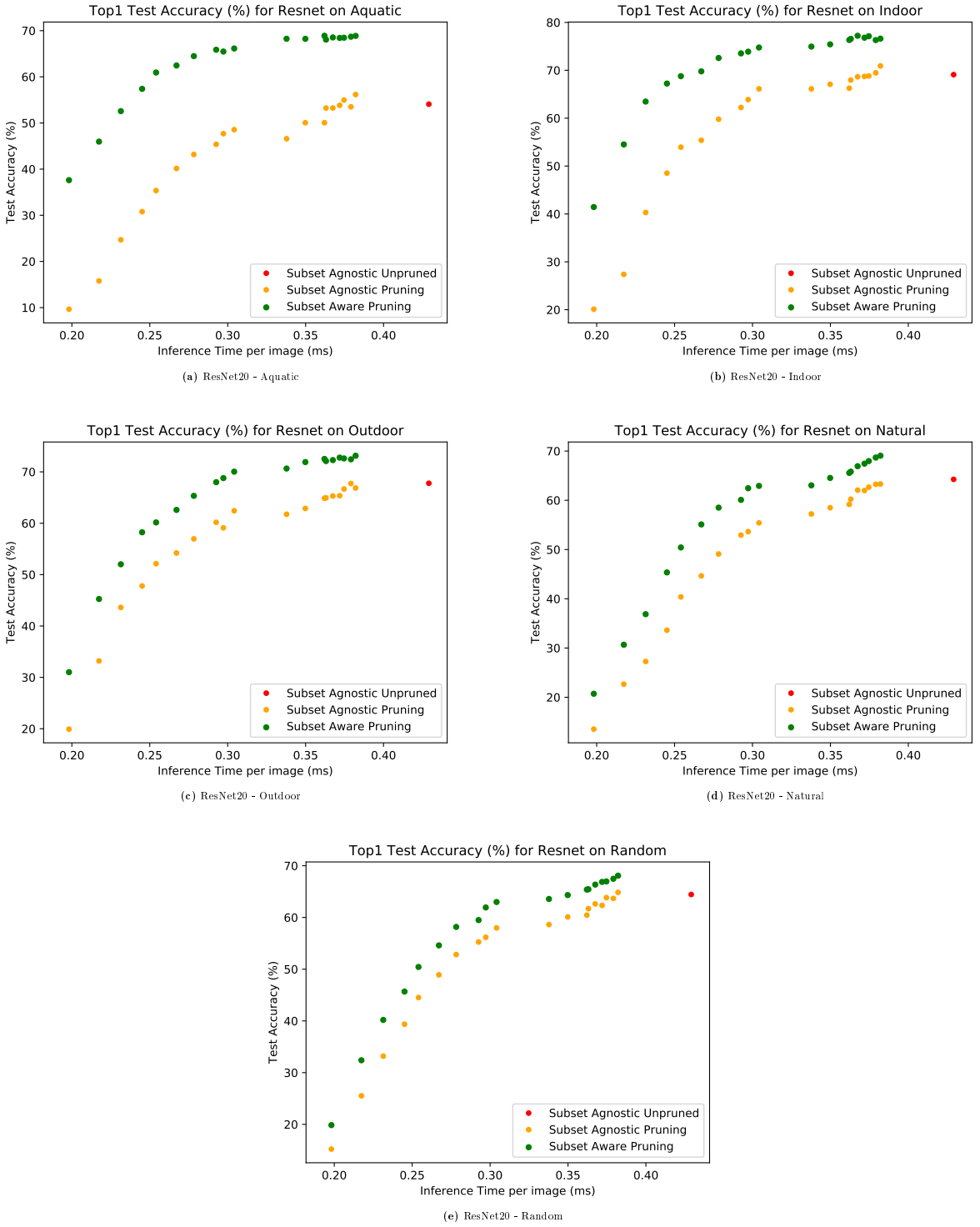


(d) AlexNet - Natural



(e) AlexNet - Random

Figure A.1: Trade-off of Test Accuracy vs Inference Time for various levels of pruning for AlexNet for all subsets.



