Imperial College London
Department of Computing

Improving the Performance of Dataflow Systems for Deep Neural Network Training

Pijika Watcharapichat

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

January 2018

Abstract

Deep neural networks (DNNs) have led to significant advancements in machine learning. With deep structure and flexible model parameterisation, they exhibit state-of-the-art accuracies for many complex tasks e.g. image recognition. To achieve this, models are trained iteratively over large datasets. This process involves expensive matrix operations, making it time-consuming to obtain converged models. To accelerate training, dataflow systems parallelise computation. A scalable approach is to use parameter server framework: it has workers that train model replicas in parallel and parameter servers that synchronise the replicas to ensure the convergence.

With distributed DNN systems, there are three challenges that determine the training completion time. In this thesis, we propose practical and effective techniques to address each of these challenges.

Since frequent *model synchronisation* results in high network utilisation, the parameter server approach can suffer from network bottlenecks, thus requiring decisions on resource allocation. Our idea is to use all available network bandwidth and synchronise subject to the available bandwidth. We present Ako, a DNN system that uses *partial gradient exchange* for synchronising replicas in a peer-to-peer fashion. We show that our technique exhibits a 25% lower convergence time than a hand-tuned parameter-server deployments.

For a long training, the *compute efficiency* of worker nodes is important. We argue that processing hardware should be fully utilised for the best speed-up. The key observation is it is possible to overlap the execution of several matrix operations with other workloads. We describe Crossbow, a GPU-based system that maximises hardware utilisation. By using a multi-streaming scheduler, multiple models are trained in parallel on GPU and achieve a 2.3x speed-up compared to a state-of-the-art system.

The *choice of model configuration* for replicas also directly determines convergence quality. Dataflow systems are used for exploring the promising configurations but provide little support for efficient exploratory workflows. We present Meta-dataflow (MDF), a dataflow model that expresses complex workflows. By taking into account all configurations as a unified workflow, MDFs efficiently reduce time spent on configuration exploration.

This thesis is dedicated to *my mum.*

Acknowledgements

First of all, I would like to express my gratitude to my supervisor, Professor Peter Pietzuch, for his great support throughout my PhD training. I am very thankful for his continuous guidance, invaluable suggestion, constructive criticism, and patience. I have been lucky to have Peter as my supervisor because he sets an excellent example for me in a number of aspects.

Secondly, I would like to thank Alexandros Koliousis. Advice and feedbacks from him have been invaluable to me. I have learnt several skills and feel genuinely thankful for his kind support.

I want to thank Victoria and Raul for our continuous discussion during project Ako. It has been a great honour for me to work with talented researchers.

I am grateful to all my friends in the LSDS group; Victoria, Divya, Lukas, Christian, Domagoj, PL, Panagiotis, Matthias, Paolo, Andreas, William, Dan, Florian, Josh, and Luo. I had a great time in 346 Huxley. I want to thank them for keeping me company and giving me all kinds of support. I also would like to thank Nicolai, Fergus, Clint, and Josh (ML).

My PhD training would not be possible without the Department of Computing (DoC) who provided me a generous funding from the start to the end. I truly appreciate the opportunity that DoC kindly gave me to pursue my PhD. I also want to thank Peter, who supported me during my write-up period.

I also want to thank Dr. Amani El-Kholy and Professor Sophia Drossopoulou, who gave me a warm-hearted support during my study.

Finally, I would like to thank Bo, my family (the Watcharapichats and Kositanants), Pa (Phattarapongsant), Khun Ar, and Go Mui for their tremendous support, constant encouragement, and belief in me. It would be impossible for me to have become who I am today without them.

Declaration

This thesis presents my work in the Department of Computing at Imperial College London between October 2013 and September 2017.

Parts of the work were done in collaboration with other researchers. All the work presented here in this thesis were supervised by Professor Peter Pietzuch.

Chapter 3. I developed all the codes and applications for the project and conducted all the experiments. The idea of this work was the result of discussions between myself, Victoria Lopez Morales, and Raul Castro Fernandez.

Chapter 4. I developed a number of DNN and machine learning applications and conducted all the experiments presented in the thesis. Regarding the implementation effort, I was responsible for building a number of operators that execute on CPU, some operators that execute on GPU, data preprocessing, machine learning applications, and the memory management algorithm. I also performed comparative studies on a number of applications across different systems. This is collaborative work led by Alexandros Koliousis. Several ideas and plans originated from the discussions between myself, Alexandros Koliousis, Paolo Costa, and Matthias Weidlich.

Chapter 5. I developed the DNN applications that perform hyperparameter explorations and conducted all the experiments shown in the thesis. Also, I performed the comparative study across different systems. I worked in collaboration with Raul Castro Fernandez, William Culhane, Matthias Weidlich, and Victoria Lopez Morales.

Publication

Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised Deep Learning with Partial Gradient Exchange. In *ACM Symposium on Cloud Computing 2016 (SoCC), Santa Clara, CA, USA*.

Raul Castro Fernandez, William Culhane, Pijika Watcharapichat, Matthias Weidlich, Victoria Lopez Morales, and Peter Pietzuch. Meta-Dataflows: Efficient Exploratory Dataflow Jobs. In *ACM Conference on Management of Data 2018 (SIGMOD), Houston, TX, USA*.

Contents

List of figures

Chapter 1

Introduction

1.1 Deep neural networks

Deep neural networks (DNNs) have brought about significant breakthroughs in pattern recognition applications over the recent years. We have witnessed tremendous improvements in recognition performance in a wide variety of domains. For visual recognition tasks, DNNs have demonstrated recognition accuracy that is comparable to human level for digit recognition [LBBH98] and face recognition [TYRW14]. In the ImageNet challenge, DNNs have set a series of new records surpassing most other machine learning techniques. Furthermore, they have been successful in performing phonetic classification for automatic speech recognition [HDY+12], which leads to a major step in applying machine learning in industry. Today, DNNs have been applied in a vast range of domains such as finance, healthcare, drug discovery, natural science, climate research, and engineering.

What makes DNNs powerful is the ability to learn data representation for the raw data. DNNs are a machine learning technique that automatically discovers data representations and extracts features necessary for recognition. More specifically, DNNs learn multiple layers of data representation corresponding to different levels of abstraction and concept. For image classification tasks, representations are learned directly from image pixels and higher-level features are then developed: a set of edges arranged into shapes at certain regions and eventually a meaningful object is inferred. This ability to obtain high-level representations from lower level ones is made possible by the internal structure of a DNN model, which is composed of multiple data processing units organised in layers.

Several studies have shown that the depth of representation layers is one of the key factors that enable a DNN to achieve high model performance for non-trivial recognition problems [SZ14, SLJ+15, HZRS15a]. For the ImageNet challenge [RDS+15a], the first place winner in 2015 used a DNN with the depth of up to 152 layers [HZRS15a]. Another factor is that the data representations in DNN layers are not handcrafted feature extractors but rather learned automatically from raw data by using a generic learning algorithm.

Like other machine learning models, data representations are learned through a training process during which a learning algorithm is used to adjust the values of *learnable model parameters*. In DNNs, the connection weights between the processing layers are the model parameters. A common learning algorithm is the *backpropagation* algorithm [RHW86] which is based on gradient descent optimisation: a model is trained by using training examples as input and computing the classification as the output. When misclassifying, the classification error is used to calculate gradients that will be used to update the model parameters. To make the learning process efficient, the parameters of the learning algorithm, known as learning hyperparameters, are specified and configured before the training starts. The learning procedure is performed in an iterative manner by passing over the entire training dataset several times until model convergence or some predefined termination condition is satisfied.

While offering powerful recognition performance to solve many challenging problems, complex DNN models with deeper architecture have been known to be difficult to train [HO06, GB10, GBB11, IS15]. The success of gradient-based optimisations largely depends on the choice of learning hyperparameters [Ben12]. Furthermore, it has been shown that they have an impact on model performance [PDDC09, Ben12]. There are practical guidelines and recommendations of how to define all the important learning hyperparameters such as the model architecture [Hea08, Ben12], learning hyperparameters [LBOM98, SMDH13, IS15, Ben12], and how model parameters should be initialised [LBOM98, GB10, HZRS15b, Ben12, MM15]. Despite the guidelines, there are a number of learning configurations that must be tuned. Finding effective hyperparameters involves systematic search techniques such as grid search or making use of domain knowledge to guide the exploration, which often leads to an impression that tuning DNN hyperparameters is an art [Ben12]. Regardless of the employed techniques, repeated experimentation is hardly avoidable, and this procedure is time-consuming.

Another difficulty in training a complex DNN model is that the model tends to perform particularly well on the training dataset when there are a large number of learnable parameters used to express the complex representations. Such model is considered a poor model because it cannot recognise new unseen data. DNNs in this state cannot generalise the problem and attempt to memorise the training data instead. This undesirable behavior is well known as the overfitting [Haw04]. There have been several techniques proposed to mitigate this problem, e.g. regularisation [SHK$^+$14, KH92, Ng04]. Complementary to these algorithmic improvements, using a large amount of data to train complex models gives a significant boost to the model performance [HNP09, CMGS10, CLN11, LNC$^+$11]. With a large volume of data, DNNs can learn to extract relevant and meaningful features. Larger dataset introduce more noise to the learning and make it much harder for the model to overfit the training data. However, the trade-off of this approach is that the training takes a considerable amount of time to finish. To benefit from large amounts of data, we need powerful computing resources and fast data processing systems to perform DNN training, so we can obtain trained models within a manageable amount of time.

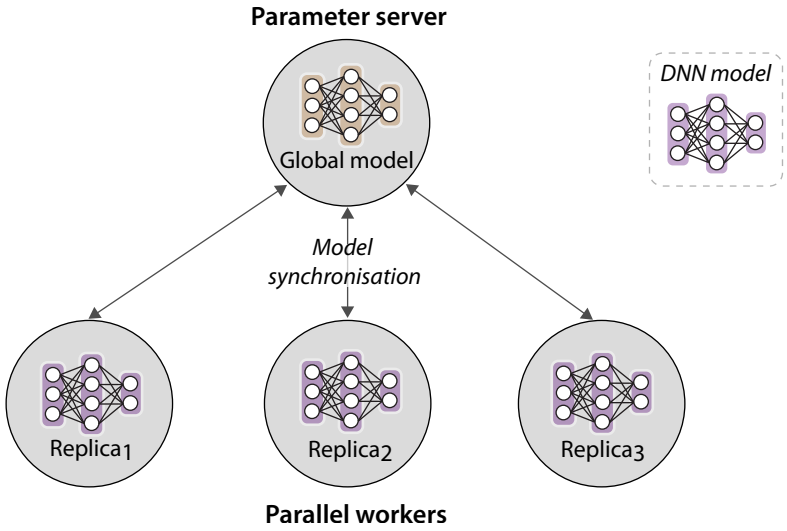


Figure 1.1: The parameter server framework for DNN training

1.2 Empowering DNNs with data-parallel computing

To achieve high classification accuracy, a DNN must have a deep structural architecture to capture complex and meaningful representations from a large amount of training data. Training such model is compute-intensive, data-intensive, and thus time-consuming. This has been one of the major current challenges in bringing DNN applications to a wide adoption. Therefore, the technology of computing hardware plays an important role in driving DNN to a success, enabling us to train many complex models with a large volume of data.

To speed up the computation in DNN layers, processors with parallel architectures such as multi-core CPUs and GPUs are commonly used. At the same time, a large volume of data imposes the need to scale the training across multiple processors or devices. The ability to scale beyond a single machine becomes more prominent when the size of the training data grows. As the consequence, distributed dataflow systems [DG08, Apa14b, IBY+07, CFMKP13] have been adopted to accelerate DNN training on distributed platforms.

Distributed dataflow systems apply *data parallelism* to perform data processing and thus enable DNN training at scale [DG08, ZCD+12, IBY+07, CFMKP13]. Training datasets are typically split into several partitions each of which is concurrently processed on a different processor or machine. Since many processors work in parallel, DNN training can finish faster. To ensure the overall model convergence, processors must synchronise their parallel training with one another throughout the process; otherwise, the training diverges. To achieve both parallelism and synchronisation in a scalable fashion, the *parameter server framework* was proposed for distributed DNN training [LAP+14]. In the next section, we introduce DNN systems with the parameter server architecture.

1.3 Scaling DNN training with parameter server framework

Similar to general-purpose distributed dataflow systems, the parameter server framework takes advantage of data parallelism. The computational workload is distributed across multiple machines, CPU cores, or GPU devices. However, what makes the framework distinct from general-purpose systems is that there is a global shared state to facilitate model synchronisation among compute nodes that perform parallel training.

The parameter server framework consists of two classes of compute nodes, *server* and *worker* nodes, as illustrated in Fig. 1.1. A server maintains *global shared model parameters* and provides read-write access to the global model, required by the workers. Typically, the system has many workers to perform data parallelism: the training data and computation workload are evenly partitioned across parallel workers. Each worker is assigned a *model replica* to carry out the training by computing the backpropagation with stochastic gradient descent over the local data partition. Since different workers hold different parts of the dataset, the computed gradients only reflect the local training. To ensure global convergence, workers must send their gradients to the centralised server and update the global model parameters. When other workers want to read the global model, they can fetch the current model state that includes gradients from different parallel workers in the system. The activity of sending gradients and fetching the current model state is essentially model synchronisation. Since the server has to handle model synchronisation requests from all workers, it can distribute the workload across multiple server instances, each of which only maintains a partition of the global model parameters.

Among the data-parallel workers, model synchronisation must happen as frequently as possible to ensure fast convergence. Without sufficient synchronisation, model replicas can diverge and thus slow down the global convergence. By synchronising frequently through the parameter servers, replicas can upload fresh model updates and read the latest state of the global model as the training iterations progress. To avoid synchronisation bottlenecks, workers carry out their parallel training independently and access the global model *asynchronously* throughout the training process.

Within a worker node, it is important to ensure high compute efficiency as the calculation in complex DNNs tend to be computationally expensive. The computation often involves a large number of matrix operations that grows proportionally to the depth of the network. Since model training is an iterative process and requires a number of passes over the training dataset, the compute efficiency directly determines the amount of time spent on each iteration and thus the training completion time. In practice, the computation in each iteration can be parallelised in several ways depending on the type of parallel hardware architecture. For example, we can parallelise each matrix operation one after another with a GPU, or we can employ data parallelism across multiple GPUs, CPU cores, or machines. Regardless of specific techniques, better hardware efficiency is always desirable as it enables us to finish the long-running training in shorter time.

In addition to accelerating model training with various parallelisation methods, the convergence rate heavily depends on the choice of learning configurations used in model replicas. Learning configurations for DNNs often involve a wide range of learning algorithm hyperparameters, such as learning rate, momentum, and weight initialisation method for model parameters. By using an effective hyperparameter configuration for a given DNN model that trains over a given dataset, the model convergence can progress robustly and effectively, resulting in a high model accuracy within a manageble amount of training time.

In the following section, we move on to the discussion of key challenges that have a direct impact on the system performance with respect to convergence time.

1.4 Problem statement

With distributed DNN systems, there are three challenges that directly influence the training completion time. We describe each of them as follows.

Model synchronisation. With the parameter server framework, the synchronisation between workers and servers is realised through network communications and thus results in network utilisation. If the deployment of a framework has too few parameter servers, the servers will suffer from network congestion because the limited physical capacity of the underlying network bandwidth cannot cope with frequent synchronisation requests from all the workers in the system. In this case, the DNN model will converge slowly or not converge at all. On the other hand, one could argue for allocating many or even redundant server nodes to avoid the problem. However, we often have a fixed-sized compute cluster as a finite compute resource. Assigning too many machines or processors to servers is wasteful because they should rather be used as data-parallel workers to speed up the computation.

This suggests that there exist a trade-off and the need in balancing the cluster resource, both compute and network, to achieve fast global convergence. In other words, the decision on *cluster resource split* must be made when training DNNs with the parameter server framework. In practice, it is often not straightforward what the optimal resource split should be because it depends on several factors, e.g. hardware specification of the cluster, the computation workload of the model that we wish to train, etc. Finding the optimal cluster configuration usually involves trial-and-error experimentation, which can take a great deal of time and effort [YRHC15].

Compute efficiency of workers. The compute workload of most DNNs consists of a long sequence of matrix operations. The compute capability of data-parallel workers thus becomes crucial as it determines the compute time of every single iteration.

Much of previous work has shown that GPU multiprocessors offer fast matrix calculations [CLL+15, JSD+14, AAB+16, ABC+16, FHJ+14, CHW+13, ZHW+15]. A GPU typically executes one matrix operation at a time by parallelising the operation using a large number of concurrent threads. Since the workload of most state-of-the-art DNN models is compute intensive, many existing DNN systems make use of multiple GPUs to further speed up computation. However, these systems pay little

attention to system efficiency, especially when the compute workload is distributed across multiple devices: each device is assigned a fraction of computation in the iteration. By doing this, the workload per device decreases as more devices are introduced into the system. Since the computational complexity of the matrix operations is heterogeneous, i.e. some operations require fewer multiprocessors to operate compared to others, the overall system efficiency tend to decrease due to the decline in hardware utilisation on each device.

Here we argue that we should always make an effort to maximise all the available processing hardware. Otherwise, it is wasteful and we miss out the opportunity of getting an even better speed-up. With better efficiency in utilising the compute hardware, we can reduce the training time given the same resource budget, or we can achieve the same training performance but with fewer devices and thus save cost.

Finding effective choices for model configurations. Given a DNN architecture to solve a recognition problem, the configuration of learning hyperparameters of model replicas is important because it has a direct impact on the rate and quality of convergence. Since training a complex model with a large dataset is often a long-running job, it is preferable to configure the model with learning hyperparameters that yield fast convergence rate. The process of finding such configurations is, however, often difficult and usually requires domain knowledge or extensive experience. This is because there are a large number of learning hyperparameters to be tuned. To solve a new recognition problem given a new training dataset, finding effective model configurations can take a significant amount of time.

To find such configurations, distributed dataflow systems are adopted to facilitate the hyperparameter explorations. However, these systems are originally designed for general-purpose data processing and thus provide little support for an efficient *exploratory workflow*. They often execute the workflow as multiple independent dataflow jobs, which is inefficient due to the following reasons: (i) it is difficult to identify not promising configurations without completing all the jobs for the global comparison and; (ii) the dataflow jobs that constitute an exploratory workflow often have substantial overlaps in their intermediate results. Such intermediate data should, therefore, be reused across jobs rather than being recomputed over and over again. These challenges impose the need for systems that are capable of performing an efficient exploration of model hyperparameters.

1.5 Research contributions

In this dissertation, we describe our solutions to address the aforementioned challenges through three distinctive systems, which are introduced next.

Model synchronisation

When adopting the parameter server framework to train a distributed DNN, one has to balance the use of compute and network resources to prevent system bottlenecks that impede the training progress.

As discussed earlier, this balance involves the decision on a cluster resource split between workers and servers. Finding an optimal cluster configuration is a time-consuming process and thus not practical.

To remove the need for decisions on the resource split, we propose to train DNNs with a *decentralised* architecture that does not have parameter servers. With this architecture, all machines in the system are used as workers. The benefit of this approach is three-fold: (i) we can remove complexity in terms of cluster configuration; (ii) all the compute resources are used for data parallelism; and (iii) all network bandwidth in the system is used for model synchronisation.

Under this decentralised approach, workers synchronise their training in a peer-to-peer fashion, i.e. they exchange gradients by communicating directly among one another. However, simple communication patterns, such as all-to-all communication, are unsuitable because the communication cost has quadratic complexity, which quickly causes network contention and this limits scalability. To enable the system to scale on a large compute cluster while maximising all the cluster resources for fast convergence, we introduce the following techniques:

Scalable decentralised synchronisation. We adopt a new approach to synchronise model replicas termed *partial gradient exchange*: each worker periodically computes an updated model gradient, partitions the gradient according to the available network bandwidth, and exchanges the partitions with other workers. Since a worker receives a different gradient partition from other workers in each synchronisation round, convergence is unaffected as workers eventually receive the complete model gradient with bounded delay.

Decouple CPU and network use. A worker decouples its use of CPU resources for model training from the use of network bandwidth for replica synchronisation: parallel compute tasks train the model replica as fast as possible, generating model gradient updates. Gradient updates are accumulated asynchronously by separate network tasks when awaiting transmission. Before transmission, the accumulated model gradients are partitioned so that synchronisation traffic fully saturates the network bandwidth while maintaining a constant latency.

We present **Ako**, a decentralised DNN system that is designed to saturate cluster resources. All nodes execute workers that fully use the CPU resources to update model replicas. To synchronise replicas as often as possible subject to the available network bandwidth, workers exchange partitioned gradient updates directly with each other. The number of partitions is chosen so that the used network bandwidth remains constant, independently of cluster size.

Compute efficiency of workers

GPUs are widely adopted for DNN training due to its ability for performing fast matrix calculation. However, training complex models on GPUs still takes a considerable amount of time. Therefore, higher compute efficiency of GPUs is preferred because it introduces opportunities to further reduce the training time, given the same resource budget.

Since DNN computation typically consists of a series of matrix operations with different computational complexities, some operations may underutilise GPU multiprocessors. In fact, hardware underutilisation is more likely to happen when the compute workload is split across multiple GPUs.

In an attempt to maximise the available multiprocessors for better speed-up, we propose to run *multiple model replicas per GPU* in parallel so that more than one matrix operation can be executed concurrently. By doing this, we benefit from two levels of concurrency: (i) multi-threaded execution within each matrix operation; and (ii) multiple matrix operations being executed concurrently.

By running multiple replicas on the same device, we benefit from data parallelism: different models execute on different parts of the training dataset. However, multiple models demand more memory resources. The physical memory capacity can put a constraint on the number of models, which in turn limits the parallelisability. Moreover, multiple replicas need to synchronise to ensure overall convergence. Since a GPU is capable of fast data processing, it is important that model synchronisations incurs low latency; otherwise, multiprocessors are idle for too long. Here we describe how we address these two requirements:

Memory reuse for efficient memory utilisation. To reduce the total amount of memory usage, previously allocated memory is reused for other purposes if it is not needed for the backpropagation computation. Since DNN training is iterative, a plan for memory allocation is constructed offline and used throughout the entire execution without any runtime decision on memory management.