(a) ResNet20 - Aquatic

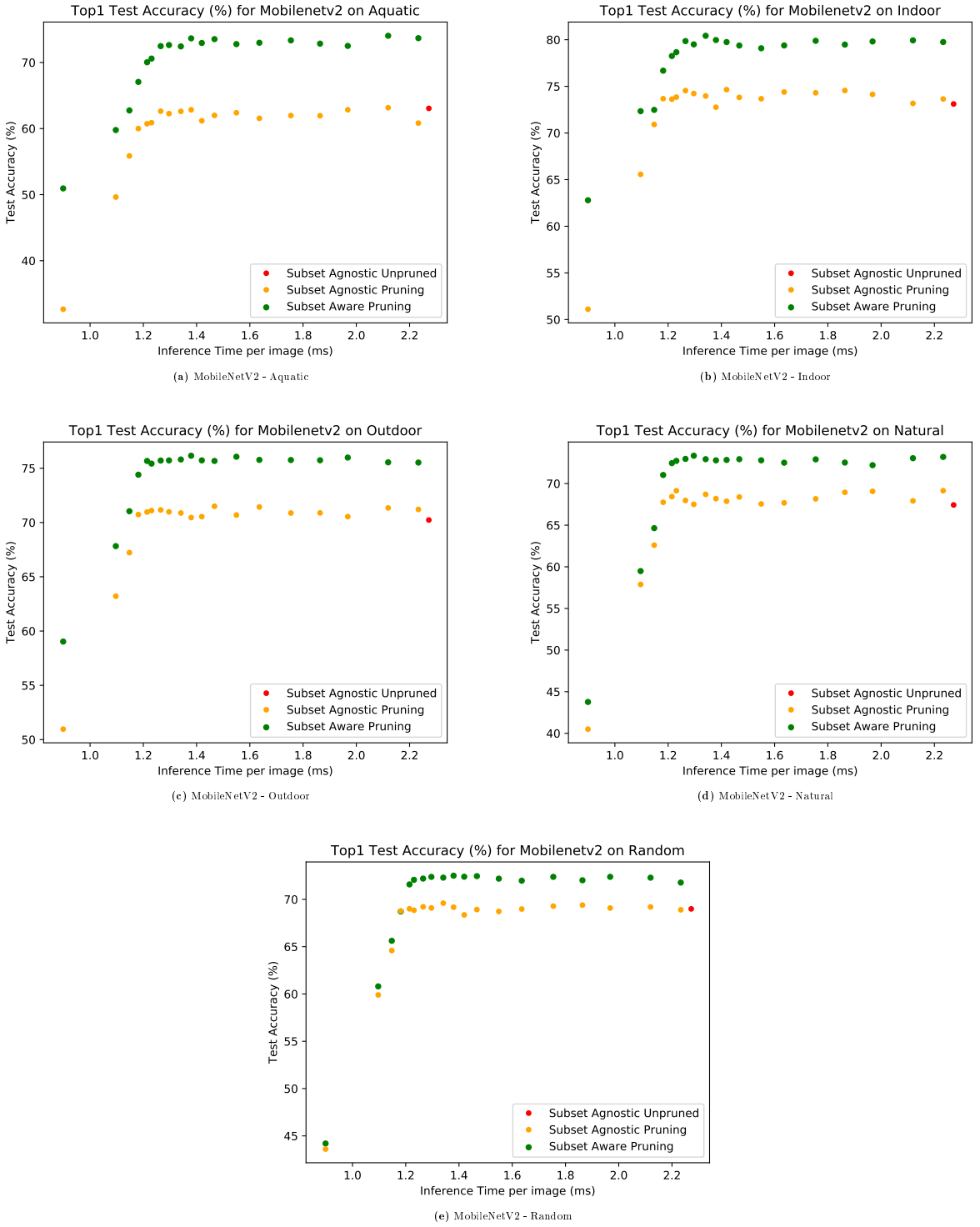
(b) ResNet20 - Indoor

(c) ResNet20 - Outdoor

(d) ResNet20 - Natural

(e) ResNet20 - Random

Figure A.2: Trade-off of Test Accuracy vs Inference Time for various levels of pruning for ResNet20 for all subsets.