Synchronise replicas with low latency. Replicas within the same device are synchronised using the bulk synchronous parallel (BSP) approach [Val90]: all gradients are aggregated when the synchronisation barrier is triggered. To achieve a low-latency data transfer across devices, model synchronisation is done in a hierarchical manner via a tree aggregation: model parameters are exchanged within the same device before cross-device.

We present **Crossbow**, an efficient GPU-based DNN system that is designed to maximise GPU multiprocessors utilisation for an even higher speed-up. When a GPU is not fully utilised, Crossbow instantiates multiple models and trains them concurrently to improve system throughput, which in turn reduces training time per iteration. Crossbow aims to translate the gain in hardware efficiency into better convergence speed.

Finding choices of model configuration

Distributed dataflow systems have been adopted to accelerate the process of exploring model configurations for DNN training [dat16]. Most of them execute an exploratory workflow as multiple independent dataflow jobs ,which can be inefficient.

We propose **Meta-dataflow (MDF)**, a new dataflow model that effectively expresses exploratory workflows and enables them to execute efficiently on a distributed dataflow system. MDF takes into account all the jobs of an exploratory workflow as a unified dataflow graph. By doing this, it offers opportunities to apply optimisations to the execution of the workflows. With MDFs, the system can efficiently explore options by reusing intermediate results to avoid redundant computation. At runtime, the poor learning configurations can be terminated early according to predefined quality functions. This helps reduce the total number of possibilities to be explored and thus the overall job completion time. The contribution of MDF is described as follows.

MDF model. An exploratory workflow is considered a single job. It extends a common dataflow model with two special operators: (i) an `explore` operator that allows the exploration of different learning configuration in the dataflow graph in the form of separate *branches*, each representing a particular hyperparameter configuration; and (ii) a `choose` operator that assesses the explored branches, discarding results that lead to poor convergence quality, e.g. slow convergence or model divergence. MDF assesses the quality of a branch with an evaluator function and uses a pruning function to select a subset of the intermediate results for computation in the next phase of the dataflow graph.

1.6 Dissertation outline

The remainder of this dissertation is organised as follows. Chapter 2 provides an overview of DNN background, DNN training systems, and discusses the challenges that determine the performance of DNN systems. First we give the background on DNNs, describe the underlying learning algorithms and characteristics. Then we touch upon the importance of using large training datasets, which significantly boosts model performance but yields long training time. Because of that, we review the advancements in compute hardware and the adoption of distributed dataflow systems that accelerate DNN training. We give details on the parameter server framework, the current state-of-the-art approach that scales DNN training to large cluster deployments. After that, we identify challenges and open problems that break down into three distinct aspects: model synchronisation, compute efficiency, and model configuration. For each aspect, we review various related DNN systems from the literature and provide supporting experimental evidence to highlight the open challenges.

Chapter 3 presents Ako, a decentralised DNN system that performs model training without parameter servers. We first point out the need for finding an optimal resource split in the parameter server framework. Then we introduce our novel model synchronisation approach, partial gradient exchange, which performs training in a decentralised fashion and thus does not require decision on the cluster configuration. We explain how gradients are partitioned and exchanged directly among workers and how this approach maximises the cluster resources (compute and network bandwidth) to achieve fast model convergence. We finish the chapter with a series of experiments on system scalability by experimenting on a 64-machine cluster, system efficiency, the effect of gradient partitions, and convergence speed compared to other DNN systems.

Chapter 4 introduces Crossbow, a high-performance DNN system that is designed to maximise the compute efficiency of GPU multiprocessors. We explain how we leverage GPU concurrency beyond the traditional multi-threaded execution by training multiple model replicas in parallel on the same device when there is room to fit more computation. We then give details how to reduce the total memory usage to support multiple replicas and how the replicas synchronise to ensure low-latency data transfer across GPUs. The chapter finishes with experiments on convergence and scalability compared to other existing DNN systems. We also demonstrate the benefit of running multiple models over an alternative approach, which is commonly used.

Chapter 5 describes Meta-dataflow (MDF), a new dataflow model that effectively expresses exploratory workflows and enables an efficient execution of the workflows on distributed dataflow systems. We explain how MDF is integrated with the traditional dataflow model. We then discuss the system challenges that need to be addressed to ensure an efficient execution of exploratory workflows, including MDF scheduling and memory management. The chapter finishes with experiments that demonstrate the effectiveness of MDF and improvement in job completion time of exploratory workflows for a DNN and other data analytics workloads.

Chapter 6 concludes and summarises the research work described in this dissertation.

Chapter 2

Background

Training DNN models to achieve high classification performance is a compute intensive and time-consuming process. Parallel and distributed training have been adopted to accelerate the training process so that trained models can be obtained in a manageable amount of time. In this dissertation, we address the system challenges that arise when training DNN models at scale. To understand the challenges in distributed model training, this chapter provides background knowledge on several aspects as well as a review literature ranging from DNN models to distributed systems designed for large-scale DNN training.

We begin the chapter by giving background on DNNs, their learning algorithm, and the requirements for achieving high model accuracy, including the choice of learning configuration and the use of large training datasets. We then review the advancements in compute hardware commonly used to accelerate data processing. We discuss the role of general-purpose distributed dataflow systems for machine learning and their limitations. After that, we introduce the parameter server framework, the current state-of-the-art architecture that achieves scalable distributed training by executing multiple data-parallel workers that synchronise via the global model parameters. From this point on, we emphasise three major challenges that determine model convergence speed and training completion time under this framework: (i) the challenge in balancing the use of compute and network resources to avoid system bottlenecks while frequently synchronising model parameters for fast convergence speed; (ii) the need for the compute efficiency of workers to achieve better speed-up; and (iii) the challenge for distributed systems of performing an efficient exploration of models with learning hyperparameters that result in the best convergence rate. We review several existing DNN systems and discuss open issues, leading to our research contributions described in subsequent chapters.

2.1 Deep neural network (DNN)

In this section, we provide the background on how a DNN model is trained and what the requirements are for achieving good classification performance. First, we introduce the fundamental learning algorithm for DNNs and explain what the computation looks like. This gives a foundation that will

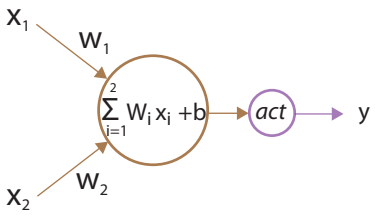


Figure 2.1: Mathematical model of a neuron in DNN

help us further understand the design and rationale behind distributed DNN systems. Next, we touch upon the fact that there are several factors directly determining the outcome of model training. We also discuss commonly used methods for tracking the learning progress and quantifying the performance of trained models. We finish the section by discussing the benefit of using large datasets to train complex DNN models as well as the challenges that arise when training with large amounts of data.

2.1.1 DNN model and its learning algorithm

Deep neural networks (DNNs) are a machine learning model that automatically learns data representations and extracts meaningful features from raw data. Since the computational model of DNNs is composed of multiple processing layers, it allows the model to learn multiple levels of data abstraction—the higher-level features are obtained by composing the lower-level features. For example, in image classification problems, features can develop from simple pixel values to edges and eventually a meaningful shape. By forming deep and hierarchical representations, DNNs can discover intricate structures in the high-dimensional data.

To solve a recognition problem, a DNN model is trained with a *training dataset*. It uses a learning algorithm to modify the internal learnable model parameters, which are the connections between layers. Inside each layer, there exists a set of neurons which are the basic computation units in DNN models. The layer connections are essentially established by the connections between neurons in different layers, and these connections are often called *weights*. Through the connections, a neuron receives an input from some other neurons in the previous layer, performs certain mathematical operations, and produces an output that can be used as an input for the neurons in the next layer.

We can express the function computed by a neuron given an input vector x as follows.

$$y = f\left(\sum_{i=1}^{k} W_i \cdot x_i + b\right) \tag{2.1}$$

In this equation, a neuron performs a weighted sum between an input vector x and its internal weights W, adds a bias parameter b, and finally applies an activation function f for a non-linear transformation. Fig. 2.1 further illustrates the basic representation of a neuron that takes numerical inputs x_1 and x_2 and has weights W_1 and W_2 associated with the inputs. After computing the weighted sum, an activation function is applied to produce the final neuron output. A well-known example

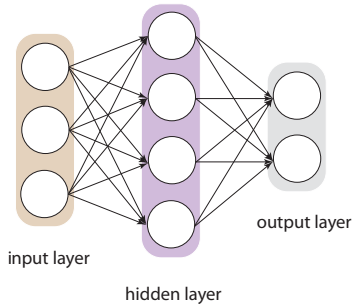


Figure 2.2: DNN model with an input layer, a hidden layer of four neurons, and an output layer

of activation functions is the logistic sigmoid, which converts a real-valued number into the range between 0 and 1. However, there are also other activation functions proven to give fast convergence rate e.g. rectified linear unit (ReLU) [KSH12, GBB11].

The *feedforward neural network* is the simplest DNN model which contains neurons organised in distinct layers. Neurons between two adjacent layers are fully connected and all connections have weights associated with them. We call this simple layer a *fully connected layer*. Fig. 2.2 illustrates an example of a multi-layer DNN model based on a stack of three fully connected layers. The first layer is usually known as the input layer, which simply takes the information from the training dataset and feeds it to the network. It just passes the input data to the next layer. The second layer is a hidden layer, which has a collection of hidden neurons connecting only with those in the adjacent layers. The last layer is the output layer, which is responsible for the classification and prediction. In practice, there are many types of DNN layers that offer specialised connectivity and capture data representations in different kinds of datasets. For example, the convolutional layer [LBBH98] is capable of extracting 2D features in the image in which spatially close objects are highly correlated.

Backpropagation as the learning method for DNNs

DNN models are trained using several *epochs*, i.e. they do multiple passes over the training dataset. Typically, the value of model weights are initialised randomly and later modified by the *backpropagation* algorithm together with an optimisation technique. The *gradient descent* optimisation [RHW86] is one of the commonly used techniques. The idea of backpropagation is to allow the model to learn from misclassification during the training process.

The backpropagation procedure involves two phases: feedforward and backward. In the feedforward phase, the training examples are presented to the model and propagated from the first layer to the last layer. The classification result is then compared to the expected output, i.e. the ground truth or label, and a loss function is used to compute the error of misclassification. In the backward phase, the error is propagated backwards, from the output layer to the first layer, and used to calculate the gradient of the loss function with respect to each weight in the model and the input of each layer. The gradient is then used according to the optimisation technique to *update* the weights such that the

Algorithm 1: Stochastic gradient descent (SGD)

```
Define : model parameter W, gradient g, learning rate η
for each epoch do
    for each iteration do
        g ← computeGradient(W)
        W ← W − η · g
```

loss function is minimised. The entire procedure is repeated for several epochs until the model has converged.

The equation below shows a simple weight update based on the gradient descent optimisation.

$$W(t+1) = W(t) - \eta \cdot \frac{\partial E}{\partial W(t)} \tag{2.2}$$

where $W(t)$ is the model weights in iteration t, η is the learning rate and $\frac{\partial E}{\partial W(t)}$ are the calculated gradients over a loss function.

As shown in Algorithm 1, the weight update procedure is repeated for many iterations. In each iteration, a set of training examples is used to compute the gradients for weight updates. With gradient descent, the number of training examples per iteration can vary from one to the total number of training examples in the dataset. If a single training example is used at a time, it is called stochastic gradient descent. Such a mode of learning results in frequent weight updates per epoch while the gradient is an approximation based on a single example. In contrast, using the entire dataset to compute gradients is known as batch gradient descent. This mode results in less noisy gradients but requires a number of epochs for convergence. In practice, the size of examples is in between the two modes: this results in frequent weight updates while producing better gradient approximates. It is common to use the term *mini-batch* [Mit97] to denote a set of training examples. The size of mini-batch can vary depending on the characteristics of the problem itself and the quality of the training dataset.

Learning hyperparameters

Backpropagation is conceptually simple. However, it involves many learning *hyperparameters* and there is no clear formula to guarantee that DNN models will converge to a good solution. Moreover, the training can be slow if DNN models have a large number of layers. This is because the loss surface is non-convex and highly dimensional with many local minima and flat regions [LBOM98]. Therefore, it is necessary to tune the learning hyperparameters to ensure successful training. Below we discuss some commonly used hyperparameters in DNN training.

Learning rate. According to a guideline [Ben12], the initial learning rate is one of the most important hyperparameters that should always be tuned. For most training, typical values are between 10^{-6}

and 1 depending on the architecture of the neural network. Although the default value of 0.01 has been widely used, there is no guarantee that this value will always give the best result. Furthermore, research has shown that different weights may have different requirements on the learning rate value to avoid divergence [LBOM98]. To improve convergence, one may need to adapt the learning rate during the training process [KSH12, HZRS15a].

Mini-batch size. The value of the mini-batch size determines how frequently the model is updated. It is commonly set between 1 and hundreds [Ben12]. Tuning the mini-batch has a direct effect on the training completion time. A larger mini-batch size results in faster computation [FHJ+14, SFD+14] because matrix operations can be executed more efficiently. However, this requires more passes over the dataset to achieve the same loss target due to fewer model updates per epoch.

Momentum. is a popular technique that enhances the basic gradient descent for robust convergence and training acceleration [WKH94]. With momentum, the weight adjustment does not only depend on the current gradient but the previous weight change is also taken into account. Conceptually, momentum is used to speed up the learning process by remembering a history of gradients in the past iterations and adding such history to the weight update in the current step. The momentum coefficient is typically a small positive value ranging between 0 and 1 which controls how fast past gradients are downweighted in the moving average operation.

The equation below shows how the momentum term can be used together with the gradient descent optimisation.

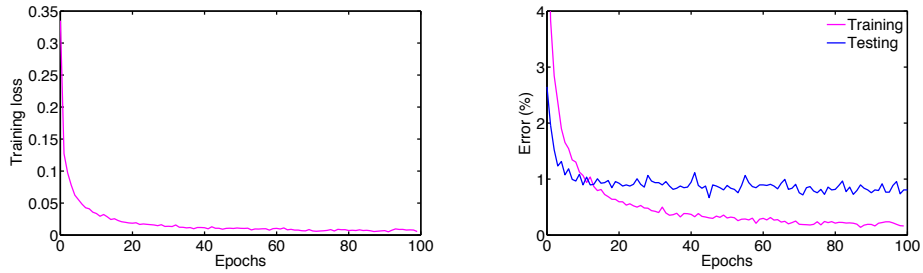
$$W(t+1) = W(t) - \eta \cdot \nabla F[W(t)] + \alpha \cdot \Delta W(t) \tag{2.3}$$

where $\Delta W(t)$ is the change in weight in iteration t and α is the momentum coefficient.

Weight initialisation scheme. Many state-of-the-art DNN models consist of a large number of learnable weights and this makes them difficult to train. In addition to the learning algorithm, the initial value of model weights have a significant impact on the learning process [LBBH98, SMDH13]. Good weight initialisation strategy helps set good initial conditions for the optimisation procedure [GB10]. Poorly initialised weights can result in model divergence or being unable to learn [MM15]. Initial weight values are often drawn from a distribution with zero mean and certain standard deviation [LBBH98]. To date, there have been many proposed strategies to facilitate the training and speed up the convergence rate of deep complex DNNs [KSH12, MM15, HZRS15a].

Model architecture and internal weight connections. There are a number of DNN architectures that have been devised to solve various kinds of problems. Most of them, however, share a common set of layer types. Below we discuss some commonly used layer types for image classification problems.

While fully connected layers are the building block for traditional multi-layer neural networks, *convolutional layers* play an important role for *convolutional neural networks*, a specialised network that takes into account the spatial structures within the data. A convolutional layer is comprised of a set of learnable *filters* or *kernels* which have a receptive field extending through the depth of the



(a) Loss (b) Accuracy

Figure 2.3: Convergence plot based on training and testing data

input. During the feedforward phase, each filter is used to convolve across the input dimensions. This operation eventually produces an activation map per filter. By stacking all the maps together, we obtain the output of the convolutional layer. Through the backpropagation process, the network learns filters that get activated when a feature at a particular location in the input is detected. Due to the convolution operation, convolutional networks exploit the spatially local correlation through the local connectivity between the filters and input regions. The extent of such connectivity can be freely set and thus becomes another hyperparameter that needs to be defined.

It is a common practice to interleave *pooling layers* with convolutional layers [LBBH98]. The purpose of a pooling layer is to down-sample the spatial size of the representation. This helps progressively reduce the number of model parameters and computational workload for a DNN. There are several ways to implement a pooling function among which *max* and *average pooling* are the most common.

The size of a DNN layer, i.e. a layer's width, is usually variable, and there is no clear rule of how deep a model should be. Since both aspects control the model's representational capacity, the model performance depends on finding a model architecture to fit the task at hand.

An example of DNN that solves an image classification task. We want to solve an image classification task by training a convolutional network, ConvNet, on the Cifar-10 image dataset [Kri09]. The network architecture of ConvNet consists of three convolutional layers, each of which interleaves with a pooling layer, and two fully connected layers. Given the network architecture and training dataset, we specify learning configurations as follows: we use a fixed learning rate at 0.01, a momentum at 0.9, a batch size of 128, and initialise the model weights based on a normal distribution with a zero mean and a variance of 0.5. With all these training configurations, we can train our ConvNet until convergence.

Model convergence and classification performance

The goal of model training is to minimise a loss function and the misclassification error. As the training continues, the value of training loss decreases, suggesting that the model is converging.

Such decreasing trend can be monitored in an epoch or iteration. Fig. 2.3(a) shows an example of a convergence plot based on training loss collected at all epochs. Alternatively, training progress can be reported based on the trend in the classification error, as shown in Fig. 2.3(b).

As the training progresses, the training loss decreases monotonically. If the training lasts for too long, the *overfitting* problem can arise—the trained model starts to perform particularly well on the training dataset. Under this circumstance, the learnable model parameters are adapted to specific features of the training data that may have no strong causal relation to the original target function. In this case, one can observe that the model's performance is high based on training data but poor based on unseen data.

To observe how well a model can generalise the problem, we can use a separate dataset that is not part of the training to monitor the model performance and identify when the training should stop. We often call such dataset the *validation dataset*. During the training, we track both training loss and validation loss. If the validation loss shows an increasing trend, overfitting is happening and the training should terminate. By using this mechanism as a stopping criterion, the trained model is likely to generalise the problem better and classify future data more accurately [Mit97].

2.1.2 Model training for the real-world applications

The performance of a DNN directly depends on the model's capability for learning data representation. Given a new recognition problem to solve, finding the neural network whose architecture fits the complexity of the task can be challenging. As discussed earlier, we can easily improve the model's capacity and complexity by increasing the layer width and model depth. With a number of hidden layers, a model can form a hierarchical recomposition of data representations. However, when a model is overly complex, e.g. containing too many parameters, it may memorise data instead of learning to generalise the problem. On the contrary, if a model is configured too simplistic, the predictive performance will become poor. Either case leads to a poor generalisation regardless of the underlying reasons.

Since high representational capability is necessary for solving a complex task, downsizing the number of model parameters is not desirable. Therefore, *regularisations* are adopted to mitigate the overfitting problem. A regularisation is a technique that reduces overfitting by introducing a complexity penalty to the loss function. There have been many proposed regularisation strategies [Mit97, KH92, Ng04, SHK$^+$14, ZF13a, WZZ$^+$13] and developing effective regularisation strategies has been one major active research area in machine learning.

One common regularisation is to apply a data augmentation technique that increases the model performance by introducing variances during the learning process [SSP03, CMGS10, KSH12, How13, WYS$^+$15, HZRS15a, WGSM16]. With data augmentation, we artificially expand the training dataset while preserving the labels. This is done by applying data transformations and making use of the distorted data for the training. For example, when we train a ConvNet model, we can apply a wide variety of 2D transformation techniques to the training images, such as random cropping,

Dataset	Type	Number of examples
MNIST [CBD+90]	Grey-scale image	60,000
Cifar-10 [Kri09]	RGB-based image	50,000
ImageNet [RDS+15a]	RGB-based image with text	14,197,122
COCO [LMB+14]	RGB-based image	2,500,000
PASCAL VOC [EGW+10]	RGB-based image	500,000
Yahoo! Music Ratings [KDK11]	Rating	10,000,000
YouTube Comedy [Goo11]	Video	1,138,562

Table 2.1: Statistics of datasets commonly used in the machine learning research community

horizontal flipping, etc. In prior work [KSH12], the authors report that they use data augmentation to increase the size of the training dataset by the factor of 2048. Without the augmentation, the model significantly suffers from overfitting. Similarly, DeepImage [WYS+15] makes use of an aggressive data augmentation, tens of thousands times larger than the original dataset, to prevent overfitting.

The evidence from the literature [HNP09, CMGS10, KSH12, WYS+15] suggests that the quality of a learned model depends on the size of the training data. With a large dataset, a DNN model can be scaled up and a more complex architecture with better representation capacity can be used [SZ14, HZRS15a, SLJ+15, SLJ+15]. In fact, even plain multi-layer neural networks such as perceptrons also benefit from larger datasets [CMGS10]. Table 2.1 shows the statistics of datasets that have been commonly used in machine learning research. The availability of sufficiently large datasets is a key factor that drives DNNs to its success.

While the advancement of DNNs in achieving world-record accuracy is underpinned by the increasing model complexity and the quantity of data, training these models is computationally intensive. The computation in most layer types involves matrix operations. For example, in ConvNet, each convolutional layer performs a number of independent convolutions over training images while each fully connected layer carries out dense matrix-matrix multiplications [HZMR16]. The number of these matrix operations is linear to the layer's width and the model's depth.

With a large quantity of dataset required, many passes over such dataset according to the back-propagation algorithm make the training process time-consuming. To get an idea how long a training process takes, we use examples for recent work that solves the ImageNet task [RDS+15a]. In this work [KSH12], the authors describe AlexNet, which won the ImageNet Challenge in 2012. Training the model with a version of ImageNet dataset [BDFF10] took 6 days on two GPUs. With further modifications in model architecture, ZFNet [ZF13b] has been reported to achieve a better accuracy on the same benchmark. Training this network took 12 days to complete 70 training epochs. With much simpler convolutional layers but a deeper architecture, VGGNet [SZ14] demonstrates the importance of extensive depth and the training took up to 3 weeks. Interestingly, the Inception-v3 model [SLJ+15] shows that a deep model does not always have to arrange its layers in a strictly sequential way. By using a multi-branch model architecture, the classification performance is improved. This model took

1 week to train with multiple GPUs. The current state-of-the-art, ResNet [HZRS15a], holds a new record—up to 1202 layers are used, and it outperforms human ability on this particular task. The deepest model took over a week to train.

In the next section, we discuss the advancement in computing hardware which play an important role in reducing training time, allowing us to train complex DNN models within a manageable time frame.

2.2 Technology and advancement in compute hardware

Advancements in computer architecture is one of the most important factors that lead to the success of DNN due to fast data processing. In this section, we discuss two processor types that are commonly used for DNN training: multi-core processors and GPUs. We touch upon how DNN computation can be parallelised on different computer architectures. We then discuss the opportunity to scale the training across multiple processors or devices, which allows us to make use of aggregate compute resource and further speed up DNN training in a distributed system.

Multi-core process (CPUs)

A recent trend in computer architecture is to move from single-core to multi-core processors. This gives us an opportunity to exploit *multi-threading* execution and a linear speedup with respect to the number of cores. An obvious way to make use of multi-core to accelerate the training with a large amount of data is to assign different training examples to different cores and execute them in parallel. When each core finishes its computation, they synchronise with one another via main memory. Multi-core systems have significant performance advantages: (i) the shared main memory has low latency, high throughput, and high capacity; (ii) given future technology trend, the continuous increase in the core number per machine guarantees a good prospect of scalability; and (iii) it is straightforward to scale out DNN training beyond a single machine by using clusters made from commodity hardware.

Graphics processing units (GPUs)

Graphics processing units (GPUs) are specialised processors with dedicated memory, which are capable of performing fast matrix operations and floating-point arithmetic. In comparison to CPUs, GPUs proportionally provide more transistors to arithmetic logic units and fewer for flow control. To date, the current generation of GPUs have evolved into many-core processors for the purpose of data-parallel computation. A GPU usually contains over a hundred processing cores and can execute thousands of concurrent threads that are scheduled on the available cores with little overhead. These threads typically share the same instruction but execute on different data elements. Such parallelisation is known as single-instruction-multiple-data (SIMD).

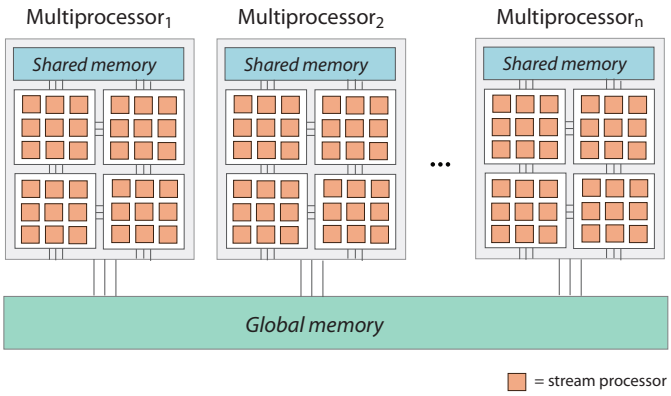


Figure 2.4: Simplified schematic for the architecture of GPUs

Fig. 2.4 illustrates a simplified version of a GPU architecture. GPUs provides two levels of parallelism. There are many multiprocessors (MPs) each of which contains several stream processors (SPs) that execute the actual computation. The computation is organised into groups of threads called thread blocks. Each block is scheduled to execute on a multiprocessor in which each thread in the block is scheduled to run on a stream processor. All the threads in the same block have access to a small amount of shared memory where they synchronise with one another during the execution. In the meantime, all the threads across different multiprocessors can access a larger GPU-wide memory, called global memory. Access to the global memory is much slower than that to the shared memory within the same multiprocessor. However, simultaneous access is possible and the hardware can carry out the memory access in parallel across several stream processors. Since GPU computation and memory accesses within the device are highly parallel, the bottleneck often arises when transferring data between RAM and the global memory.

In general, if a compute task is well-suited to SIMD, it is a good candidate for GPU parallelisation. A computation whose ratio of arithmetic operations to memory operations is high will gain maximum performance from a GPU. This is because the volume of fast arithmetic instruction can effectively amortise the cost of the slow memory accesses. Since DNNs are mostly trained with the stochastic gradient decent algorithm (SGD), which has a high arithmetic density, GPUs are considered a good fit for this algorithm.

To further speed up the computation, multiple GPUs can be used on the same machine. In this case, low-latency high-bandwidth interconnects, such as the InfiniBand link, a PCI Express, are important because they help reduce the GPU-to-GPU communication latency. With new GPU generations, the connectivities among different devices are based on a peer-to-peer framework, e.g. the cube mesh topology [NVI16].

Distributed training with a cluster of machines

Increasing the scale of DNNs with respect to the number of model parameters and the amount of data impose the need for a scalable system to accelerate the training. Since training a complex DNN model is both compute- and data-intensive, using a single powerful machine is constrained by the rate at which compute hardware improves. In fact, this has been dominated by the rate at which data size has been growing. Especially in the industry scale, data is collected and stored in a distributed fashion. In this case, it is reasonable to process data in a distributed fashion so bottlenecks of data transfers to single machine are minimised. Furthermore, it has become more convenient to use a distributed cloud computing platform that can scale out and accommodate more resources, e.g. processors, disks, or memory.

With a similar concept to the parallelisation across multiple cores and GPU devices, each machine computes gradients based on the local data and periodically synchronises with the other machines through the network. Since communication latency can introduce delay during the training process, one should pay careful attention to how to keep the latency under control.

In the next section, we describe how distributed DNN training can be realised using data-parallel systems such as distributed dataflow systems and discuss techniques used to achieve good system scalability.

2.3 Distributed dataflow systems for machine learning

In this section, we will focus on distributed DNN training. We start by discussing how model training can take advantage of distributed dataflow systems that parallelise the computation. Since these systems are based on batch processing, it is worth discussing how a synchronous parallel style imposes a synchronisation barrier, which eventually leads to limited system scalability. Next, we describe the idea of using shared global model parameters together with an asynchronous execution to improve the scalability. The idea essentially leads to the development of the parameter server framework, a scalable approach commonly used for training large-scale DNNs. We finish the section by reviewing some open-source systems that support the parameter server framework.

2.3.1 Distributed dataflow systems

With a large amount of training data, performing a DNN training becomes time-consuming. Therefore, large-scale data processing frameworks are appealing. Distributed dataflow systems such as MapReduce [DG08], Spark [ZCD+12], Dryad [IBY+07] have been widely adopted to process data-intensive jobs. These general-purpose dataflow frameworks adopt a data-parallel computational paradigm, which performs data processing in batch, resulting in high system throughout. These systems effectively operate based on the bulk synchronous parallel (BSP) approach, which is well-suited for parallelising batch-style algorithms.

Several machine learning algorithms including backpropagation have shown to benefit from data-parallel frameworks [CKL+06, DCML08, WHK08]. To accelerate DNN training, we distribute the gradient computation: the training dataset is first partitioned into many non-overlapping sets. Each partition is assigned to different mappers, which apply a map function to the local data and independently compute partial gradients. At the end of each iteration, partial gradients from all mappers are sent to the reducer, which computes the the global gradients by summing all partial gradients computed by the mappers. After that, the model parameters are updated using a traditional batch gradient descent and the updated model is sent back to the mappers for the next iteration. This procedure is repeated for many epochs until the model has converged. Such distributed gradient computation offers an improvement in wall clock speed over a single machine approach [CKL+06].

Since these general-purpose distributed systems are not originally built for machine learning workloads, it is difficult to express complex iterative computations. Mahout [Apa14a] on Apache Hadoop [Apa11] and MLlib [Apa16] on Apache Spark [Apa14b], have been developed to facilitate the development of scalable machine learning applications. These libraries comprise of distributed implementations of a wide variety of standard learning algorithms. Due to the tight integration with the underlying systems, the libraries have highly optimised components within the data processing ecosystem, which translates into performance gains.

Although a batch-processing style allows distributed gradient computation to scale, the scalability is limited to only small clusters, e.g. with tens of nodes [LAP+14, WCD+15]. This is because they require synchronous iterative communication: all computation related to the same iteration must complete before the next iteration begins. This imposes a synchronisation barrier overhead between consecutive iterations. When we deploy such a system on a cluster of machines, there is a high probability that some machines will execute more slowly than the others due to random variations in execution time. This problem is known as the straggler problem [CHK+13]. Each iteration can only proceed at the rate of the slowest machines. Due to the increase of processor waiting time, the system becomes inefficient, and the overall training is delayed.

While some learning algorithms can be successfully trained with minimal global model synchronisation [MMS+09], research has shown that the backpropagation algorithm needs iterative model updates for convergence [MHM10]. By executing data-parallel processors in isolation and only synchronising them at the end, the perceptron algorithm does not converge [MMS+09]. However, convergence can be achieved if data-parallel workers synchronise periodically [HGM10]. This evidence confirms that frequent model synchronisation is necessary for the backpropagation algorithm.

With the strong requirement of frequent synchronisation, the straggler problem occurs even more often. This makes distributed synchronous training become inefficient and unable to take full advantage of the compute resources in the cluster. To prevent processors from wasting cycles, *asynchronous* gradient computation approaches have been studied [NBB00, ZLS09, CHK+13]. With asynchronous training, the synchronisation barrier is relaxed: data-parallel processors compute gradients in parallel and proceed to the next iterations independently. Different processors may use

different versions of model parameters to compute gradients and global model updates are performed incrementally [NBB00].

However, it is difficult to adopt the asynchronous execution on bulk-processing frameworks such as MapReduce. If the state of the global model parameters modify asynchronously, this will violate the idempotence and deterministic guarantees that are the principal component to support fault tolerance and task backup features [HGM10]. Another reason that makes these systems ill-suited for asynchronous training is that they do not expose any global shared state. There is no explicit interface available for the data-parallel processors to access the intermediate global state when needed [PL10]. To expose the continuously modified global state, a new framework with global shared model parameters has been proposed, and we give further details about this in the following section.

2.3.2 Computing distributed gradients with shared global model parameters

According to prior work [HGM10], BigTable [CDG+06] can be used to support the mutable state of global model parameters in the MapReduce framework. BigTable is a distributed sorted map that stores and persists data entries indexed by row, column, and timestamp. Since time is associated with the data, it is possible to maintain multiple versions. By storing model parameters in BigTable, data-parallel processors in MapReduce can asynchronously read the current state of the model and write the model updates.

Similarly, Piccolo [PL10] is a data-centric programming model that enables the computation on different machines to share distributed mutable state using in-memory key-value table. The data entries in the table are partitioned across multiple nodes for load-balancing. By exposing shared global state through a set of interfaces, Piccolo permits efficient implementations of applications that require instant access to intermediate state.

Other work [SN10] has proposed a scalable system for the inference of latent topic models using a distributed key-value store. Specifically, the system makes use of a distributed memory object caching system called Memcached as the synchronisation mechanism to handle different versions of the global state. The system relies on an asynchronous communication pattern. As a result, all the cluster resources, including CPU, network bandwidth, and disk, can be used simultaneously to accelerate model training.

The above systems show various implementations in an attempt to make the global shared state explicit and accessible for data-parallel processors. By doing so, synchronisation among processors becomes asynchronous, and the system scalability is improved. Since these systems share the same fundamental concept, they can be grouped together under the category of a *parameter server framework*.

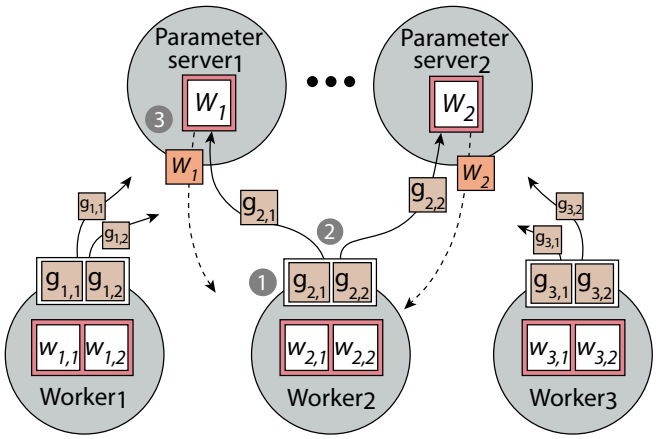


Figure 2.5: DNN architecture with parameter servers

2.3.3 Parameter server framework

Similar to general-purpose distributed dataflow systems, a parameter server framework takes advantage of data parallelism. The computational workload is distributed across multiple machines, CPU cores, or GPU devices. What makes it distinct from general-purpose systems is that there exists global shared mutable state that can be accessed asynchronously by computing nodes or instances [PL10, DCM+12, LAP+14, CSAK14].

Parameter server framework consists of two classes of compute nodes: servers and workers. A compute node can be either a CPU core, machine, or GPU device. A server maintains the global shared model parameters via centralised shared key-value store. The server can be implemented in a distributed fashion: there can be multiple server instances each of which maintains a partition of the global model parameters. The key-value store provides read-write access to the shared model parameters required by the worker instances. There are often multiple worker instances for data parallelism. Both training data and the computation workload are evenly partitioned across the parallel workers. The workers access the current state of the model parameters asynchronously during the training. They only communicate with the servers, and not among themselves.

In Fig. 2.5, suppose that we want to train ConvNet W which is represented with a simple vector of two parameters, W_1 and W_2. In the system, there are two servers and three workers. First, the training data is split across worker nodes, and the global model, initialised with a weight initialisation scheme, is partitioned across the two server instances. Each server is responsible for only a certain part of the model, denoted as W_n. Each worker is given a *replica* of the global model to begin the training process. During the training, each worker independently computes gradients g over its partitioned data and refines the local model replica w according to the backpropagation algorithm (step 1). In this example, the model replica at worker j maintains $w_{j,1}$ and $w_{j,2}$, whose gradients are represented as $g_{j,1}$ and $g_{j,2}$, respectively.

After processing a batch of training examples, the state of a model replica w only takes into account the gradients from its local data. To obtain the information of gradients from all workers, they have to synchronise with the parameter servers. The parameter servers keep the global model W updated by applying all local gradients g (step 2), and also give back the new version of the global model W to the workers (step 3). In this example, since each parameter server is responsible for a disjoint part of the model W_n, different parts of g have to be sent to the servers that manage the corresponding parameter. All workers perform the model synchronisation with the parameter servers asynchronously, and the training operates iteratively until convergence.

The parameter server framework uses asynchronous data communication to synchronise the model between workers and servers. With asynchronous communication, model synchronisation does not block the computation [DCM+12, LAP+14, CSAK14]. This improves the system efficiency because both compute and network resources are concurrently used. Another important advantage is that the system becomes less vulnerable to the straggler problem. Worker nodes do not need to wait for each other and thus spend most of their time on the computation rather than waiting, This enables the system to scale to a larger cluster and faster convergence can be achieved by adding more compute resources. There have been studies showing system scalability on a cluster of thousands of nodes [DCM+12, LAP+14].

In addition to scaling workers, the framework also allowing scaling for higher aggregate network bandwidth to support frequent model synchronisation. When network traffic between the workers and servers is high, we can prevent network bottlenecks by adding more server nodes, each of which will be responsible for a smaller part of the global model. In prior work [LAP+14], the authors show the use of a large number of parameter servers in large-scale machine learning training.

Dataflow sytems that support a parameter server architecture

Here we review some open-source systems that support DNN training with the parameter server framework.

Singa [OTW+15, WCD+15, WCC+16] is a distributed machine learning platform capable of training DNN models over large datasets at scale. The system adopts the parameter server framework and supports both synchronous and asynchronous training. In Singa, the user must define the model configuration of a DNN, how to deploy the parameter server framework, and what synchronisation scheme or protocol to use. After that, the runtime transparently takes care of all issues related to the distributed training such as communication, coordination, and partitioning.

The logical system architecture in Singa consists of *groups* of workers and parameter servers. Each worker group communicates with a single server group, which maintains a complete copy of the model parameters and handles communications with multiple worker groups. A server group contains multiple parameter servers each of which manages only a partition of model parameters. If there are multiple server groups in the system, they have to periodically synchronise model parameters to ensure global convergence. In the meantime, a worker group trains on a complete model replica

over a partition of a training dataset and communicates with a corresponding server group. Within the same group, the model replica may be partitioned across multiple workers, i.e. each worker is responsible for computation over a part of the model. To derive the gradients of the entire model, workers coordinate the training within the same group in a synchronous fashion.

Singa supports a wide range of distributed training methods on top of the parameter server framework, e.g. Downpour [DCM+12], Sandblaster [DCM+12], AllReduce [WYS+15], synchronous and asynchronous communication patterns, and model and data parallelism. With the concept of worker and server groups, the cluster can be configured flexibly to realise different training methods. Given a cluster resource budget, training method, and model configuration, the user finds the optimal cluster configuration to exploit the cluster resources for the best convergence speed.

TensorFlow [AAB+16, ABC+16] is a comprehensive machine learning system that operates at large scale and supports the training and inference of an algorithm on a broad range of processing hardware, including CPUs, GPUs, and TPUs. The system uses the dataflow programming model and describes an algorithm as a dataflow graph, which also maintains the mutable state of the model parameters. At runtime, the components of such stateful dataflow graph mapped across hardware devices according to user requirements. This enables users to explore various kinds of parallelisation schemes to speed up the training on the platform of choice.

TensorFlow expresses computation in an algorithm with a stateful directed graph in which vertex represents a unit of computation, called an operation, and each edge represents the input or output of the vertex. The value of vertex input and output is called *a tensor*, which is an n-dimensional array with a primitive data type. For example, a simple matrix multiplication operation takes two 2D tensors as the inputs and produces one resulting 2D tensor as an output. Since the graph is stateful, an operation may also maintain mutable state that is read and modified during execution. Such special operations are known as *variables*. For DNN training, model parameters are stored in tensors held in variables and accessed by operations during training.

TensorFlow is flexible enough to allow users to choose parallelisation. To realise the parameter server framework, the dataflow graph has multiple replicas of the graph portion that performs the gradient computation. A replica is assigned with a partition of the dataset to realise data parallelism. To synchronise the model updates among replicas, there is a model update operation that maintains the global model parameters as a set of TensorFlow variables. Replicas then asynchronously apply gradients to these variables. They can also read a variable's current state for the latest version of the model parameters and advance to the next training step. Since users can freely specify how to map a TensorFlow graph to physical hardware, each replica can be placed on a different device to achieve the parallel gradient computation. The model update operation and the global model parameters may be placed on separate sets of devices. At runtime, a replica uses a client thread to drive the training loop over the replica's graph portion. By doing this, the execution of the TensorFlow graph is performed in an asynchronous fashion.

Discussion

The parameter server framework allows DNNs to be trained at scale on a large computer cluster, giving flexibility in scaling both workers and servers. To achieve good system scalability, the framework alleviates synchronisation bottlenecks by employing asynchronous communication patterns between workers and servers. By doing this, the compute resources in the cluster are utilised more efficiently because they spend more time on useful computation rather than waiting for network communication. As discussed earlier, the goal of distributed training is to accelerate convergence speed and obtain a trained model as soon as possible. This imposes the need for (i) frequent model synchronisation to facilitate fast convergence rate; (ii) high compute efficiency of the workers to speed up the computation of each iteration until convergence; and (iii) the right DNN model configuration that yields fast learning. In the following sections, we discuss challenges in addressing these requirements when training distributed DNNs.

2.4 Model synchronisation

In this section, we discuss details of model synchronisation that happens between servers and workers. First, we review systems that demonstrate that asynchronous model updates can effectively remove synchronisation bottlenecks, permit frequent parameter exchange, and eventually result in fast convergence. Next, we describe a technique that enforces an upper bound on the synchronisation delay to guarantee the formal convergence of SGD algorithm. Since, ideally, we want to optimise cluster resources in such a way that yields the best convergence speed, we discuss the need for balancing both compute and network resources to prevent the system from either being compute- or network-bound.

2.4.1 Asynchronous model updates

Previous research has shown that distributed gradient computation with asynchronous model synchronisation results in fast convergence and good scalability.

DistBelief [DCM+12] is a software framework that makes use of compute clusters, potentially with thousands of machines, to train large DNN models via centralised parameter servers. This work has shown that *asynchronous* stochastic gradient descent (ASGD) works well when training non-convex DNN models. ASGD introduces asynchrony in two ways: the model replicas in worker nodes execute independently and the non-overlapping parameter server shards run independently. Each model replica performs model training in a weakly synchronised fashion and is allowed to accumulate its local gradients multiple times before synchronising with servers. Under such asynchronous execution, it is almost always the case that replicas compute gradients based on stale model parameters. This is because, by the time they finish computing new gradients, some other replicas may have updated the global model. Through this relaxed consistency, the system tolerates the variance in the processing speed of different replicas, and the cluster resources are used more efficiently. Therefore, the system scales to a large number of processing cores.