(a) MobileNetV2 - Aquatic

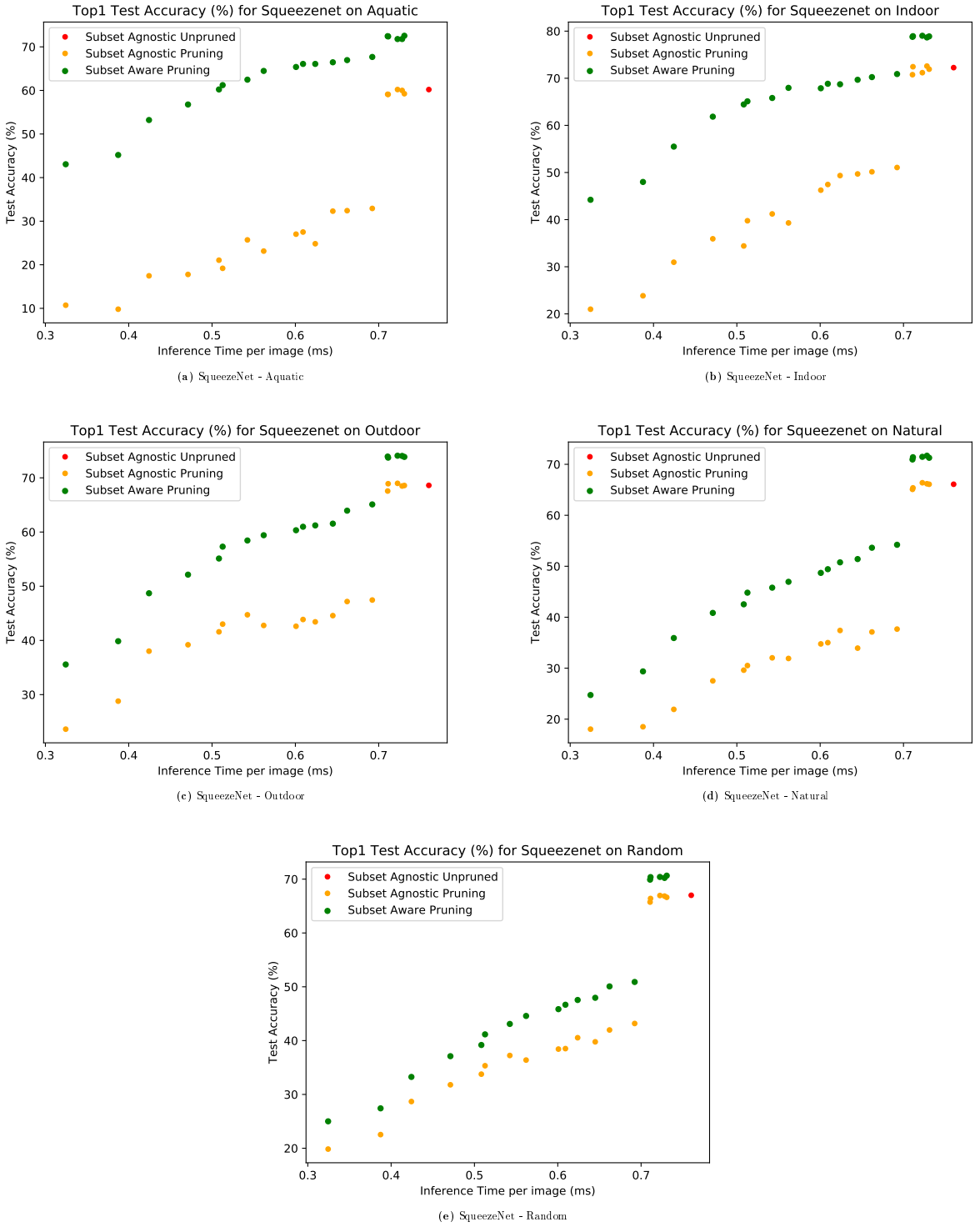
(b) MobileNetV2 - Indoor

(c) MobileNetV2 - Outdoor

(d) MobileNetV2 - Natural

(e) MobileNetV2 - Random

Figure A.3: Trade-off of Test Accuracy vs Inference Time for various levels of pruning for MobileNetV2 for all subsets.