Project Adam [CSAK14] is a parameter-server based DNN system that is built from the ground up to support asynchronous training. This work achieves high system performance and scalability by exploiting asynchrony. It has been observed that DNNs are resilient to inconsistencies. There are three key mechanisms that enable asynchronous execution in Project Adam: (i) the system allows model parameters to be accessed and modified without locks [NRRW11]; (ii) the system performs asynchronous batched updates to the global model by taking advantage of the fact that the model updating operation is associative and commutative; and (iii) To further improve system efficiency, the system optimises and carefully balances both computation and network workload. It has been observed that most of the time the CPU of parameter server machine is underutilised. Therefore, Project Adam restructures the computation across the cluster and offloads some parts of the gradient computation from workers to servers to cut down communication overhead.

Hogwild! [NRRW11] is a simple SGD update scheme that eliminates the overhead associated with locking. This work, with a theoretical proof and empirical experiments, shows that SGD can be implemented and executed in parallel without locks. Under such *lock-free* scheme, processors can equally access the shared model parameters and update any part of the model parameters. Although there is a possibility that processors could overwrite each other's updates, the incidence of overwriting is low when the model is sparse because only a small portion of model parameters is modified. As a result, when the overwrite occurs, it introduces negligible error to the overall computation. This work shows that we can benefit from a near linear speedup with the number of processors on sparse learning problems.

These systems perform model updates in a fully asynchronous fashion. They empirically show that asynchronous execution can improve system scalability and result in faster convergence. In the next section, we discuss a variant of asynchrony that incorporates a bounded staleness so that model convergence can be guaranteed formally.

2.4.2 Model updates with bounded staleness

In a series of research work [HCC+13, CHK+13, CCH+14, XHD+15, WDQ+15], the authors propose a variation of the parameter server framework for distributed machine learning, which adopts a new computation model, namely *stale synchronous parallel (SSP)*. The difference between SSP based and asynchronous frameworks is that the SSP model allows some workers to proceed ahead of the others by a certain amount. In SSP, when a worker asks for the model parameters from the servers, it is given a stale-versioned model, i.e. the returned model excludes some recent updates. However, the model is guaranteed to include all updates older than a given age, which is essentially a staleness bound [Ter13]. It is possible that the model may include new recent updates. For example, if the requesting worker is at iteration i and s is a predefined staleness bound, the returned model will be guaranteed to at least include all the updates from the iteration 0 to $i-s$. SSP workers typically have their own *clock* to keep track of their iterations and only submit their model updates at the end of each

clock cycle. Due to the effect of the bound, no worker is permitted to make progress more than s iterations ahead of the slowest one.

In some SSP systems workers may read model parameters from their own local caches if the model version is valid, and fetch from the parameter servers if a newer version is required. With a relaxed consistency in SSP, workers can spend less time on waiting for the network communication and more time on data processing. Another advantage of this is that straggling workers have a better chance to catch up.

By limiting the maximum bound of staleness, the SSP model ensures the correctness of machine learning algorithms. According to theoretical analysis [HCC+13, ZLS09, NBB00, AD11], the formal convergence is guaranteed when the synchronisation scheme applies a staleness bound. For gradient descent based algorithms such as matrix factorisation and topic modeling, SSP is proven to result in model convergence [HCC+13]. In addition, it has been shown that SSP is a generalisation of the bulk synchronous parallel (BSP) model but provides faster convergence [HCC+13].

There has been research work proposing synchronisation protocols based on a bounded delay. In a prior proposal [NBB00, ZLS09], the delayed updates occur with a cyclic protocol to minimise the synchronisation overhead and write-write conflicts among data-parallel workers, i.e. the workers take turns in a round-robin fashion to send gradients to the master that maintains the shared model parameters. Workers process training data in parallel but the version of model parameters used for their gradient computation is different. Due to the round-robin rule, a worker operates with a delay of $n-1$ where n denotes the number of model replicas in the system. This is a special case of the SSP computational model where the bounded delay is fixed at $n-1$.

2.4.3 Challenge in balancing the use of compute and network resource

In this section, we explain that frequent updates to the global model are necessary for model convergence in distributed DNN training. However, this generates high network traffic and is likely to cause network bottlenecks due to the fact that the physical capacity of network bandwidth is finite. We argue that it is necessary to balance the use of cluster resources to avoid system bottlenecks, both compute and network ones.

Frequent model updates and cluster network utilisation

Evidence from many studies has shown that fresh parameters and frequent model updates yield better learning progress per iteration [DKW+14, LBOM98, LK09]. However, frequent propagation of updates results in high network traffic between the parameter servers and workers. Due to the finite physical capacity of the underlying network, network bottlenecks can happen. The parameter server framework offers flexibility to scale distributed training in terms of both computation and synchronisation. Similar to adding more compute nodes to parallelise the computation further, we can make use of more network bandwidth for model synchronisation by scaling the parameter servers [LAP+14]. As a result, network bottlenecks can be mitigated. Together with this solution, we

also discuss other complementary techniques that help reduce further the network traffic incurred by frequent model synchronisation.

In Project Adam [CSAK14], the network communication requirements between workers and servers are reduced by restructuring the computation across machines. The authors made an observation that the fully connected layers of DNN models tend to contain much more parameters than the other layer types. Rather than directly sending model updates, an alternative set of smaller matrices including activation and error vectors are sent to the servers and the model updates are later constructed based on these vectors at the servers. This computation offloading is beneficial because the CPUs in the parameter server machines are not as utilised as those in the worker machines. However, such special treatment of a specific DNN layer imposes the need for different communication protocols with updating the weight parameters.

In DistBelief [DCM+12], network traffic can be controlled by adjusting certain system parameters which determine the frequency of gradients being pushed to and fetched from the parameter servers, n_{push} and n_{fetch} respectively. If n_{push} is set as 1, gradients are pushed to the servers immediately after being computed; if it is set to be greater than 1, gradients are accumulated locally and the accumulated gradients are sent every n_{push} steps to reduce the communication overhead. A similar rule is applied to the n_{fetch} parameter. This approach directly reduces network traffic by taking advantage of the observation that DNNs can be trained with staleness. However, if these system parameters are set too high, model replicas may start to diverge from each other.

In prior work [ACDL14], a communication infrastructure based on the AllReduce operation over a spanning tree is proposed to propagate gradients and model parameters across machines in an efficient way. With spanning tree, machines in the cluster are organised to form a binary tree. After the gradients are computed by data-parallel workers, they are passed up the tree and summed up along the way to the root node where the global sum is obtained. To let the worker machines see each other's updates, the global sum is passed back down the tree and broadcast to all the machines. By adopting such an AllReduce communication pattern, the total amount of network traffic decreases and the system scalability is improved. However, the AllReduce operation is synchronous and requires additional techniques to mitigate straggling nodes.

Transforming data representation for the purpose of reducing the amount of network transmission was studied before [LAP+14, LASY14]. The authors adopt data compression and filtering techniques. While a lossless data compression algorithm can remove redundant information, filters selectively remove a fraction of gradients whose absolute value is too small and thus unlikely to affect the weight updates on the servers. Prior work [SFD+14] demonstrates an interesting approach in which signal quantisation is applied to gradients before sending them over the network for global model updates. However, these techniques have a trade-off: their computation cost is usually not trivial and introduces additional overhead to the workers, which already have high compute workload. Furthermore, many of them require a user-defined threshold, which can be hard to select because it directly affects the convergence behaviour.

With a parameter server framework, frequent model updates are necessary for training progress. To avoid slow convergence due to network bottlenecks, parameter servers are scaled out to facilitate model synchronisation. At the same time, workers also need to scale in order to accelerate the gradient computation. Next, we emphasise the need and challenge in balancing cluster resources, both compute and network, to avoid system bottlenecks and eventually achieve fast convergence speed when training distributed DNNs.

Balancing computation and network communication in distributed training

When training a distributed DNN with a cluster of machines, a DNN system must optimise two different performance aspects to reach the fastest time-to-convergence. It must achieve:

(i) a high **hardware efficiency**, which is the time to complete a single iteration. With more workers, a system increases parallelism and thus reduces iteration time and

(ii) a high **statistical efficiency**, which is the improvement in the model per iteration. For this, workers must synchronise their model replicas as often as possible to maximise global information gain. Since the synchronisation frequency is limited by the available network bandwidth, DNN systems scale out to multiple parameter servers to improve statistical efficiency.

There is a trade-off between hardware and statistical efficiency when allocating resources for workers and parameter servers in a fixed-sized compute cluster: allocating more machines to workers improves hardware efficiency, but it also reduces statistical efficiency unless more parameter servers are added. Conversely, more parameter servers increase the available network bandwidth for model synchronisation. This enables more frequent model updates and thus improves statistical efficiency, but if the parameter servers take resources away from the workers, hardware efficiency is reduced.

In practice, modern distributed deep learning systems [AAB+16, CLL+15, CSAK14, OTW+15] require these decisions on *resource allocation* [YRHC15]. For a given resource budget, typically the majority of machines execute as workers while the rest form a group of parameter servers, together maintaining the global model. For example, TensorFlow [AAB+16] supports a typical resource split resulting in several worker machines that synchronise with the centralised group of parameter servers; Singa [OTW+15, WCD+15, WCC+16] supports even more advanced cluster configurations with multiple parameter server groups.

2.5 Compute efficiency

In a parameter server framework, the computational capability of data-parallel workers is crucial as it determines the time of each iteration. While multi-cores processors are commonly used to scale out DNN training on a cluster, prior work has shown that GPUs offer fast matrix computations and can scale up the computation on a single machine [RMN09, KSH12, ZZY+13, CHW+13].

In this section, we focus on the use of GPUs to exploit data parallelism. We begin the section by reviewing several machine learning systems that train DNNs with GPUs. Next we look at how

the computation can be parallelised further with multiple devices and how the training workload is distributed across GPUs. Towards the end of the section, we argue with supporting evidence that most existing systems do not necessarily utilise the compute hardware in an efficient way and thus miss out opportunities to maximise the compute resource for a better speed-up.

2.5.1 GPUs for machine learning computation

Here we review a set of machine learning systems that can train DNN models on GPUs. We begin with an overview of each system and then discuss how multiple devices are used to scale up the training process.

GPU-based DNN systems

cuDNN [CWV+14] is a GPU-accelerated library with optimised low-level computational routines for DNN execution on NVIDIA GPU devices. The library includes forward and backward propagation variants of several DNN layers such as convolution, pooling, softmax and several types of activation functions. The implementation of these primitives is highly optimised for the execution on NVIDIA parallel architectures. With its provided primitives, users are free to define abstractions of their own DNN models, e.g. by combining available layers. This allows an easy integration with the library to many existing DNN frameworks.

MXNet [CLL+15] is an open-source machine learning system that supports multiple programming languages. The system makes use of the cuDNN library to execute DNN layer operations. It combines two programming paradigms, declarative symbolic expression and imperative computation, to achieve good programmability and efficient execution.

MXNet makes use of symbolic declaration to specify a DNN structure and presents it as a computation graph. Multi-output symbolic expressions, called symbols, realise basic algebraic operations for matrices and complex DNN layers. To evaluate a symbol, a free variable is bound with data to produce the required outputs that are previously declared. For DNN training, the symbol evaluation is performed during the feedforward computation and automatic symbolic differentiation is performed during backpropagation. The advantage of constructing a computation graph before execution is that it gives an opportunity to optimise system performance. MXNet has a graph optimisation layer operating on top of its scheduler. Another benefit of the graph representation is that the memory utilisation can be optimised. —two practical techniques, in-place and co-share, are used for efficient memory use. These techniques permit memory recycling and allow multiple operations to use the same memory allocation when there is no access conflict.

Apart from the declarative part, tensor computations are performed in an imperative fashion and lazy evaluation is used to resolve the data dependency issue. When the model parameters are updated with gradient descent optimisation, it is straightforward to describe how the gradient is applied to the current state of weights via an imperative program, e.g. `layer.weight -= rate * layer.gradient`.

Caffe [JSD+14] is a deep learning framework that enables researchers and developers to train, test, fine-tune, and deploy many state-of-the-art DNN models. The Caffe library is written in C++, integrated with cuDNN APIs, and provides an efficient execution of DNN models on both CPUs and GPUs. The framework has been designed with modularity, which allows an extension to support new types of computation. In addition, there is a clear separation between model representation and the actual implementation. This makes the framework attractive for researchers exploring new DNNs.

The architecture of DNN models in Caffe is represented as an arbitrary directed acyclic graph. The vertices in the graph are a layer that takes one or more input and produces one or more output. Each layer performs both the forward and backward pass. A pair of consecutive layers communicate through 4-dimensional arrays called blobs, which maintain e.g. batches of training data, model parameters, and parameter updates. Blobs gives a unified interface for memory accesses, especially when executing model on heterogenous hardware. To facilitate efficient memory management, Caffe only allocates exactly as much memory as needed in a lazy fashion. Once allocated, memory is used over repeatedly by the batches of training data that pass through the network. Moreover, an input blob can sometimes be used as the output blob if there is no point in keeping the input. However, this has to be specified explicitly by the user in the model config file.

TensorFlow [AAB+16, ABC+16] can perform a DNN training on a variety of computation devices including CPUs, general-purpose GPUs, and Tensor Processing Units (TPUs). TensorFlow expresses a DNN computation as a dataflow graph. To run a graph across different hardware types, a common abstraction is defined. Each device type implements a set of foundational methods, such as those for launching a kernel of an operation, allocating memory for tensors and states, and data transfers. When running on multiple devices, TensorFlow transparently establishes the communication between devices which hold different parts of the graph. By following data dependencies expressed in the graph, it becomes straightforward to execute independent subcomputations in parallel. To support communication, tensors with primitive data types are used as the common format for devices to exchange inputs and outputs. Tensors have a dense representation at the lowest level of execution and this simplifies the implementation of memory allocations and serialisation.

In most machine learning computation, the dataflow graph is executed multiple times. To obtain an efficient execution, TensorFlow adopts a procedure called *deferred execution* to optimise the graph and pre-plan the execution on a given set of devices. To materialise the computation on devices, a placement algorithm calculates a feasible set of devices for each operation in the graph and also finds out if multiple operations should be collocated on the same device. By combining all the techniques, TensorFlow can prune, place, and partition a large computation graph to achieve low-latency execution. In addition, TensorFlow allows users to choose their parallelisation of choice e.g. replicating the entire graph across devices for data parallelism or collocating specific model parameters with certain operations according to the user's domain knowledge.

Omnivore [HZMR16] is a prototype distributed system that trains DNN models on commodity CPUs and GPUs. This work demonstrates that each device can be treated as a blackbox whose

performance and speed-up gain can be predicted based on the maximum FLOPS of the underlying hardware. The authors of this work point out that there is a trade-off underlying all system design decisions, both single- and multi-device systems. They propose optimisers with a predictive model that can help plan the deployment for large-scale DNN training.

In this work, data parallelism is maximised via a data batch technique, that takes advantage of the fact that the implementation of GEMM (General matrix-matrix multiplication) kernels in many existing libraries is highly optimised. Convolution operations are considered as one of the most compute intensive operations. By applying a technique known as lowering [CPS06], it becomes possible to perform straightforward GEMM for the convolution operation. However, the trade-off is that it demands significantly more memory resources to operate. Unlike CPUs, GPUs have restricted off-chip memory and thus may not be able to benefit fully from the data batch technique.

In multi-device setting, the authors propose that the computation of DNN model can be mapped across physical devices by adopting both data and model parallelism. The model is split into layers, each of which is mapped to a separate compute node. Since the convolution is compute intensive, multiple compute nodes can be used to perform data parallelism in an asynchronous fashion. The computation related to global model parameters, such as reads and writes, is collocated with the node maintaining the fully connected layers. With such a physical mapping, one arising challenge is the decision on the number of data-parallel nodes for the convolution layers. The authors show that the assigned number impacts system performance.

In the next section, we discuss some practical techniques for data parallelism across multiple devices and methods to synchronise models among GPUs.

Data parallelism across multiple GPUs

Multiple GPUs are commonly used to scale up DNN training. One approach is to distribute the gradient computation across devices based on data parallelism. While this speeds up the computation in each iteration, it imposes the need for an additional mechanism to combine the gradients among devices. Here, we look at various practical synchronisation techniques, both synchronous and asynchronous, that have been adopted in different DNN systems.

Synchronous training

Most open-source DNN systems support synchronous communication across multiple GPUs. With the synchronous approach, the convergence rate and statistical efficiency is close to the sequential execution on a single device. Furthermore, it is straightforward to perform model updates across devices.

MXNet [CLL+15] achieves data parallelism by splitting the computation workload over a given set of GPUs. If there are n devices and a training batch size of b, each GPU is given a complete model replica and trains with b/n training samples for each interaction until convergence. The gradients from all GPUs are aggregated before model parameters are updated. With a synchronous scheme,

training on multiple GPUs should result in the same convergence rate as sequential training on a single GPU.

To perform model synchronisation, MXNet relies on a key-value store based on the parameter server architecture. The parameter server exposes the interfaces of a key-value store, i.e. `get()` and `push()`, which enables gradient aggregation and model re-distribution between devices. If GPU devices are used for the weight updates, GPU peer-to-peer communication may be triggered to accelerate the data transfer. However, the side effect of this is that it can result in high GPU memory usage.

Caffe [JSD+14] supports synchronous SGD training with data parallelism similar to MXNet. With multiple devices, Caffe executes an entire batch on each GPU. If we have n GPUs and the batch size specified in the configuration file is b, the effective batch size eventually becomes $n * b$. Caffe relies on the tree reduction strategy to aggregate gradients from GPUs. The model parameters are updated on the root device according to a tree topology and the updated version is re-distributed down the tree to the rest of the GPUs.

TensorFlow [AAB+16, ABC+16] was originally designed for asynchronous training at scale. However, it is possible to implement synchronous training. Users have control on how TensorFlow variables in the graph, e.g. model parameters are accessed with the use of a queue to coordinate the execution of synchronous training. A blocking queue is used as a synchronisation barrier to ensure that all workers read the same version of the model parameters. To aggregate gradients from all workers, a per-variable queue can be used to ensure atomic writes.

Although a GPU cluster is likely to have fewer machines compared to a CPU cluster, the straggler problem remains an issue, limiting the overall system throughput. To alleviate this problem, the backup worker strategy is adopted [CMBJ16]. Unlike MapReduce where a backup worker only starts after a straggler is detected, TensorFlow's backup workers run in parallel with normal workers from the beginning of each iteration. As soon as the parameter server receives all gradients required for the current iteration, model updates are performed without waiting for those that have not finished. The slower workers are considered redundant, and their gradients are ignored. By doing this, the convergence rate remains similar to single machine execution and, at the same time, the system throughput improves.

With **CNTK**, Seide et al. [SFD+14, FHJ+14] studied the parallelisability of SGD on fully connected DNN models for speech recognition problems. In this work, data parallelism is achieved by partitioning each batch across k GPUs, which compute k sub-gradients. When all sub-gradients are derived, they are aggregated. Each GPU is responsible for aggregating a $1/k-th$ portion of the gradient parameters. The computed subgradients on all GPUs are first split into k non-overlapping portions, each of which is then sent to the corresponding GPU for gradient aggregation. After being summed up, all the aggregated gradient portions are re-distributed back to all GPUs where the full aggregated gradients are constructed so the local model update can be performed. This procedure is equivalent to an *all-reduce* operation. To be able to overlap computation with inter-GPU communication, each

Algorithm 2: Elastic averaging algorithm [ZCL15]

Define : Central model W, model at i^{th} worker w_i,
worker clock t_i, communication period τ,
moving average factor α, learning rate η
while *not converged* **do**
  if τ *divides* t_i **then**
    $w_i \leftarrow w_i - \alpha \cdot (w_i - W)$
    $W \leftarrow W + \alpha \cdot (w_i - W)$
  $\nabla F(w_i) \leftarrow$ `computeGradient` (w_i)
  $w_i \leftarrow w_i - \eta \cdot \nabla F(w_i)$
  $t_i \leftarrow t_i + 1$

batch is further broken up into two halves. While exchanging the sub-gradients of the first half of the batch, the sub-gradients of the second half are being computed using model parameters that are outdated by $m/2$ samples where m is the size of the batch.

Asynchronous training

There have been proposals for asynchronous training on multiple GPUs to reduce the idle time of multiprocessors due to the straggler problem. Here we discuss the potential effect of asynchrony on the convergence behavior and an interesting synchronisation method that improves the stability of convergence.

Mitliagkas et al. [MZHR16] show that, while asynchronous training yields better hardware efficiency, it can introduce an extra momentum term to the SGD update. The authors call such a term an *implicit* momentum to differentiate from the traditional *explicit* momentum value. This work suggests that there is an optimal value of the total momentum, which is the sum of the implicit and explicit. Due to the asynchrony-induced momentum, the explicit term should be tuned to ensure that the total value does not exceed the optimal value. Otherwise, the momentum effect becomes excessive and gives poor statistical efficiency. This study also shows that the value for the explicit momentum decreases as the number of workers increases. In certain cases, a negative value of the explicit term improves the convergence rate. By properly tuning the explicit term, the training requires fewer iterations to achieve a target model performance.

Zhang et al. [ZCL15] propose the *elastic averaging SGD (EASGD)* algorithm to speed up DNN training across GPUs without the need for momentum tuning. Similar to many asynchronous methods, there can be multiple parallel workers that maintain their own local model parameters for an independent execution and periodically coordinate with one another via the centralised parameter server. However, the coordination does not depend on gradients but rather the *elastic force* that links local models to the centralised one. With such elasticity, the algorithm allows the local workers to perform more *exploration*: local model parameters can deviate from the one at the parameter server. With less frequent communication with the parameter server, the workers have more chance to explore

the parameter space. This work shows that more exploration is beneficial for DNN training and can improve the model performance due to the existence of many local optimal.

The workflow for elastic averaging is described in Algorithm 2. This workflow can be adapted to support both synchronous and asynchronous training, depending on the model update rule applied among workers (line 5). Moreover, a momentum scheme can be integrated as part of the model update operation (line 9). The model update rule for the centre variable takes the form of a moving average over both time and space of parameters computed by the local workers (line 7). The frequency of communication is defined by the value of τ.

To understand the effectiveness of the algorithm, the authors provide a proof of convergence rate and stability analysis of the asynchronous variant of EASGD under the round-robin communication scheme. The analysis reveals that EASGD is stable and does not suffer from chaotic behaviors and exponential divergence. The performance of EASGD with a high momentum value is also assessed and compared to sequential execution as well as other parallel methods, such as Downpour algorithm [DCM+12]. According to the results, EASGD outperforms the other approaches by giving faster convergence. While the other asynchronous techniques can easily become unstable when the number of workers increases, EASGD shows robustness and achieves better model performance measured by the testing accuracy.

Although asynchronous approaches improve the overall system efficiency by reducing the waiting time among different devices, the efficiency in utilising the compute resources in each device is rather an independent issue, which we believe deserves a distinct consideration. In the next section, we discuss an open problem where most existing systems do not necessarily guarantee to maximise the capability of the underlying multiprocessors to speed up the training process.

2.5.2 Open problem regarding the efficiency of compute resource utilisation

Since the workloads of most state-of-the-art DNN models for complex recognition problems are compute intensive, many GPU-based systems exploit multiple GPUs to speed up the computation. However, current systems pay little attention to the system efficiency especially, when the compute workload is distributed across multiple devices. As discussed earlier, data parallelism is realised across devices by assigning each device a fraction of the computational workload in an iteration. The workload per device decreases as we use more devices for the training. As a result, the overall system efficiency decreases due to the decline in hardware utilisation of each device. Moreover, the problem of efficiency of such multi-GPU system is likely to worsen over time as the number of multiprocessors per device continues to increase in newer hardware generations.

To support this observation, we run an experiment and measure how GPU multiprocessors are utilised when four different DNN workloads are trained using MXNet. We train models on 1, 2, and 3 homogeneous NVIDIA TITAN X GPUs [NVI15b, NVI15a]. We run each application for 10 seconds and collect the utilisation profile from each device via the NVIDIA profiler called nvidia-smi [NVI17].

DNN application	Dataset	1 GPU	2 GPUs	3 GPUs
LeNet	MNIST	65%	36%	28%
ResNet-32	Cifar	80%	67%	46%
ResNet-50	ImageNet	92%	88%	84%
Inception	ImageNet	92%	85%	80%

Table 2.2: Averaged GPU utilisation when training four DNN applications on a single GPU and multiple GPUs using MXNet

With multiple devices, the computation workload of each iteration is split evenly across devices and we record the overall averaged GPU utilisation.

Table 2.2 shows that, for each DNN workload, the averaged GPU utilisation continues to decrease as the number of devices grows. With more GPUs used, an individual device tends to be underutilised. Some applications with relatively small workload, such as LeNet, also underutilise the hardware when running on a single GPU.

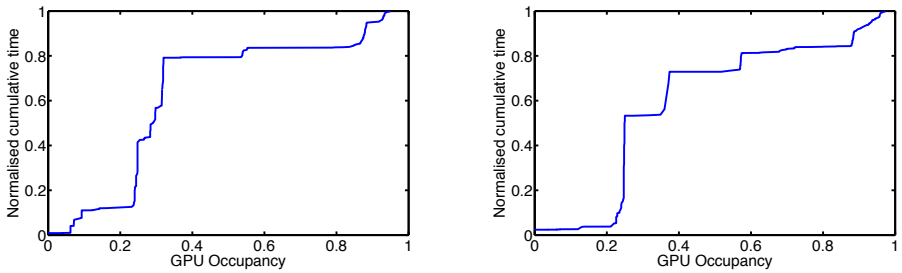
In addition to the GPU utilisation, we run another experiment and observe the system behavior through a GPU occupancy metric. With the occupancy information, we can understand whether or not task execution takes full advantage of the parallelism provided by the GPU parallel architecture.

In this experiment, we train ResNet models on two datasets, Cifar and ImageNet, on a single GPU. We train each model for one iteration using our own GPU-based DNN system, which will be described in details in §4. We collect achieved GPU occupancy measurements and the execution time of all CUDA function calls that happen in the iteration. We then present the collected measurements in a cumulative distribution function (CDF) to illustrate the distribution of GPU occupancy with respect to the execution time.

Fig. 2.6 shows that the GPU occupancy stays below 0.4 for most of the execution time for both applications. Low GPU occupancy means that the SIMD execution has room left for more concurrency. This implies that we can actually execute more computation to utilise the underlying stream multiprocessors more efficiently.

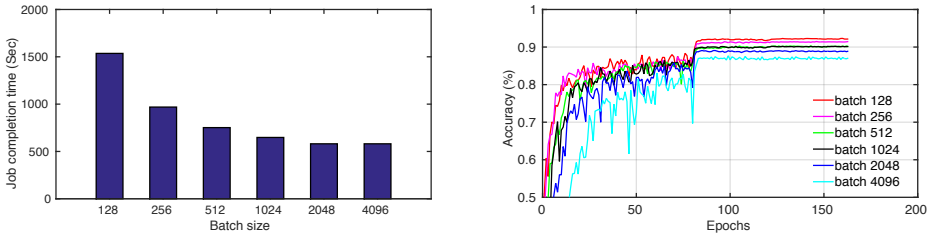
To mitigate such system inefficiency, Seide et al. [FHJ+14, SFD+14] propose that one should consider increasing the mini-batch size to improve the efficiency of a multi-device system. The authors make an observation that the GPU matrix product (GEMM) becomes less efficient for smaller matrices due to the GPU's caching mechanism. Furthermore, they argue that DNN training is considered optimal if computation and inter-GPU communication happen concurrently with a perfect overlap. With a large batch size, the latency caused by data transfers can be effectively hidden and thus reduces idle multiprocessor time.

To understand the effect of the training batch size, we also run an experiment by training a ResNet model on the Cifar dataset [Kri09] with a fixed number of training epochs but varying mini-batch sizes (128 to 4096). We collect the model's classification performance based on the testing data after each training epoch and measure the overall job completion time.



(a) ResNet-20 on Cifar-10 (b) ResNet-18 on ImageNet-1000

Figure 2.6: CDF of the achieved GPU occupancy when training ResNets with two datasets



(a) Job completion time (b) Convergence

Figure 2.7: The effect of the mini-batch size on hardware efficiency and quality of training

In Fig. 2.7(a), the results show that a larger batch size reduces the overall job completion, suggesting that the efficiency in hardware utilisation is better. However, Fig. 2.7(b) shows that a larger batch size has a side effect: not only does it slow down the convergence rate but it also gives a lower final classification accuracy. From this experiment, we learn that, by choosing to increase the batch size for the training, there is a trade-off between hardware efficiency and convergence quality.

2.6 Model configuration

Training DNNs to achieve high model accuracy depends not only on the availability of large datasets but also the choice of model configuration. Given a DNN architecture to solve a problem, the configuration of learning hyperparameters for model replicas directly affects the rate and quality of convergence. Since training a complex model with a large dataset is often a long-running job, it is desirable to configure the model with learning hyperparameters that yield fast convergence rate. To find such configurations, dataflow systems are adopted to perform hyperparameter exploration [dat16]. In this section, we discuss the importance of model hyperparameters and review existing systems that are capable of doing automatic hyperparameter exploration. We also identify open problems and discuss challenges for these systems in performing an efficient exploration of model hyperparameters.

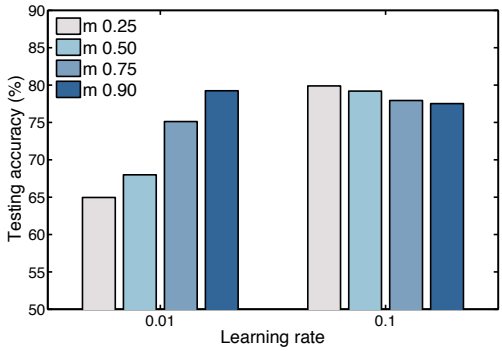


Figure 2.8: Effect of learning hyperparameters on model convergence

2.6.1 Importance of model hyperparameter configuration

One of the biggest advantages in using SGD to train a DNN model is that the algorithm can result in fast learning with a large amount of training data [Ben12]. However, in practice, the major drawback is that it requires manual tuning of learning hyperparameters. The choice of hyperparameter configuration can have an major effect on model performance and convergence speed [PDDC09, Ben12]. As discussed in §5, training a DNN model involves a number of hyperparameters. Without expert knowledge and familiarity with the dataset, one can end up with a search over large space of possible parameters to find the best configuration. A search procedure inevitably requires a number of experiments, running the training several times with various configurations. Because of the need for such hyperparameter tuning, DNNs have known to be difficult to learn.

We run an experiment to demonstrate that the hyperparameter configuration has an impact on the convergence quality in terms of convergence speed. In this experiment, we train a ResNet model [SIV16] on the Cifar10 dataset [Kri09] with various hyperparameter configurations. We train the same model architecture but tune two hyperparameters, learning rate and momentum. We explore two and four candidate values for the learning rate and momentum, respectively. The total number of possible combinations is 8. After a fixed amount of training time, we report testing accuracy of trained model from each run.

Fig. 2.8 demonstrates that different configurations result in different classification accuracy. A higher accuracy suggests that the configuration learns faster because the same model architecture is used throughout the experiments. Since fast training is desirable, the values of learning hyperparameters are commonly tuned before the long-running training process starts out.

There have been recommendations developed by experts over the years of experience that help guide new ML practitioners how to select model hyperparameters [LBBH98, SMDH13]. Despite these guidelines, different DNN model architectures require different hyperparameters to result in high classification performance. Moreover, many new architectures and learning techniques have been devised over the past years [GB10, HZRS15b, IS15, SIV16]. Thus, a standard strategy for finding the best combination of learning hyperparameters is still to run the training with various combinations of

hyperparameter values and pick the configuration that gives the most promising performance. Since hyperparameter exploration is a time-consuming process, distributed dataflow systems are used to speed this up. It has been demonstrated that using a cluster of machines to select hyperparameters is beneficial and has an important effect on the results [PDDC09].

Next we discuss a set of existing systems that are capable of performing automatic parameter exploration according to a user-defined range of hyperparameter values and search methods.

2.6.2 Systems supporting an automatic hyperparameter exploration

ML Pipelines [Spa15] is an API for the MLlib library [Apa16] in Apache Spark [Apa14b] that enables users to develop machine learning workflows on top of MLlib. Unlike other machine learning libraries, the goal of ML Pipelines is to provide support for creating and tuning machine learning pipelines, and executing them on a distributed system. Since machine learning workflows involve sequences of a wide variety of operations, such as data-processing, feature extraction, and model training, many workflows turn out to be complex. Moreover, each step in the pipeline often requires a decision on selecting the corresponding hyperparameters. With ML Pipelines, the parameter exploration in each step is encapsulated, and thus makes it convenient for machine learning practitioners to develop and modify workflows for new tasks.

A machine learning pipeline in ML Pipelines is comprised of a sequence of stages. There are two basic types of pipeline stage, called transformer and estimator. A transformer takes a Spark `DataFrame` or `RDD` as an input and produces a transformed version of it; an estimator is used to produce a model or classifier. It must be trained with input dataset first before it is used as a transformer that transforms new inputs for inference. To create a pipeline, users declare each stage, configure the corresponding parameters, and then chain all the stages together. At runtime, these stages are executed one by one; the input `DataFrame` is transformed as it goes through different stages. With the provided high-level APIs, several stages with various hyperparameter configuration are bundled together to construct a complex workflow, and the pipeline structure can be visualised as a directed acyclic graph.

ML Pipelines gives support for finding the best model for a given task by enabling hyperparameter exploration. When creating an estimator, users can also use a parameter grid called `ParamMap` to specify the range of hyperparameter values upon which the estimator is trained. To measure the model performance of an estimator, users define how they want the model to be evaluated after the training. ML Pipelines supports some built-in evaluation mechanisms, such as cross-validation, to facilitate the hyperparameter optimisation process.

MLbase [KTD+13] is a platform that enables users to experiment with new machine learning algorithms. Users specify their machine learning tasks via a simple declarative language and let the MLbase optimisers handle parameter exploration and model selection. MLbase provides high-level primitives that support the execution of many common machine learning algorithms on distributed systems where the system trade-offs are abstracted away from the users; including message passing,

data partitioning, and load-balancing. These functionalities are implemented based on the map-reduce paradigm.

The architecture of MLbase is comprised of a master and workers. Users submit a declarative task to the master. The system then transforms the declarative task into a logical learning plan describing a general workflow for how to perform the task. Since the parameter space for the exploration can be highly dimensional, MLbase has an optimiser that downsizes the possibilities in the search space to finish the task in a reasonable amount of time. An optimised logical plan is then transformed into a physical learning plan consisting of multiple executable MapReduce-like operations. This is the time when the learning algorithms and model hyperparameters are concretely specified. MLbase trains and evaluates several models with different learning algorithms by using a down-sampled dataset for each model execution to shorten the exploration process. The MLbase master distributes these models to the worker nodes and use the cross-validation method to measure the classification performance of each model configuration. The end result of the workflow is a trained model that can be used for prediction.

OpenMole [RLRC13] is a scientific workflow engine that enables users to explore high dimensional model parameters by distributing the computation workload across various computing environments. Typically models in many scientific domains, such as complex systems, involve a large number of runs over a large space of parameters, which makes model simulations compute intensive. However, the execution of models under different parameter values can be performed independently. This type of workflow is a good fit for *task parallelism* i.e. multiple models can be evaluated concurrently without dependencies. A workflow can be described using a domain-specific language that exposes advanced workflow design constructs. At execution time, OpenMole encapsulates complexity in terms of job deployment by transparently distributing tasks across the available compute resources.

Scikit-learn [PVG+11] is a Python package that exposes a wide range of machine learning algorithms for data analysis and data mining. In Scikit-learn, model hyperparameters are passed as arguments to the constructor of estimator classes and thus performing hyperparameter search is rather straightforward. Model selection is done by wrapping an estimator object in a special instance called `GridSearchCV` which takes a user-defined parameter grid. At runtime, `GridSearchCV` picks the parameters from the grid to train the estimator and the computation can be distributed across multiple cores. As of the time of writing, the package does not provide support for distributed platforms. The performance of each trained estimator is evaluated according to cross-validation and the one with the highest evaluation score is returned. Scikit-learn also supports complex estimators through a *Pipeline* object, which combines an *estimator* with many data transforming instances called `transformers`. Since many types of `transformers` also have hyperparameters, `GridSearchCV` can be used to tune parameters of all steps in the workflow.

2.6.3 Open problems in efficiency of hyperparameter exploration

Choosing a suitable value for learning hyperparameter is important for DNN training. Unfortunately, the space of possible hyperparameter values can be large. Since hyperparameter exploration often requires many experiments, it is crucial that the underlying data processing system performs the search in an efficient way.

Most existing systems that we have reviewed provide practical support for automatic exploration. This effectively removes the burden of manual tuning. However, we identify two shortfalls that affect system efficiency:

Lack of early termination. Systems do not have a runtime mechanism that allows an early termination once the exploration job is already submitted for execution. Typically, they execute the submitted jobs to completion as the result can only be evaluated at the end. We argue that early termination is necessary because it is wasteful to pursue training models with inferior hyperparameter configurations. Although a sensible range of each hyperparameter is usually defined before the actual exploration, the interaction between different hyperparameters at runtime may give results that are difficult to anticipate. With early termination, the compute resource can be freed early and thus is used to explore more promising hyperparameters. As a result, the overall exploration finishes in a shorter time.

Computation redundancy. In many cases, hyperparameter exploration based on a task parallelism strategy is inefficient. When executing an exploration as multiple independent jobs, some part of the computation among them may have substantial overlap in the intermediate results. In this case, reusing the intermediate data across jobs is preferable over recomputing them repetitively. It is more efficient to compute such intermediate results once and only diverge the computation when necessary.

2.7 Summary

Training complex DNN models is compute intensive process. To achieve high classification performance, they are trained over a large amount of data, which makes the training process time-consuming. To accelerate it, distributed systems and parallelisation techniques such as the parameter server framework have been adopted. In this chapter, we pointed out three major challenges that have a direct impact on the performance of distributed DNN systems: (i) the challenge in balancing the use of compute and network resources to avoid system bottlenecks while frequently synchronising model parameters for fast convergence speed; (ii) the need for compute efficiency of workers to achieve even better speed-up; and (iii) the challenge for distributed systems in performing an efficient exploration of models with learning hyperparameters that result in the best convergence rate. These challenges motivate our research work, which will be described in detail in the following chapters.

Chapter 3

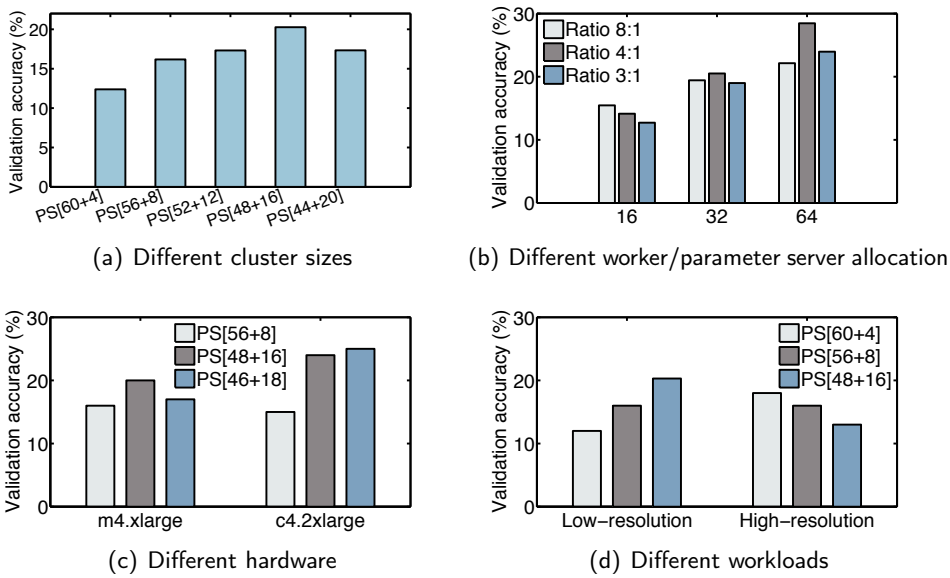
Model Synchronisation With Partial Gradient Exchange

3.1 Overview

When adopting the parameter server framework to train a distributed DNN, an open problem is that DNN systems must balance the use of compute and network resources to achieve the fastest model convergence. In a typical deployment, the workers are compute-bound, and the parameter servers are network-bound [CSAK14]: if a deployment has too few parameter servers, the model may converge slowly or not converge at all because model replicas are not synchronised frequently enough due to the limited network bandwidth between workers and servers; with too many parameter servers, compute resources, which could rather be used for additional model replicas, are wasted, leading to slower convergence.

Many existing DNN systems rely on distributed parameter servers, which require users to make a decision on cluster resource allocation, i.e. finding an optimal resource split between workers and servers so that the system does not suffer from compute and network bottlenecks. The process of finding such a cluster configuration is time-consuming as several factors affect the configuration. To address this challenge, we present, *Ako*, a DNN system that performs model training with a *decentralised* architecture that does not require resource split decisions because all machines are used as workers. With this architecture, workers must synchronise differently from ones in a parameter server framework: they must directly exchange model parameters among each other. To enable the system to scale to large deployments, Ako must deal with network communication caused by frequent model synchronisation and ensure that there are no network bottlenecks.

We begin this chapter by highlighting the problem of finding an optimal resource split in a parameter server framework and verifying it with empirical evidence. We then introduce a new scalable synchronisation approach, *partial gradient exchange*, which allows a DNN system to scale without explicit cluster configuration. We describe how workers synchronise their training while maximising all the cluster resources for fast convergence without introducing system bottlenecks.



(a) Different cluster sizes (b) Different worker/parameter server allocation

(c) Different hardware (d) Different workloads

Figure 3.1: Effect of system and workload changes on best cluster resource allocation

We finish the chapter with a series of experiments on system scalability, system efficiency, and convergence speed compared to other DNN systems that rely on the distributed parameter server approach.

3.2 Resource allocation problem

An optimal resource allocation for workers and parameter servers should result in the fastest time-to-convergence. In this section, we show experimentally that the best resource allocation depends on many factors, including the cluster size, the hardware capabilities, and the training data.

We deploy a DNN system with parameter servers on a 64-machine cluster, training a model for the ImageNet benchmark (see §5.5.1 for details). Fig. 3.1(a) shows the accuracy after one hour of training for different resource splits between workers and parameter servers on the cluster. The result shows that the best accuracy is achieved for an allocation of 48 workers and 16 parameter servers. Note that the extreme allocations that emphasise hardware efficiency (60 workers) or statistical efficiency (20 servers) exhibit 38.9% and 14.5% worse accuracy, respectively, compared to the best allocation.

Cluster size. The ratio of the optimal allocation between workers and parameter servers changes with different cluster sizes. Fig. 3.1(b) shows the accuracy for three allocation ratios (3:1, 4:1 and 8:1) as the cluster size changes. While a 8:1 ratio (i.e. 2 parameter servers) yields the best accuracy for a 16-machine deployment, this is not the case when the cluster size increases: with a 32-machine or 64-machine cluster, a ratio of 4:1 achieves the best accuracy for this workload.

Hardware. The best allocation also depends on the machine hardware. In Fig. 3.1(c), we show the accuracy of the ImageNet DNN model for two different hardware configurations ("m4.xlarge" and

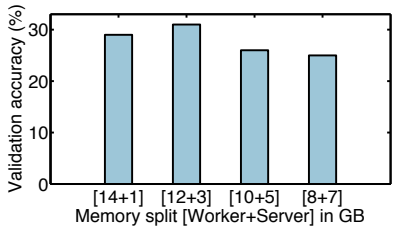


Figure 3.2: Effect of memory allocation with co-location

"c4.2xlarge" VMs) on a 64-machine Amazon EC2 deployment. For the slower "m4.xlarge" VMs, an allocation of 16 parameter servers gives the highest accuracy of 20%, but, with the faster "c4.2xlarge" VMs, 18 parameter servers achieve 25%.