(a) SqueezeNet - Aquatic

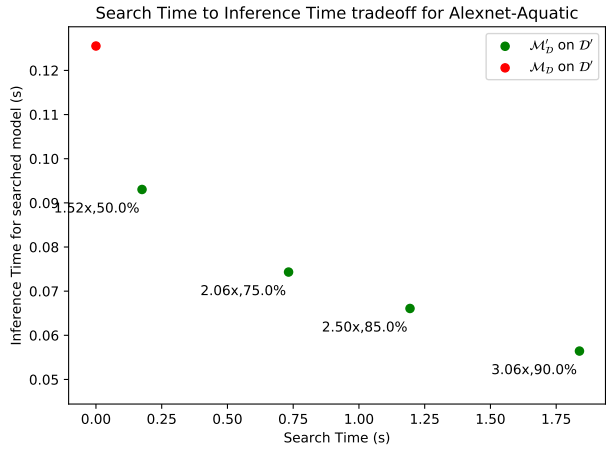
(b) SqueezeNet - Indoor

(c) SqueezeNet - Outdoor

(d) SqueezeNet - Natural

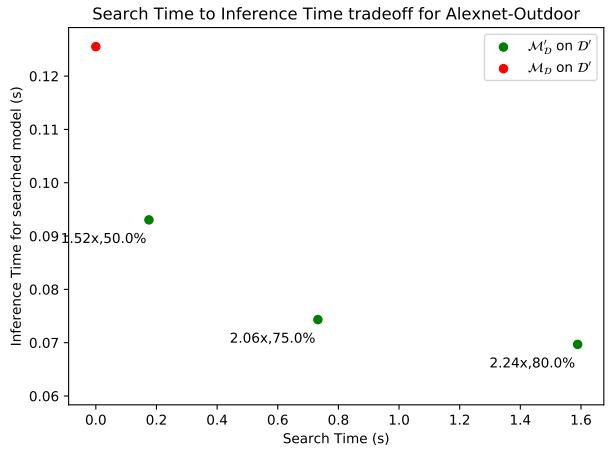
(e) SqueezeNet - Random

Figure A.4: Trade-off of Test Accuracy vs Inference Time for various levels of pruning for SqueezeNet for all subsets.