Workload. In practice, input data changes, e.g. when new types of training data become available, which also affects the optimal allocation. Next, we vary the training data for the ImageNet benchmark between low-resolution (100×100-pixel) and high-resolution (200×200-pixel) images. Fig. 3.1(d) shows that the low-resolution images achieve the highest accuracy with 48 workers and 16 parameter servers. This, however, turns out to be the worst allocation for the high-resolution images, which perform best with 60 workers and only 4 servers.

Co-located deployment. A heuristic is to colocate each worker with a parameter server on the same machine [WDQ+15]. Such an approach, however, does not solve the problem: as we show in §3.5.2, it exhibits worse convergence due to the large number of global model partitions. In addition, it still requires a decision on how to share the resources on each machine: besides allocating CPU threads, the memory of the machine must be shared.

Fig. 3.2 shows the accuracy achieved on a co-located 32-machine deployment after one hour of training with different memory splits between the workers and the parameter servers. When each machine has 16 GB of RAM, an allocation of 3 GB for the parameter server and 12 GB to the workers results in the highest accuracy (31%); an equal split achieves the worst accuracy of 25%.

The parameter-server based systems that we have reviewed so far provide little support for finding the optimal split between workers and servers. It is easy to configure the cluster in a suboptimal way, and resulting in slow convergence speed. Since DNN training is a long-running job, it is important to find the optimal split between servers and workers to prevent different kinds of system bottlenecks.

To address resource allocation issue in DNN systems with parameter servers, Yan et al. [YRHC15] propose a performance model that estimates the system scalability and searches the possible configuration space. Their performance model quantifies the impact of different types of parallelisation and synchronisation strategies on both computation and network communication. The model takes DNN architecture, some learning hyperparameters, the cluster hardware specification, and information of system configuration as input. Based on this information, it identifies the potential system bottlenecks, and estimates the training time and system performance. The performance model is then used in an optimiser to analytically explore various system configurations and eventually to make a decision

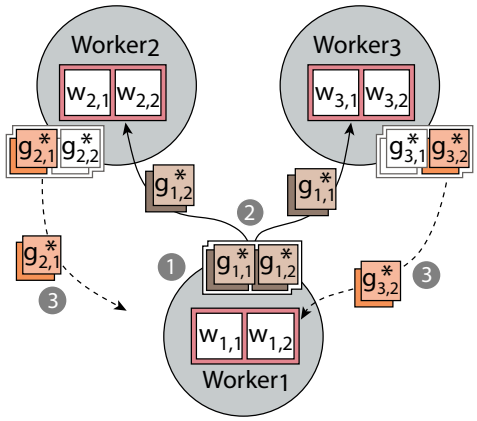


Figure 3.3: Decentralised DNN architecture with partial gradient exchange

on the optimal system configuration that minimises DNN training time. The proposed performance model is validated with a small cluster and shows high estimation accuracy. It is unclear, however, how bounded staleness can be included in the modelling and how it would affect its accuracy. In addition, such an offline approach may struggle to account for dynamic effects such as stragglers, leading to inaccurate predictions.

3.3 Distributed DNNs with partial gradient exchange

Instead of using parameter servers to synchronise the model updates produced by workers, we adopt a *decentralised* synchronisation scheme in which workers communicate directly with each other, without intermediate nodes. This avoids the challenge of having to decide on a resource split between workers and parameter servers.

A strawman solution for decentralised synchronisation is to use *all-to-all* communication between the workers, but this does not scale: n workers would require $O(n^2)$ network bandwidth for the synchronisation, but worker bandwidth only grows linearly with cluster size, $O(n)$.

To reduce the bandwidth requirement of decentralised synchronisation, workers could propagate model updates *indirectly*, i.e. with some workers relaying the updates [WZGZ14]. Such an approach, however, degrades statistical efficiency because it suffers from higher synchronisation latency, and it requires typically additional all-to-all full model exchanges [LKKU15].

Therefore, we want to design a new decentralised synchronisation approach that (i) *scales* near linearly with the cluster size, and (ii) also exhibits *high statistical efficiency* that matches current parameter-server-based approaches.

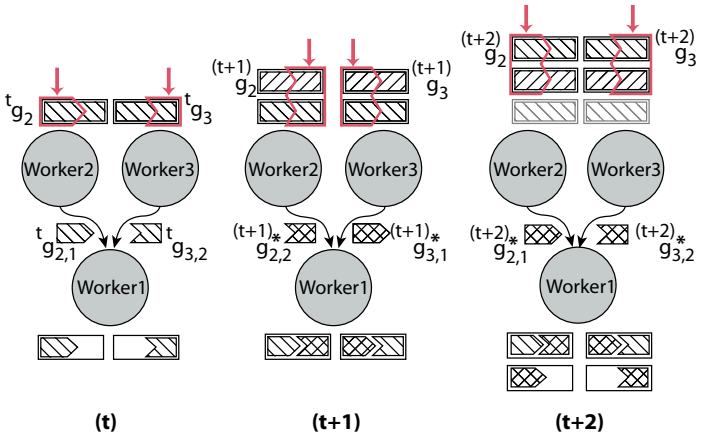


Figure 3.4: Accumulation of gradient partitions

3.3.1 Partial gradient exchange algorithm

We describe a new approach called *partial gradient exchange* that reduces the communication cost of gradient synchronisation. In partial gradient exchange, workers create disjoint *partitions* of the full gradient update. In a single round, a worker sends only one partition to each other worker, with the remaining partitions transmitted in subsequent rounds. While gradient partitions await transmission, workers update them locally as new gradients become available, which ensures that model updates are propagated with low latency.

Partial gradient exchange can thus control the network usage based on the size of the gradient partitions, while still maintaining high statistical efficiency without centralised parameter servers.

Gradient partitioning. Fig. 3.3 illustrates the synchronisation procedure with partial gradient exchange. In step 1, each worker j processes the m data points in its mini-batch and creates a local gradient g_j. Each worker then accumulates the gradient with any previously-unsent local gradients g^*_j (see below), and partitions it into p disjoint *gradient partitions*, $(g^*_{j,1}, \ldots, g^*_{j,p})$.

Each gradient partition, $g^*_{j,i}$, $i \in [1 \ldots p]$, is sent to other workers in a round-robin fashion (step 2). If p is equal to the number of workers, each worker receives a different gradient partition; if p is smaller, multiple workers receive the same partition; and if p is larger, only a subset of all partitions is exchanged in a single round.

Since all workers perform partial gradient exchange concurrently, the other workers also share their gradient partitions (step 3). The received gradient partitions are applied to the local model w_j by updating the weights. We refer to the above three steps as a *synchronisation round*.

Gradient accumulation. A single synchronisation round does not send the complete gradient for a mini-batch to all workers. Instead, it is necessary to use p synchronisation rounds to transfer the complete gradient at time t. To avoid discarding unsent gradients while new gradients are continuously calculated, gradient partitions are accumulated across synchronisation rounds.

Algorithm 3: Partial gradient exchange

1 **function** `generateGradients` *(j, d, t, η, τ)*
  input : worker index j, mini-batch data points d, gradient computation timestamp t, learning rate η, staleness bound τ
2 **while** ¬`converged` **do**
3 **if** $c_j \leq min\,(s_{j,1}, \ldots, s_{j,n}) + p + \tau$ **then**
4 ${}^{t}g_j \leftarrow$ `computeGradient` $({}^{t}w_j, d)$
5 ${}^{(t+1)}w_j \leftarrow {}^{t}w_j + \eta \cdot {}^{t}g_j$
6 ${}^{t}g^*_j \leftarrow {}^{(t-1)}g^*_j + {}^{t}g_j - {}^{(t-p)}g_j$
7 $({}^{t}g^*_{j,1}, \ldots, {}^{t}g^*_{j,p}) \leftarrow$ `partitionGrad` $({}^{t}g^*_j, p)$
8 **for** $i = 1 \ldots n$ ***in parallel*** **do**
9 $k \leftarrow i \bmod p$
10 `sendGradient` $(i, {}^{t}g^*_{j,k})$
11 $c_j \leftarrow c_j + 1$
12 **function** `updatePartialModel` *(j, i, $g_{j,p}$, η)*
  input : receiver worker index j, origin worker index i, gradient partition $g_{j,p}$, learning rate η
13 $w_{j,p} \leftarrow w_{j,p} + \eta \cdot g_{j,p}$
14 $s_{j,i} \leftarrow s_{j,i} + 1$

Fig. 3.4 shows how gradients are accumulated. At time t, each worker j computes its local gradient ${}^{t}g_j$, which is partitioned. The worker checks if there are previously-generated gradients, and as there are none, each partition is sent directly to the other workers.

In the next synchronisation round $t+1$, each worker produces a new gradient ${}^{(t+1)}g_j$. Since there exist previous gradients ${}^{t}g_j$, they are accumulated through addition, ${}^{(t+1)}g^*_j = {}^{t}g_j + {}^{(t+1)}g_j$, before being partitioned and sent to the other workers. After this synchronisation round, the gradients ${}^{t}g_j$ computed at time t have been transmitted to the other workers, thus completing the t mini-batch.

In the following synchronisation rounds, the process is analogous, limiting the accumulation to the last p generated gradients to avoid the transmission of already-sent gradients. This can be seen in Fig. 3.4 for the synchronisation round $t+2$, in which only the last $p=2$ generated gradients, ${}^{(t+1)}g_j$ and ${}^{(t+2)}g_j$, are accumulated.

Since the communication is asynchronous, accumulated gradient partitions may not be received in their expected synchronisation rounds. Although this introduces staleness in the local model, it does not compromise convergence, as we explain in §3.3.3.

Algorithm. We formalise partial gradient exchange in Alg. 3. Each worker executes two functions, `generateGradients` and `updatePartialModel`, asynchronously.

The `generateGradients` function computes the local gradient ${}^{t}g_j$ from the training data d and the local weights ${}^{t}w_j$ (line 4). It then updates the local weights (line 5) and accumulates the last p-generated gradients in an incremental fashion (line 6). After that, a worker creates (line 7) and disseminates (line 10) the gradient partitions to the other workers. This process is repeated until

convergence is reached (line 2): the procedure is stopped when the validation accuracy has not improved after a fixed number of evaluation rounds.

The `updatePartialModel` function is executed when an accumulated gradient partition is received by a worker. It updates the local model $w_{j,p}$ asynchronously (line 13).

3.3.2 Number of gradient partitions

The number of gradient partitions p impacts the statistical efficiency of partial gradient exchange. There is a trade-off: when p is small, a worker exchanges large gradient partitions, synchronising local models more quickly but requiring more network bandwidth; when p is large, a worker uses less bandwidth but requires more synchronisation rounds to receive a full mini-batch gradient update.

The best choice of p therefore depends on the available network bandwidth, and workers can use a cost model to select p when training begins: let m be the local model size, and n the number of workers, the amount of data to send the full gradient update to all workers is $m(n-1)$. With partial gradient exchange, the amount becomes $\frac{m}{p}(n-1)$ as only one partition is sent to each worker.

Assuming a rate γ at which workers compute new gradient partitions, which is profiled during system start-up, partial gradient exchange requires a bandwidth usage of $\frac{\gamma m(n-1)}{p}$ to be sustainable, i.e. have a constant transmission delay. Given an available bandwidth B at each worker (e.g. 1 Gbps), and assuming full-bisection bandwidth, the workers thus select the partition number p as:

$$p = \left\lceil \frac{\gamma m(n-1)}{B} \right\rceil \tag{3.1}$$

3.3.3 Bounding staleness

The gradients computed by each worker may use weights from previous mini-batches, which introduces staleness [Ter13]. This staleness has two sources: (i) a simple delay due to the asynchronous updates to the local models [AAGNS12] because a worker computes new gradients without receiving updates from all the other workers; and (ii) a distributed aggregated delay [AD11] because a worker only completes a mini-batch after it has received all p gradient partitions, requiring multiple synchronisation rounds.

To guarantee convergence, partial gradient exchange thus imposes a staleness bound, analogous to the stale synchronous parallel (SSP) model [CHK+13]: it limits the generation of new local gradients when a worker has advanced in the computation further than τ iterations compared to all other workers (line 3 in Alg. 3). To do so, clocks are used to keep track of iterations and staleness: each worker j maintains multiple *staleness clocks* $s_{j,n}$, one for each other worker n, and a *local staleness clock* c_j. The local clock is incremented after each produced gradient (line 11); the other workers' staleness clocks are incremented when partial updates are received (line 14).

As partial gradient exchange does not provide a converged shared global state of the model, it is necessary to maintain a clock per other worker to obtain the clock differential when checking the staleness bound: this is used to calculate the least progressed worker with respect to the current

worker. The staleness check also depends on the number of partitions p because p synchronisation rounds are necessary to fully propagate a model. Therefore, the used staleness bound is $p + \tau + 1$ (line 3). Since the updates are incorporated into the local model as soon as they are available, the partial gradient exchange procedure reduces empirical staleness, similar to the eager stale synchronous parallel model [DKW+15], which leads to faster convergence.

3.3.4 Staleness analysis of gradient accumulation

According to gradient accumulation in Ako, each gradient partition contains multiple stages of staleness as follows.

$$\sum_{k=0}^{p-1} g_{j,i}(k)$$

where g is the subgradient portion i produced by worker j, and k is the relative staleness whose upper bound is p. If the chosen number of partitions is smaller than that the number of peers in the system, it is possible that more than one peer contributes the same gradient portion to a receiving worker. Therefore we can further describe the accumulated value of the gradient portion at the receiver side as follows.

For gradient portion i,

$$\sum_{j \in J(n)} \sum_{k=0}^{p-1} g_{j,i}(k)$$

where j represents worker index and $J(n)$ is a non-empty subset of the worker index set $\{1, ..., n\}$. There are n workers in the system. With all these definitions, model updates can be done as follows. For each model portion $i = 1 \ldots \mid w \mid$,

$$w_i(t+1) = w_i(t) - \eta \cdot \sum_{j \in J(n)} \sum_{k=0}^{p-1} g_{j,i}(k)$$

3.4 System architecture of Ako

We describe the architecture and implementation of Ako, a decentralised DNN system that uses partial gradient exchange for synchronisation. To combine parallelism for model training with low communication latency for synchronisation, the architecture of an Ako worker follows a stateful distributed dataflow model [FMKP14]: as shown in Fig. 3.5, execution is broken into a series of short, data-parallel *tasks*. Tasks can update in-memory state and exchange data with each other, and also over the network.

We first summarise Ako's design goals (§3.4.1) and then give implementation details (§3.4.2).

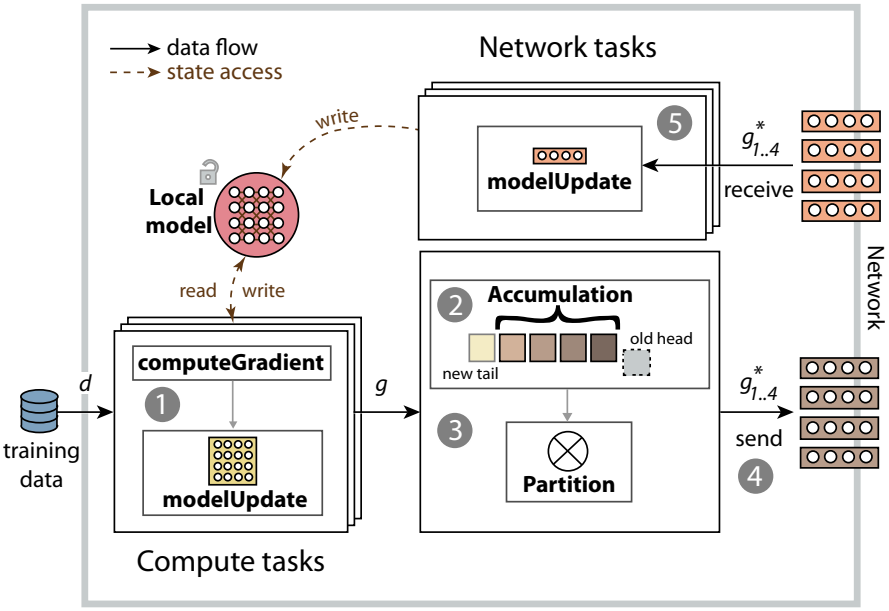


Figure 3.5: Architecture of an Ako worker

3.4.1 Design goals

(1) Full utilisation of CPU and network resources. We want each worker to fully utilise all CPU cores for training the local model (for highest hardware efficiency), while also saturating the available network bandwidth for synchronisation (for highest statistical efficiency). Fig. 3.5 shows how workers decouple *compute tasks* that train the local model from *network tasks* that exchange gradients, thus utilising all resources.

(2) Low model contention. Both compute and network tasks must access the local model without contention. Workers represent the model as a list of matrices, each corresponding to a layer. Tasks can therefore modify distinct components independently.

(3) Low-latency synchronisation. Since the statistical efficiency depends on the latency of the gradient exchange, workers must exchange gradients with low delay. For this, workers use parallel network tasks to prepare and transmit updates as soon as they are generated.

(4) Support for complex processing pipelines. Real-world DNN systems include multiple pre- and post-processing steps, such as image resizing or model validation. Ako's dataflow-based architecture means that it is easy to extend with custom processing tasks.

3.4.2 Architecture and implementation

Next we describe the implementation of an Ako worker. As shown by the numbered steps in Fig. 3.5, a worker uses a pool of compute tasks to (1) compute gradients, and a pool of network tasks to

(2) accumulate gradients, (3) partition gradients, (4) send gradients to other workers, and (5) receive gradients from other workers.

(1) Gradient computation. Compute tasks train the model in parallel. Each compute task has exclusive access to a partition of the input data as well as lock-free access to the local model.

The gradients generated by different parallel compute tasks must be aggregated at the end of a mini-batch in order to update the local model. After aggregating its generated gradient, a compute task checks if the mini-batch is finished, and if so, updates the model. Note that this occurs concurrently with other compute tasks reading the model. While such lock-free concurrent access leads to inconsistencies, it does not degrade statistical efficiency significantly [NRRW11].

Workers control the staleness across all distributed model replicas in a decentralised fashion. Each worker maintains a *staleness counter* that represents the difference in the gradient production of all workers: the counter is increased for each locally-generated gradient and decreased for each partial gradient update received from other workers. To control staleness, a compute task does not generate new gradients when its staleness counter is higher than the staleness threshold τ (see §3.3.3).

(2) Gradient accumulation. The computed gradients at the end of a mini-batch are accumulated by a pool of network tasks (see §3.3.1). Each worker maintains (i) the accumulated gradient from the previous round and (ii) an *accumulation queue*, which stores the last p generated gradients ordered by their creation round.

Every time a gradient is produced, a network task updates the accumulated gradient by adding the new generated gradient to the accumulated gradient and subtracting the old ${}^{(t-p)}g_j$ gradient at the head of the accumulation queue. The queue is then updated, removing its head and adding the new gradient to its tail.

(3) Gradient partitioning. Before the accumulated gradients are sent over the network, a network task partitions them using range-partitioning, with the position index in the internal representation of the convolutional kernels and fully-connected layer neurons as the partitioning key (see §3.3.1). The number of partitions is calculated according to the cost model from §3.3.2.

(4) Gradient sending. After that, a network task sends the gradient partitions, tagged by the partitioning range, to other workers. Each worker has a unique identifier, and the partitions are sent round-robin. After the number of synchronisation rounds equals the number of partitions, complete gradients have been received by all workers.

(5) Gradient receiving. Concurrently, workers receive gradient partitions from other workers, which must be merged to achieve convergence. Network tasks apply the gradients immediately to the corresponding parts of the local model without obtaining locks. Given that updates are applied at a fine granularity, tasks are likely to update different parts of the DNN model, which reduces the probability of lost updates.

When a worker fails, it loses its partial model, the *staleness counter* and the contents of the *accumulation queue*. In addition, since it can no longer contribute gradients to the other workers, the

failed worker can disrupt the other workers' *staleness counters* if it takes time to recover. Next we describe how Ako handles worker failures.

Ako's workers rely on *checkpointing* to save their partial models and the staleness counters, similar to SEEP [FMKP14] or TensorFlow [AAB+16]. Gradient exchanges that have not yet been applied before the failure, as well as the content of the accumulation queue can be safely discarded due to the stochastic nature of the training process [DXS+15]. This simplifies the system design at the cost of a few more iterations to achieve convergence. Finally, SEEP's master node also notifies workers of failures so that failed workers are removed from the staleness counters. Entries to the counters are re-added when workers recover.

3.5 Evaluation

The goals of our experimental evaluation are (i) to explore Ako's scalability and convergence compared to parameter server architectures (§3.5.2); (ii) to observe its statistical (§3.5.3) and hardware efficiency (§3.5.4); (iii) to explain the efficiency results by examining Ako's resource utilisation (§3.5.5); and (iv) to investigate the impact of gradient partitions (§3.5.6) and accumulation (§3.5.7) in partial gradient exchange on training performance.

3.5.1 Experimental set-up

Training datasets and DNNs. We train DNN models on two well-known datasets for visual classification: (i) MNIST [LBBH98] contains 28×28-pixel grey-scale images of handwritten digits with 10 classes. The dataset has 60,000 training images, and 10,000 images for testing; (ii) ImageNet [RDS+15b] has more than 14 million high-resolution images divided into 1000 classes, each with a ground truth label. We randomly select 100 classes to obtain a subset of approximately 120,000 images as this reduces the convergence time in experiments.

We train DNNs with 3 convolutional (interleaved with max-pooling) and 2 fully-connected layers. As the datasets have different task complexities, we use more convolutional kernels and fully-connected layer neurons for the ImageNet models. Prior to training, the datasets are partitioned evenly across the workers. The model parameters are initialised in the same way across the approaches using warm-start [DCM+12]. The model training is performed under mini-batch gradient descent. A batch size of 256 is used in our experiments to allow frequent gradient exchange while maximising the cluster compute resource.