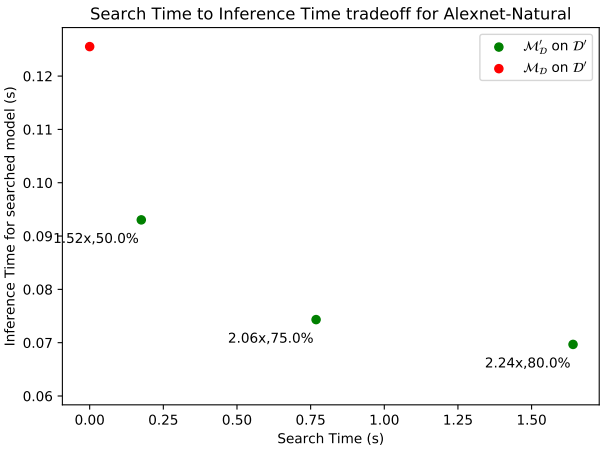


(a) AlexNet - Aquatic

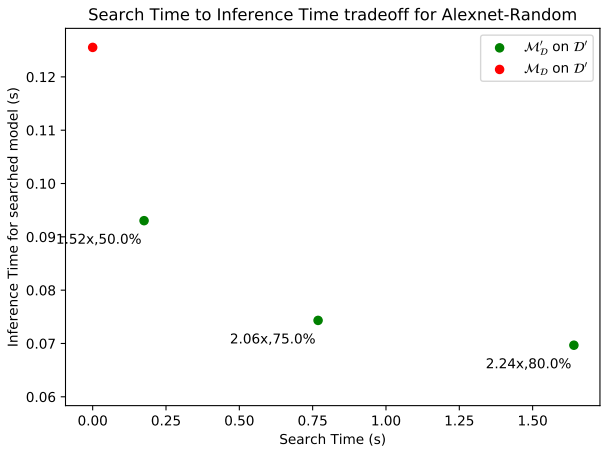
(b) AlexNet - Indoor



(c) AlexNet - Outdoor

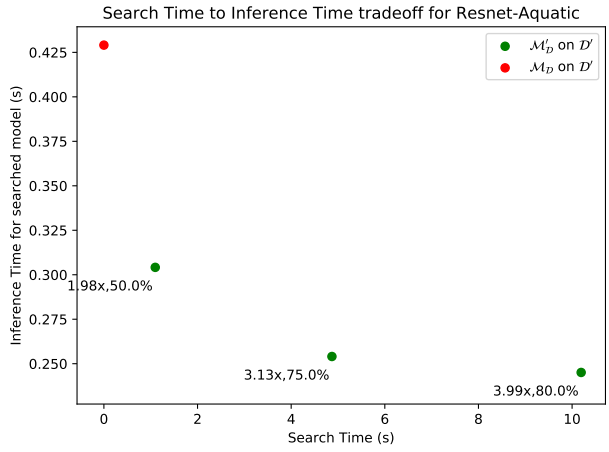


(d) AlexNet - Natural

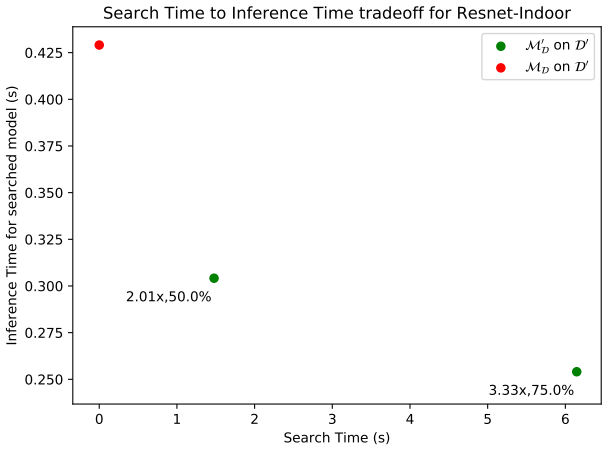


(e) AlexNet - Random

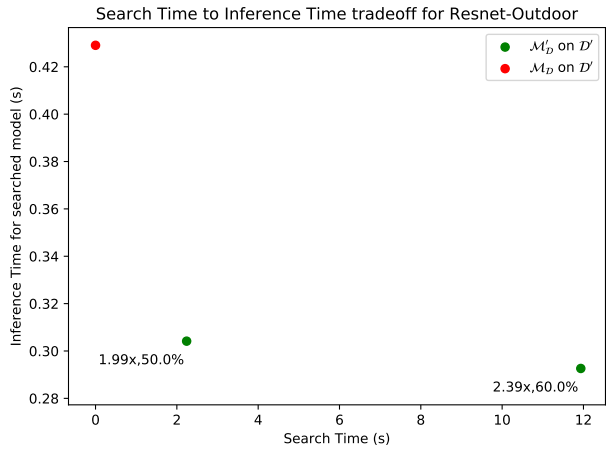
Figure A.5: Trade-off between search time per minibatch and inference time performance of searched model for AlexNet on all subsets. All models shown here have no accuracy loss compared to $\mathcal{M}_{\mathcal{D}}$ infered on $\mathcal{D}'$. The labels next to the points show (improvement in GOps (-x times), pruning level (%)) of that model