System comparisons. We compare (i) Ako with partial gradient exchange to (ii) an architecture with distributed parameter servers (PS) and (iii) a decentralised architecture with all-to-all communication between workers (All-to-All). To be comparable, all approaches are implemented on top of the SEEP stateful distributed dataflow platform [FMKP14] with the same optimisations. To put the absolute performance of Ako into perspective, we also compare against a CPU-based distributed TensorFlow (TF) and Singa (SG) deployments on the same hardware.

Dataset	Accuracy	TensorFlow	All-to-All	Ako
MNIST	99.0%	> 20 min	14 min	7 min
ImageNet	30%	3.3 h	> 4 h	1.5 h

Table 3.1: Time to reach target validation accuracy

For Ako and All-to-All, all machines in the cluster execute workers. PS support an arbitrary number of machines to act as parameter servers, each maintaining a model partition that asynchronously synchronises with workers. We explore different allocations of workers and parameter servers: we denote a deployment with w workers and p parameter servers as PS[w+p], and mark the best configuration, as determined empirically through exhaustive search, with an asterisk (PS*[w+p]).

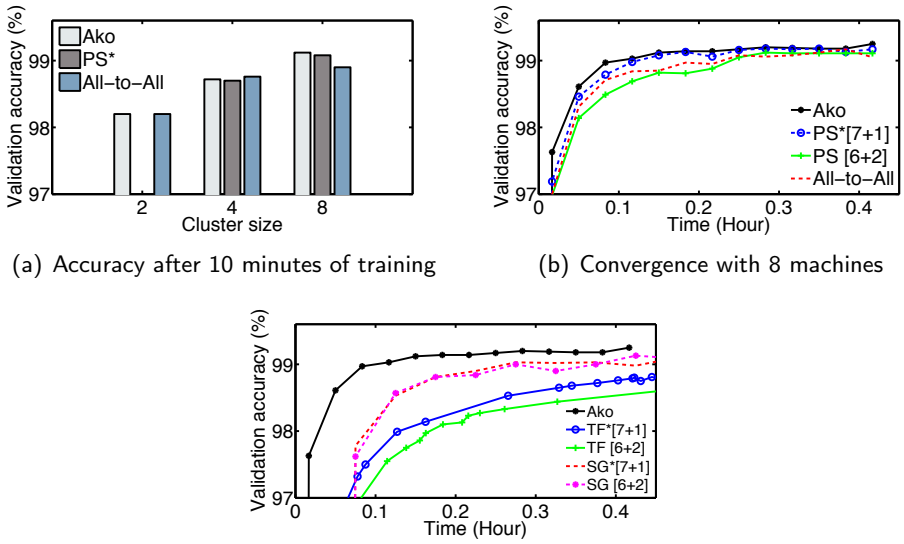
For PS, we also consider a co-located deployment [WDQ+15] in which each worker executes a parameter server (PS[w+w]). As there are two processes sharing memory on each machine, we manually choose an allocation that yields the best training performance.

For TF and SG, we use a parameter server architecture to train DNNs with the asynchronous Downpour algorithm [DCM+12]. The global model parameters in TF are represented as a set of TF *variables*, i.e. persistent mutable tensors, which can be assigned to different nodes to scale the parameter servers. In general, a DNN layer is defined as one variable in the TensorFlow computation graph [Ten16a, Ten16b, Ten16c], and, at runtime, variables are assigned to parameter servers using a round-robin strategy [Goo15].

Similar to prior work [HCC+13, CCH+14, DKW+15], we empirically set the staleness bound τ according to the used dataset and DNN model. As a heuristic, we increase τ proportionally to the number of used workers.

Performance metrics. We validate the DNN models based on the top-1 accuracy with the corresponding validation data. We measure the model training performance in terms of convergence and quantify training progress based on the validation accuracy over time. In addition, we collect the number of epochs required to achieve a predefined accuracy goal for assessing the statistical efficiency of different approaches.

Cluster hardware. We use two clusters with different hardware capabilities and sizes based on the workload: (i) we train DNNs with the MNIST dataset and conduct the hardware utilisation study on a private 16-machine cluster with 4-core Intel Xeon E3-1220 CPUs at 3.1 Ghz with 8 GB of RAM and 1 Gbps Ethernet; (ii) for ImageNet, we use a 64-machine Amazon EC2 cluster with "m4.xlarge" Intel Xeon instances, each with 4 vCPU cores at 2.4 Ghz and 16 GB of RAM. Since cluster resource utilisation is of interest, we examine the average CPU usage (in percentage) and accumulated network usage (in MBs) while training the model with different synchronisation approaches.



(a) Accuracy after 10 minutes of training

(b) Convergence with 8 machines

(c) Comparison with TensorFlow and Singa

Figure 3.6: Model convergence with different cluster sizes (MNIST)

3.5.2 What is the convergence and scalability?

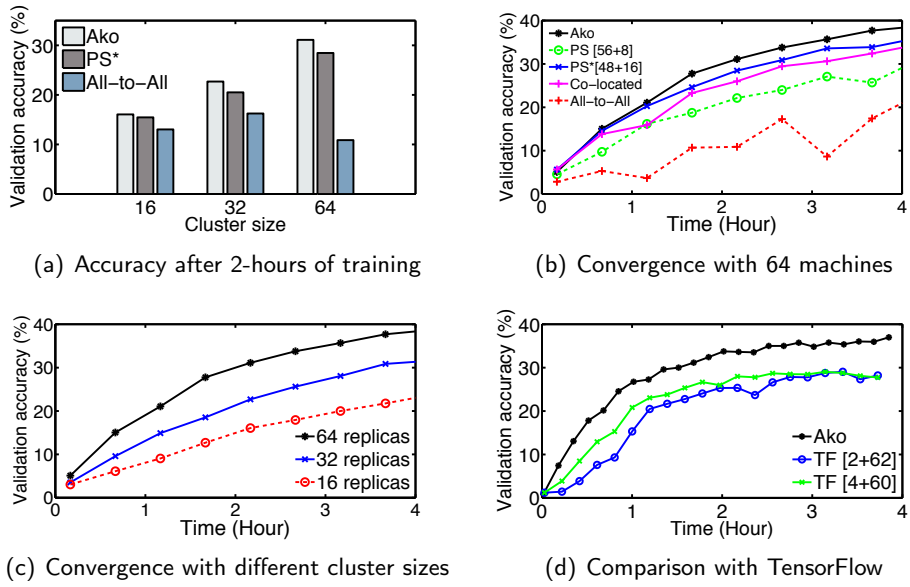
We evaluate the convergence speed of Ako in comparison to the other approaches on both the MNIST and ImageNet datasets. We also explore the scalability by training each of the models with different cluster sizes.

MNIST. We train a DNN for the MNIST dataset and vary the cluster size between 2, 4 and 8 machines. We collect the validation accuracy over 20 minutes of training. For the PS approach on 4 machines, we use a PS*[3+1] deployment; on 8 machines, we consider PS*[7+1] and PS[6+2]. We pick the best configuration to compare to the other approaches.

After 10 minutes of training, Fig. 3.6(a) shows that Ako achieves a similar convergence to PS* and slightly better convergence than All-to-All with 8 machines. Fig. 3.6(b) shows the convergence over time on the 8-machine cluster across the three approaches (with different PS allocations). The plot confirms the results from Fig. 3.6(a): Ako exhibits similar convergence to the best allocation for PS, which is PS*[7+1], and converges faster than the All-to-All approach. Synchronisation in Ako and PS does not require as much communication as for All-to-All, whose convergence is affected by the higher synchronisation delay.

With 8 machines, we also compare Ako to distributed TensorFlow and Singa when training under the same DNN workload. We deploy two parameter server configurations for both TensorFlow and Singa, TF*[7+1], TF[6+2], SG*[7+1] and SG[6+2], with training performed using asynchronous Downpour [DCM+12].

Fig. 3.6(c) shows that Ako converges faster than both configurations of TF and SG. Also summarised in Table 3.1, it takes Ako 7 minutes and TF* more than 20 minutes to achieve a target of validation accuracy of 99%. We speculate that the difference in convergence speeds between Ako and



(a) Accuracy after 2-hours of training

(b) Convergence with 64 machines

(c) Convergence with different cluster sizes

(d) Comparison with TensorFlow

Figure 3.7: Model convergence with different cluster sizes (ImageNet)

the two systems results from the synchronisation under Downpour, which allows workers to process an entire mini-batch before updating the global model.

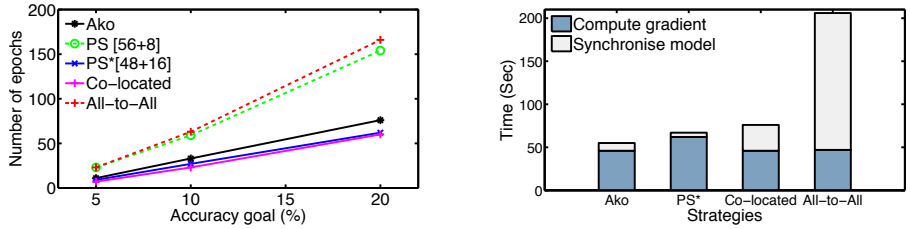
ImageNet. Next we train a more complex DNN with the ImageNet dataset on 16, 32 and 64 machines. We collect the validation accuracy after 2 hours and also observe convergence over time.

Fig. 3.7(a) shows that, after 2 hours of training, Ako achieves a higher validation accuracy than PS* on 32 and 64 machines. In contrast, the All-to-All approach converges more slowly due to its high synchronisation cost. With more machines, convergence improves for Ako and PS but not for All-to-All. PS claims a larger fraction of the machines for parameter servers, whereas Ako makes use of all machines as workers, speeding up convergence.

Fig. 3.7(b) explores convergence over time on 64 machines with different resource configurations. Overall, Ako requires less training time than PS* to reach the same accuracy after the early phase of learning: Ako has 64 workers to finish each epoch, but PS* has only 48 workers as the other machines are used as parameter servers. The difference in the number of iterations between the two strategies grows as training continues, which leads to different convergence rates.

Other configurations for PS, including (i) too few parameter servers (PS[56+8]) and (ii) co-located parameter servers (PS[64+64]), exhibit even worse convergence: the network contention at the parameter servers in PS[56+8] prohibits a tight synchronisation among workers; gathering all the global model partitions across 64 servers in PS[64+64] also causes additional delay.

Fig. 3.7(c) shows the convergence over time for Ako with different cluster sizes. Given that partial gradient exchange selects a number of gradient partitions that keeps the communication cost constant, Ako scales gracefully. With 64 machines, each worker partitions gradients into 20 partitions



(a) Epoch number for given accuracy goal (b) Break-down of epoch time

Figure 3.8: Experiment with 64 model workers (ImageNet)

before sending them to the other workers. The communication cost thus remains constant, avoiding bottlenecks that would increase the synchronisation delay.

Next we compare Ako with distributed TensorFlow on the same 64-machine cluster training ImageNet. We consider two TensorFlow configurations, TF[62+2] and TF[60+4], because our DNN has five layers, each of which is defined as a TensorFlow variable in the graph.

Fig. 3.7(d) shows that Ako exhibits competitive convergence from the beginning of training. To get to a validation accuracy of 30%, Ako takes 1.5 hours while TF takes more than 3 hours (see Table 3.1). When converged, Ako also achieves a higher validation accuracy. Our experiment demonstrates that the performance of Ako is en par with that of a standard deployment of a state-of-the-art distributed deep learning platform.

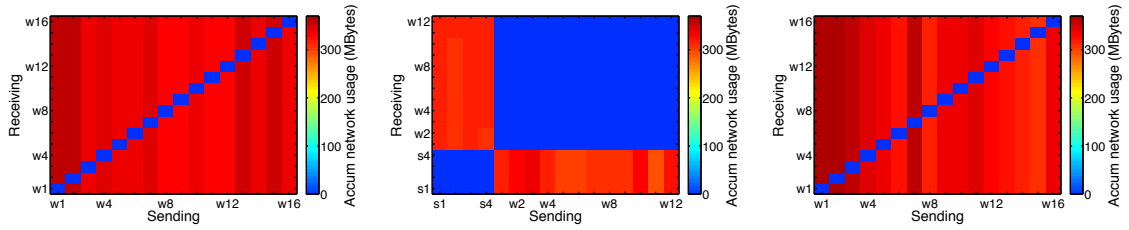
3.5.3 What is the statistical efficiency?

Next we assess how progressive epochs under partial gradient exchange contribute to the convergence for ImageNet on 64 machines. We define three validation accuracy goals (5%, 10% and 20%), and observe the number of epochs required to achieve them.

Fig. 3.8(a) shows that the PS approach requires the fewest passes over the training data for a given accuracy when the number of server nodes is high enough for the centralised parameter synchronisation: 16 servers for 48 workers in PS*[48+16], and 64 servers for 64 workers in the co-location case (PS[64+64]).

Ako requires extra epochs for the same accuracy, making it less statistically efficient than PS—gradients are partitioned before exchange, which means that workers receive incomplete gradients but with low latency; only after multiple rounds, they obtain complete gradients.

With too few parameter servers (PS[56+8]), the efficiency of the parameter server approach declines and becomes the same as that of All-to-All, which suffers from diverging models due to insufficient synchronisation.



(a) Ako (b) PS (c) All-to-All

Figure 3.9: Average network usage with 16 machines (ImageNet)

3.5.4 What is the hardware efficiency?

Given that Ako trains the DNN model for ImageNet faster than PS*, the best distributed parameter server configuration, while having lower statistical efficiency, must have higher hardware efficiency. To confirm this, we collect the time per epoch across the approaches together with their break-down into (i) gradient computation and (ii) model synchronisation time.

Fig. 3.8(b) shows that Ako has a shorter epoch time than PS, regardless of its configuration; as expected, the decentralised All-to-All approach takes the longest time.

All approaches except All-to-All spend most of their epoch time on gradient computation. Compared to the other approaches, the optimal PS*[48+16] configuration requires longer to do this because it has only 48 workers for processing. In fact, its gradient computation time alone exceeds Ako's overall epoch time.

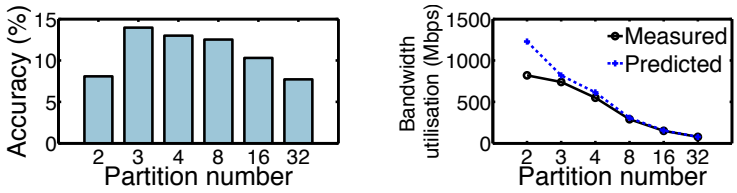
Comparing PS* to the co-location configuration (PS[64+64]), the later takes more time for the model synchronisation. This is due to the fact that its workers must synchronise with all 64 machines to obtain a new version of the model, prolonging the epoch time compared to PS*[48+16].

3.5.5 What is the resource utilisation?

We now investigate how Ako utilises the CPU resources and network bandwidth of the cluster compared to the other approaches. We deploy the ImageNet DNN model with Ako on our 16-machine cluster, and compare to the best PS*[12+4] configuration and the All-to-All approach. We measure the average CPU utilisation across the whole cluster and the accumulated network bandwidth usage between all machine pairs.

The average CPU usage of the workers for Ako, PS*[12+4] and All-to-All is 87%, 84% and 85%, respectively; the distributed parameter servers have an average utilisation of 17%, underutilising their CPU resources. Updating the global model parameters does not require as much CPU resources as the matrix multiplications and convolutions performed by the workers.

Fig. 3.9 shows the accumulated network usage in MBs. For Ako, the network usage between all machine pairs is high, fully saturating the network while still achieving a low synchronisation delay. All-to-All also fully saturates the network, but suffers from a high delay due to the introduced queuing.



(a) Accuracy (b) Network usage

Figure 3.10: Effect of gradient partitions

PS* has a much lower network usage, with substantial network bandwidth between workers left unused. The peak network usage of Ako is lower than that of All-to-All because it partitions gradients before exchange according to its cost model. Ako's workers exchange gradients as frequently as possible while respecting the capacity limit of the network.

3.5.6 What is the effect of gradient partitions?

Next we investigate the effectiveness of how Ako chooses the number of gradient partitions according to its cost model (§3.3.2). We execute the ImageNet DNN model with Ako on our 16-machine cluster across different partition numbers, and measure the validation accuracy and the workers' bandwidth usage.

Fig. 3.10(a) shows that 3 gradient partitions results in the highest accuracy, which is the partition number predicted by the cost model. With only 2 partions, the partition size becomes larger. This is beneficial for the learning progress because it improves statistical efficiency—it takes fewer rounds to exchange the complete gradient. However, the gradient exchange has higher latency due to network contention, which increases the divergence of the local models during training. Having more than 3 partitions also reduces the accuracy because the information exchange between the workers becomes less effective, which reduces the statistical efficiency. In this experiment, the partition number of 3 keeps staleness as low as possible while maximising network resource utilisation without suffering from high transmission delay.

Fig. 3.10(b) compares the measured and predicted network usage according to the cost model. For almost all partition numbers, the measured bandwidth usage is close to the predicted one. The difference is largest with 2 partitions because the predicted usage by the model goes beyond the 1 Gbps bandwidth available in the cluster.

3.5.7 What is the benefit of gradient accumulation?

In partial gradient exchange, the accumulation of gradient updates ensures the eventual completeness of gradients sent to workers. We conduct an experiment to evaluate how this improves training quality on the ImageNet DNN model with 8 machines. We measure the validation accuracy of Ako with and without gradient accumulation.

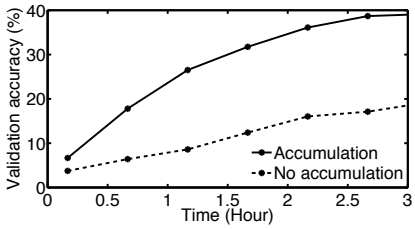


Figure 3.11: Benefit of gradient accumulation

Fig. 3.11 shows the convergence over time. Without accumulation, the resulting model exhibits worse accuracy over time with slower convergence: the best accuracy on validation is only around 20%, while gradient accumulation improves this to nearly 40%. If workers do not receive complete gradient updates, the statistical efficiency of the system is reduced.

3.6 Related work

DNN systems with parameter servers. Scalable deep neural network (DNN) systems, such as DistBelief [DCM+12], TensorFlow [AAB+16], Project Adam [CSAK14], Singa [OTW+15, WCD+15, WCC+16], Poseidon [ZHW+15] and SparkNet [MNSJ15], speed up the distributed training of DNN models using parameter servers. To avoid network bottleneck while synchronising multiple workers with a centralised global model, parameter servers are scaled to gain higher accumulative bandwidth [LAP+14].

TensorFlow [AAB+16] expresses DNNs as dataflow graphs that train under a parameter server architecture. For distributed parameter servers, TensorFlow uses a round-robin strategy that assigns different DNN layers to parameter server nodes. This assignment can lead to imbalances in the computational load and network utilisation among servers because the model is partitioned at a relatively coarse granularity. In addition, the scaling of parameter servers can be limited by the number of mutable tensors (i.e. variables) in graph.

While scaling parameter servers relieves network contention, there are further techniques to reduce network communication, including data compression [AAB+16, LAP+14] and filtering [LAP+14]. In Bösen [WDQ+15], messages are prioritised—ones that lead to more significant model progress are transmitted first. In exchange for more efficient network usage, such techniques, however, increase CPU utilisation.

Poseidon [ZHW+15] uses Bösen's parameter server architecture with a hybrid synchronisation model: workers establish direct connections to offload network communication from the parameter server. While Poseidon reduces the size of model updates by exchanging sufficient factors [XKZ+15] for dense fully-connected layers, its communication cost can grow quadratically with respect to the number of nodes. In general, distributed parameter servers can adopt a hybrid architecture to increase the accumulative bandwidth at the server side, but this further complicates resource allocation decisions.

Singa [OTW+15, WCC+16, WCD+15] is a deep learning platform that supports multiple partitioning and synchronisation schemes, thus enabling users to easily use different training regimes at scale. Through the concept of *logical groups* of execution units, Singa supports complex cluster topologies, e.g. one with multiple server groups. This flexibility allows users to tune configurations for given problems and the available cluster resources, but requires them to make configuration decisions empirically. Ako is a step towards removing some of this configuration complexity.

Rather than assigning servers and workers to different machine groups, Bösen [WDQ+15] uses collocation, i.e. each node contains both servers and workers. This type of allocation requires gradients to be aggregated to different model partitions across nodes before redistributing the updated model parts back to all workers. Although this approach maximises the aggregated network bandwidth for parameter servers, the two-step all-reduce procedure adds substantial delays when the system scales, even under bounded staleness.

DNN systems without parameter servers. Several DNN systems make use of decentralised training to simplify the deployment in distributed environments while maximising data parallelism. Since direct communication between nodes can grow quadratically when adding more workers, this requires scalable synchronisation strategies.

Wang et al. [WZGZ14] propose a decentralised protocol for parameter sharing with custom synchronisation topologies. Network contention is reduced by having a subset of workers relay gradients to the rest of the topology. As a result, model updates propagate more slowly, resulting in a higher convergence time.

Similarly, in MALT [LKKU15], workers exchange gradients with a subset of workers selected by a Halton sequence. Due to the synchronisation delay, this suffers from slower convergence, especially for complex neural network models. In contrast, workers in Ako always exchange partial model updates with all others.

In CNTK [SFD+14, FHJ+14], gradients are aggregated across all nodes, followed by model redistribution. To reduce the bandwidth requirement for densely-connected speech DNNs, gradient values are quantised aggressively before being exchanged over the network. To ensure model convergence, the resulting quantisation error must be added to the gradients in the following rounds, compensating for the inaccuracy. By representing gradients as 1-bit values, the work shows that there is little negative impact on model accuracy as the quantisation error feedback is treated as type of delayed update. In a similar fashion, delayed updates in Ako result from gradient partitioning.

Mariana [ZJL+14] synchronises multiple GPGPU workers in a linear topology. By sending gradients and model updates synchronously via adjacent workers, there is only minimal network contention, but this approach increases synchronisation delay due to the larger hop count.