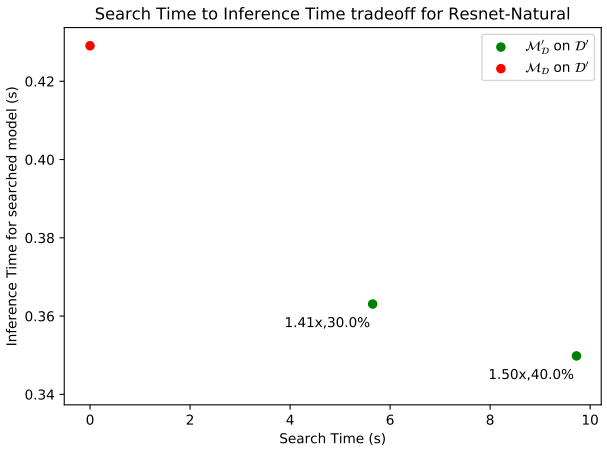
(a) ResNet20 - Aquatic



(b) ResNet20 - Indoor



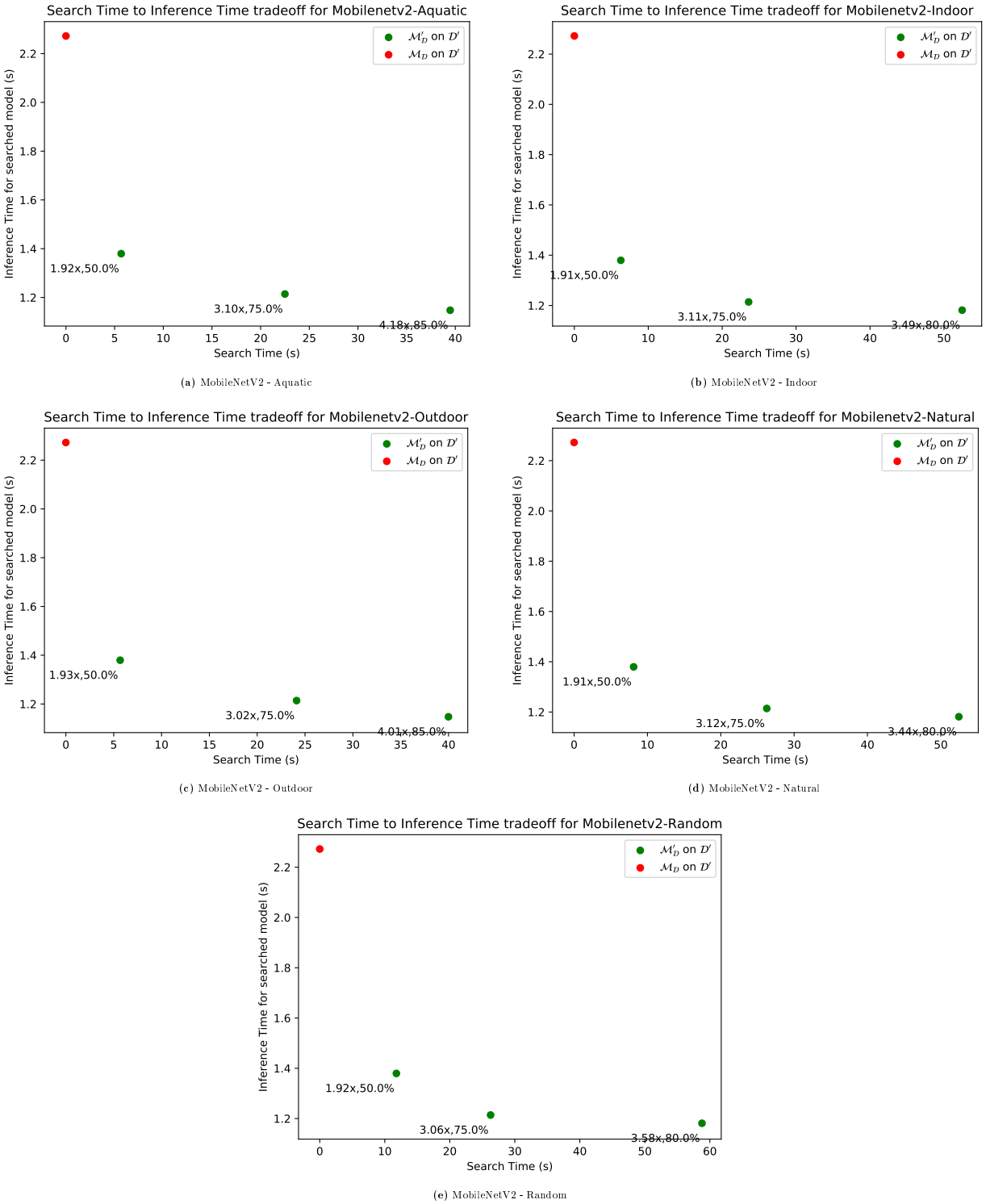
(c) ResNet20 - Outdoor



(d) ResNet20 - Natural

(e) ResNet20 - Random

Figure A.6: Trade-off between search time per minibatch and inference time performance of searched model for ResNet20 on all subsets. All models shown here have no accuracy loss compared to $\mathcal{M}_{\mathcal{D}}$ infered on $\mathcal{D}'$. The labels next to the points show (improvement in GOps (-x times), pruning level (%)) of that model



(a) MobileNetV2 - Aquatic

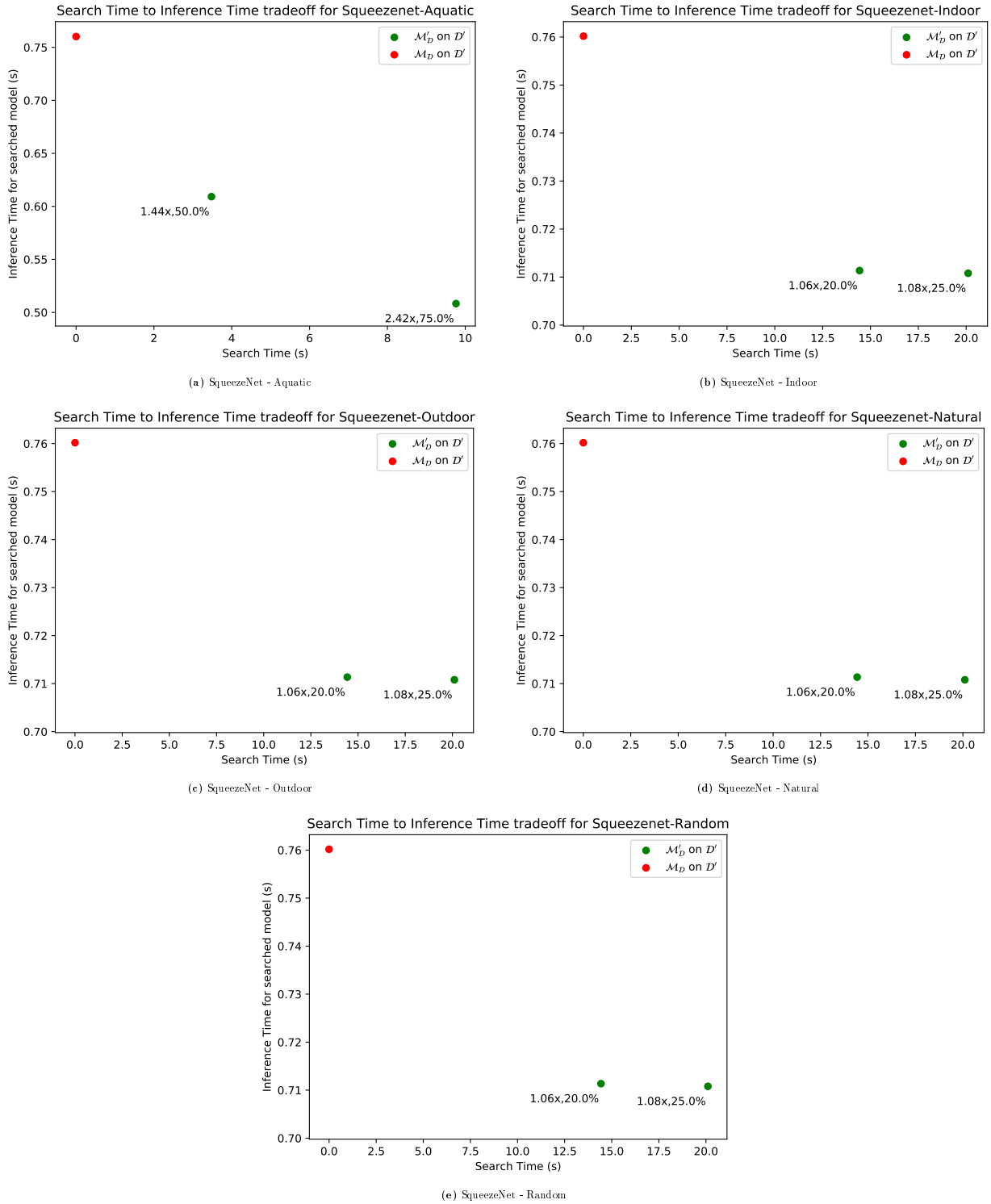
(b) MobileNetV2 - Indoor

(c) MobileNetV2 - Outdoor

(d) MobileNetV2 - Natural

(e) MobileNetV2 - Random

Figure A.7: Trade-off between search time per minibatch and inference time performance of searched model for MobileNetV2 on all subsets. All models shown here have no accuracy loss compared to $\mathcal{M}_{\mathcal{D}}$ infered on $\mathcal{D}'$. The labels next to the points show (improvement in GOps (-x times), pruning level (%)) of that model



(a) SqueezeNet - Aquatic

(b) SqueezeNet - Indoor

(c) SqueezeNet - Outdoor

(d) SqueezeNet - Natural

(e) SqueezeNet - Random

Figure A.8: Trade-off between search time per minibatch and inference time performance of searched model for SqueezeNet on all subsets. All models shown here have no accuracy loss compared to $\mathcal{M}_{\mathcal{D}}$ infered on $\mathcal{D}'$. The labels next to the points show (improvement in GOps (-x times), pruning level (%)) of that model

Appendix B

B.1 Profiling in PyTorch

This section details the practical implementation of data collection for the attributes described in Sec.4.1. PyTorch forward and backward hooks are attached to the network's convolution layers and at each hook the following values are recorded for pruning levels $5x|x \in [0, 18]$ and batch sizes {2,4,8,16,32,64,70,80,90,100,110,120,
128,140,150,160,170,180,190,200,210,220,230,240,256}. Results provided are for both the NVIDIA Jetson TX2 and the NVIDIA RTX 2080Ti devices.

Mini-batch training latency (Φ) is profiled in the same manner for the two systems (embedded and server) using the `torch.cuda.Events` API as this allows to time asynchronous GPU events.

Memory Used (Γ) is profiled differently for embedded (NVIDIA Jetson TX2) and server (NVIDIA RTX 2080Ti) systems due to the difference in memory management where the embedded device has a unified memory space, where the server system has an independent memory space for the GPU.

- **NVIDIA Jetson TX2** - Profiled using information from the `/proc/meminfo` system file according to the equation `TotalMemUsed = MemTotal - MemFree - Buffers - Cached`. The terms

on the RHS of the equation correspond directly to values found in `/proc/meminfo`. As the Jetson has a unified memory, information in this file accounts for both GPU and CPU memory.

- **NVIDIA RTX 2080Ti** - Profiled using the `pynvml` tool with the command `nvmlDeviceGetMemoryInfo.used`.

The system records the maximum value of Γ observed until the point of profiling.