Deep Image [WYS+15] uses a custom-built supercomputer with GPGPUs for DNN training. Workers are responsible for individual model partitions, and exchange updates through a butterfly network topology. In contrast, Ako does not partition the model, but instead performs full gradient synchronisation over multiple rounds.

3.7 Summary

To achieve the best performance, distributed DNN systems must fully utilise the cluster CPUs for model training and the cluster network bandwidth for model synchronisation. Today's DNN architectures, however, rely on distributed parameter servers, which makes it difficult for users to allocate cluster resources to them in an optimal way, i.e. without introducing compute or network bottlenecks.

We described Ako, a decentralised DNN system that does not require parameter servers, yet scales to large deployments with competitive performance. Ako achieves this through a new scalable synchronisation approach, *partial gradient exchange*, in which gradient updates propagate to all workers but only using constant bandwidth by sending partitions. We showed experimentally that an implementation of Ako with asynchronous compute and network tasks has better performance on a fixed-sized cluster than one with parameter servers.

Chapter 4

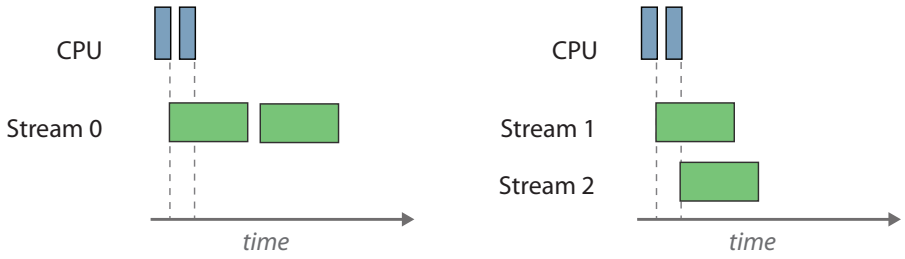
Efficient Execution of DNN Training on GPUs

4.1 Overview

GPUs are capable of fast matrix computation and thus have been widely adopted for DNN training. However, training most state-of-the-art DNNs on GPUs still takes a considerable amount of time. Therefore, high compute efficiency of GPUs is essential because it gives opportunities to further cut down the training time, given the same compute resource budget.

Most of the time, DNN computation consists of a series of matrix operations with different degrees of computational complexity. In consequence, some operations may underutilise GPU multiprocessors. Here we argue that multiprocessors should be fully utilised to achieve best convergence speed possible. We present *Crossbow*, a DNN system that is designed to maximise GPU compute power by permitting *multiple model replicas* to execute in parallel on the same device. By doing this, more than one matrix operation can be performed concurrently. Training with multiple models imposes two immediate requirements: (i) more model replicas demand more memory resources; and (ii) model synchronisation among replicas must have low latency to maintain high compute utilisation. In this chapter, we describe how multiple models per GPU are realised in Crossbow and how the two requirements are addressed to facilitate the execution of multiple replicas.

This chapter starts with the discussion of the mechanisms underpinning GPU concurrency and parallelism. We identify opportunities to further leverage GPU concurrency beyond traditional multi-threaded executions. We then explain how to make use of a GPU parallel architecture to execute multiple model replicas on the same device. To handle the increasing demand on memory, we describe our memory reuse technique that promotes efficiency in memory usage. We give details on the design of technique and an algorithm that creates an offline memory allocation plan to avoid runtime decisions. We also introduce a hierarchical model for synchronisation that results in low-latency data transfer when synchronising replicas across different devices. We finish the chapter with experiments on convergence and scalability, in comparing to other popular DNN systems. We also demonstrate the



(a) Using the default stream (b) Using multiple per-thread streams

Figure 4.1: CPU host launches two tasks (blue boxes), each of which contains a kernel (green box) under two scenarios

benefit of training multiple concurrent model replicas over an alternative approach that is commonly used.

4.2 Concurrency in GPUs

Programming an application on a GPU relies on a heterogeneous programming model: it involves both the CPU and GPU, often referred as the *host* and *device*, respectively. The application code runs on the host and launches *kernels* which are functions executed on the device. A DNN layer is comprised of one or more kernels. Combining all layers in the model together, we effectively have a series of kernels that are sent to the device for execution. Here we refer to such a series of kernels as a *task*. When training a model with the mini-batch learning method, a task is essentially the computation of a mini-batch. A training epoch has as many tasks as the number of mini-batches.

A task is executed on the device kernel by kernel, and there is no overlap between kernels. Based on SIMD, a kernel is executed by many multiprocessors, and an application thus benefits from kernel-level concurrency. In the cuDNN library, kernels are highly optimised, and kernel concurrency is encapsulated by the library. The functions within a kernel dictates how multiprocessors are utilised. Since one kernel in the task is executed at a time, GPU utilisation varies as the computation progresses.

A series of kernels is effectively a sequence of operations to be execute on the device. The order of kernels depends on how the application code is run on the host. In CUDA, such sequence is implemented as a *stream*, a queue of device work. Whenever multiprocessors are available, a kernel will be scheduled from the queue. Since kernels in the stream are ordered in a FIFO manner, the order of execution in the stream is guaranteed according to the prescribed order and thus cannot overlap. In older GPU generations, a device has only one *default stream* (or stream 0) shared across host threads. Although there exist multiple non-default streams, two tasks cannot run concurrently if one of host threads issues a kernel to the default stream, as illustrated in Fig. 4.1(a). This effectively serialises the execution of kernels and, as a result, applications can only benefit from kernel-level concurrency.

With new hardware generations, a device can have *many* default streams. This new functionality is called *per-thread default streams*: each host thread can now have its own default stream. This means that tasks in different streams can run concurrently, as shown in Fig. 4.1(b). While kernels within the same default stream are ordered, those from different ones are independent and can overlap. This multi-stream functionality opens up a new opportunity to leverage GPU compute power as applications can exploit both kernel- and stream-level concurrency.

4.3 Leveraging concurrency to maximise GPU compute performance

In this section, we describe our idea of using multiple GPU streams to speed up DNN computation beyond the traditional multi-threaded execution when there is room to fit more computation. We discuss two types of parallelisation strategies that can be realised through the multi-stream functionality. Next, we describe our memory reuse technique whose goal is to increase the efficiency in memory usage and minimise the chance of physical memory capacity limiting the execution of multiple replicas. Towards the end of this section, we discuss our hierarchical synchronisation approach that results in low-latency data transfer when synchronising replicas across devices.

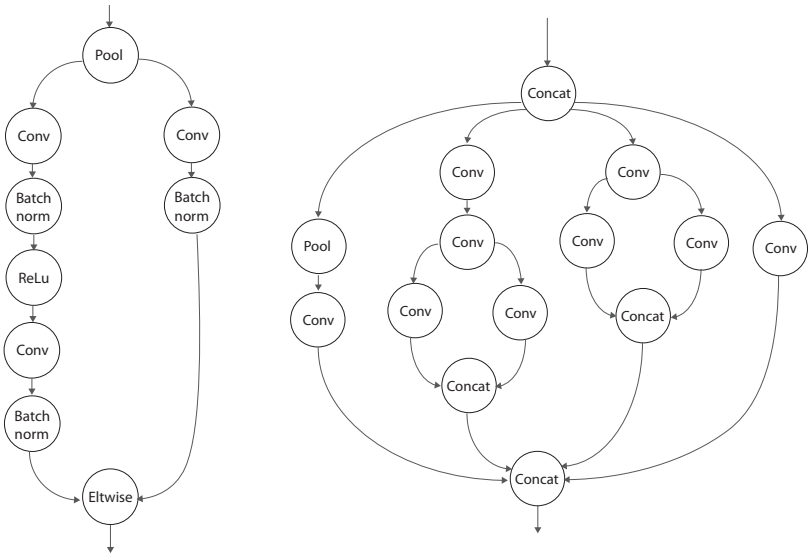
4.3.1 Parallel execution using multiple streams

With multiple streams, a GPU can schedule multiple kernels concurrently as long as there are multiprocessors available. Concurrency is necessary for speeding up a long-running job such as DNN training. Here we discuss two distinct, yet complementary, ideas that attempt to maximise GPU compute performance through multi-stream concurrency: (i) implementing *data parallelism* using multiple GPU streams; and (ii) taking advantage of the multi-branch model architecture in most state-of-the-art DNNs and executing branches concurrently with *model parallelism*.

Data parallelism using multiple replicas

To make the best use of GPU concurrency, we propose to execute *multiple concurrent tasks* through multiple streams. When a single task does not fully utilise the multiprocessors, the idle cores can be used for other computation, in this case, other tasks. With queues from multiple streams, the GPU scheduler is always supplied with work ready to be executed. Since tasks from different queues have no data dependencies, their kernels can be executed independently. By doing this, we increase hardware utilisation and increase the system throughput as epoch time decreases. In consequence, the training progresses faster.

To support the execution of multiple concurrent tasks, we associate each task with a *model replica* to carry out the computation of a training mini-batch. As a result, we have multiple model replicas executing many tasks in parallel. Since different tasks are associated with different batches of training examples, we effectively exploit data parallelism. However, our approach is distinct from that employed in most DNN systems which mainly apply data parallelism across devices.



(a) ResNet (b) Inception-v3

Figure 4.2: Multi-branch model architecture for a building block of the ResNet and Inception-v3 models

Model parallelism for a multi-branch DNN architecture

To capture complex representations in the raw data, many state-of-the-art DNNs make use of a complex network architecture: the model is no longer a simple chain of layers but contains multiple branches with different computation. In Fig. 4.2, we show examples of a multi-branch architecture in two well-known DNN models; ResNet [HZRS15a] and Inception-v3 [SLJ+15]. In these examples, more than one computation path may branch out from a common layer and eventually merge back. This pattern often repeats several times in many state-of-the-art networks to increase representational complexity [HZRS15a, SLJ+15, SIV16]. At runtime, different branches can be executed independently because there is no data dependency between them. We consider this an opportunity to apply model parallelism to further improve GPU concurrency. Therefore, we propose to make use of multiple streams to issue kernels from different branches in the same model. Since kernels in different streams may run concurrently and interleave, the multiprocessors can continuously be kept busy. The limitation of this approach is that the DNN architecture will have a direct influence on GPU utilisation. However, we consider it beneficial to make use of this approach together with data parallelism: while multiple tasks are placed on multiple streams, kernels from different branches in a task are also placed on other additional streams.

4.3.2 Efficient memory utilisation to support multiple replicas

Since running multiple concurrent tasks on the same device demands more memory, the physical memory capacity can put a constraint on the number of model replicas and limit the parallelisation.

DNN model	Dataset	Batch size	Memory usage for model parameters	Memory usage for outputs of all the layers	Total memory usage
AlexNet	ImageNet	64	0.22 GB	0.75 GB	0.97 GB
		128	0.22 GB	1.50 GB	1.72 GB
		256	0.22 GB	2.99 GB	3.21 GB
Inception-v3	ImageNet	16	0.08 GB	4.45 GB	4.53 GB
		32	0.08 GB	8.90 GB	8.98 GB
		64	0.08 GB	17.80 GB	17.88 GB
ResNet-18	ImageNet	64	0.04 GB	4.89 GB	4.93 GB
		128	0.04 GB	9.78 GB	9.82 GB
		256	0.04 GB	19.55 GB	19.59 GB
ResNet-34	ImageNet	64	0.08 GB	7.51 GB	7.59 GB
		128	0.08 GB	15.01 GB	15.09 GB
		256	0.08 GB	30.03 GB	30.11 GB
ConvNet	Cifar	64	0.87 MB	43.79 MB	44.66 MB
		128	0.87 MB	87.58 MB	88.45 MB
		256	0.87 MB	175.15 MB	176.02 MB
ResNet-32	Cifar	64	1.79 MB	619.63 MB	621.42 MB
		128	1.79 MB	1239.04 MB	1240.83 MB
		256	1.79 MB	2478.08 MB	2479.87 MB
ResNet-56	Cifar	64	3.27 MB	1064.96 MB	1068.23 MB
		128	3.27 MB	2140.16 MB	2143.43 MB
		256	3.27 MB	4270 MB	4273.35 MB

Table 4.1: Analysis of the memory usage across different DNN applications

In this section, we report findings from our study of the memory requirements of some well-known DNN models that solve image classification problems. In particular, we investigate where the memory requirement originates from. By understanding the characteristics of memory consumption, we identify opportunities to utilise the available memory in a more efficient way. At the end of the section, we describe our approach for achieving efficient memory utilisation.

Memory requirement for DNN training

To understand the memory requirement of DNN training, we conduct a study in which we observe memory usage of various DNNs when training on a GPU. We use four DNN models, AlexNet, ResNet variant, Inception-v3, and ConvNet, each of which trains with three mini-batch sizes. We collect information on the amount of memory used for storing (i) model parameters; and (ii) input and output data in all the layers of the model. Since the output of a layer is the input of the successive layer, we only take into account the memory usage of layer's output for our analysis.

Table 4.1 shows the memory utilisation across different models and batch sizes. The first important finding is that most memory usage originates from layer outputs and is dependent on the batch size. In contrast, model parameters take up only a small fraction except for AlexNet which is slightly larger in size compared to the other DNNs for solving the same ImageNet problem. The second

Layer type	Need to retain input data	Need to retain output data
Convolution	Yes	No
ReLU	Yes	Yes
Batch normalisation	Yes	Yes
Pooling	No	No
Inner product	Yes	No
Softmax	No	No
Softmax loss	Yes	Yes
Elementwise	Yes	Yes

Table 4.2: Analysis of whether the memory allocated for storing the input and output of different layer types can be overwritten as task computation progresses

finding is that a deeper network architecture utilises more memory. This is because the total amount of layer's output data grows linearly with the depth of the model. State-of-the-art models are likely to need more memory due to the increase in model depth, e.g. comparing AlexNet [KSH12] with ResNet-34 [HZRS15a], which were proposed in 2012 and 2015, respectively.

Improving efficiency with memory reuse

Training a DNN model with a deep complex architecture is a compute- and memory-intensive job. To maximise the GPU processing speed-up, one should ensure that the physical memory is not the bottleneck. Since the rate at which the memory sizes of GPUs increase over GPU generations is slower than that of the GPU compute performance [Wik17], the efficiency in utilising memory resource is important. In this section, we discuss our approach for improving memory resource efficiency. We first look at how the backpropagation algorithm operates and its implication in terms of memory requirements.

According to the backpropagation algorithm discussed in §2.1, gradients are computed based on the propagated error with respect to model parameters and layer input. Hence, the input data of a layer is often needed for the computation in the backward phase. However, there are certain types of DNN layers that do not have model parameters because they only perform a data transformation. In this case, it may be possible that the memory allocated for storing the input and output data for these layers can be safely overwritten without affecting correctness. It is feasible to allow an overwrite if we can verify that the data will not be used again in the backward phase: it is not needed for the gradient computation. Therefore, it is safe to *reuse* such allocated memory for other purposes.

We carry out an analysis of whether a layer requires the input and output data to be retained for the backward computation. We present the result of our analysis on a layer basis in Table 4.2. If a layer does not need to retain its input and output data, the allocated memory for the data can be reused for other purposes. For example, a convolutional layer needs to keep its input data and thus does not allow for reuse. However, it is fine for its output data to be overwritten. Input and output data are usually maintained in some kind of data buffers that the layer reads from and writes to. Since the output data of a layer is the input of the next, the data buffer is shared by the two layers. If the

preceding layer allows the output data buffer to be reused and the next layer also does not need to retain its input data, the data buffer can be reused safely. Otherwise, the data in the buffer must be kept until no one wants to retain it anymore. For example, if a convolutional layer is followed by a pooling layer, the output data buffer of the convolutional layer can be safely reused because neither convolution's output nor pooling's input needs to be retained.

Based on the information from Table 4.2, we develop an algorithm that allows us to pre-plan the memory allocation for data buffers. We describe this in detail in the following section.

Algorithm 4: Construct a plan for memory allocation for the output of each layer

```
1  Define : Pool of buffer pool,
2  Plan for memory allocation plan
3  for each layer l in model do
4      if forward phase then
5          size = l.getOutputSize();
6          buffer = findReusableMemoryBuffer (pool, size)
7          if no reusable buffer in the pool then
8              /* Allocate new memory for the layer's output */
9              buffer = createNewBuffer(size);
10             registerToThePool(buffer, pool);
11             updateMemoryAllocationPlan(plan, l, "new");
12         else
13             /* Use a reusable buffer from the pool */
14             owner = buffer.getOwner();
15             updateMemoryAllocationPlan(plan, l, owner);
16         /* Check if layer's output can be overwritten */
17         if output not allowed to be overwritten then
18             incrementReferenceCounter(buffer);
19         /* Check if layer's input can be overwritten */
20         if input not allowed to be overwritten then
21             /* Find the preceding layers and increment their output's reference counter */
22             prec = getAllPrecedingLayers();
23             for each layer pl in prec do
24                 incrementReferenceCounter(pl.getOutputBuffer());
25     if backward phase then
26         /* Decrement the counter of output buffer */
27         if output not allow to be overwritten during forward phase then
28             decrementReferenceCounter(getOutputBuffer());
29         /* Decrement the counter of input buffer(s), if needed */
30         if input not allow to be written during forward phase then
31             prec = getAllPrecedingLayers();
32             for each layer pl in prec do
33                 decrementReferenceCounter(pl.getOutputBuffer());
```

Algorithm to pre-plan memory allocation and memory reuse

The goal of the algorithm is to construct a plan for memory allocation of all the data buffers required for model training. We use the information for Table 4.2 to decide if a data buffer is reusable. We also check if a layer can reuse a previously allocated buffer owned by another layer. The idea is that there is no need to allocate new memory for a data buffer if there is a buffer that has been created with a suitable size and is currently available for reuse. To indicate whether a buffer is reusable at a particular stage of execution, we make use of a reference counter to recognise its availability. The plan for memory allocation is constructed only once and used throughout the entire execution because training is iterative. since the static memory plan will be used over and over again until training ends, there is no runtime decision involved.

In Algorithm 4, we describe how to create a memory allocation plan. Since layers are chained together, the output buffer of a layer is the input buffer of the next layer. We associate a layer with an output data buffer, and the input buffer is simply the output buffer of the preceding layer. To reason about the reusability of a buffer, the buffer is associated with a reference counter whose value is non-negative. A zero counter indicates that no layer requires the data in the buffer to be retained while a positive integer indicates the number of layers requiring the data to be maintained for the backward computation. At the end of the algorithm, we should be able to tell whether a layer needs newly allocated memory for its output buffer or whether it can simply use an existing buffer from another layer that has been previously created.

In the algorithm, we iterate over all the layers in the same sequence as during actual runtime execution. For each layer, we first find the size of its output data. This is straightforward to calculate using the information about the input data and the arithmetic operations of the layer. Before opting to create a new data buffer for the output data, the algorithm checks for a reusable memory buffer (line 6) by consulting a global *pool of buffers*, which keeps track of information about all the buffers that have been created up to this point in the actual execution time. Valid buffers for reuse are those that have a reference counter of zero and a suitable buffer size, i.e. equal or greater than that of the output data. If the global pool suggests that there is no candidate that meets these two criteria, the layer needs to allocate a new memory for its output buffer at runtime (line 9).

Regardless of which data buffer a layer reuses to store its output, important step is to check the layer's requirement for keeping data for the backward computation (line 17). If so, we increment the reference counter of the buffer. Although the input buffer of a layer belongs to the preceding layer, it can always enforce its requirement on retaining the input data (line 20). If a layer has multiple inputs, it enforces its requirement to all input data buffers (line 23).

When the data in a buffer is no longer needed, i.e. the computation in the backward phase for the layer finishes, its reference counter is decremented to reflect the change in its reusability status (line 28 and 30). When the counter reaches zero, the buffer becomes reusable. This is useful for the computation in the backward phase (just like the forward phase) because the backpropagation also needs data buffers to propagate the error across layers.

By running this algorithm for a given DNN model, we obtain a plan that describes the details of how the data buffers in each layer should be handled at runtime: either created first or simply reused. The more data buffers can be reused, the less memory needs to be allocated, bringing better memory efficiency.

4.3.3 Synchronisation among multiple model replicas

Training with multiple model replicas imposes the need for synchronisation to ensure overall convergence. Since these replicas are colocated on the same device, we synchronise them under the *bulk synchronous parallel* (BSP) approach. As described in §4.3.1, a replica is associated with a task and a batch of training examples at a time. Due to the synchronisation barrier enforced by BSP, replicas wait for each other to finish the current tasks before moving on to the next training batch. Replicas exchange their model parameters and incorporate all the weight changes from one another.

The idea of using multiple replicas is also applied to a setting in which there are multiple devices on the same machine. While synchronising replicas on the same device is straightforward, exchanging model parameters across devices incurs data transfers that result in latency and waiting time for multiprocessors. To keep the latency low, we adopt a *hierarchical model synchronisation* to facilitate an efficient training on GPUs. The hierarchical synchronisation is done by using tree aggregation: replicas on the same device exchange their model parameters before the in-device synchronised model is used to synchronise across different devices. By doing this, the amount of data transfer is proportional only to the number of devices, not the total number of replicas.

Model parameters from different replicas can be mixed in various ways during synchronisation. One common technique is to collect all the gradients from the replicas and use the averaged gradient to update the model parameters of all replicas. Due to the gradient averaging operation, the gradient is essentially computed based on a large training batch whose compute workload is distributed across replicas. In other words, if we have n replicas each of which is given a batch of b training examples, the effective batch size in the system will become $n \times b$ because the gradients are averaged. While a large batch size improves GPU concurrency, the side effect is that it can slow down convergence speed. Alternatively, we can employ other synchronisation algorithms, e.g. model averaging [ZCL15], which maintain the effective batch size as that given to an individual replica, i.e. b rather than $n \times b$.

To maximise the utilisation of multiprocessors, relaxed synchronisation can be applied: replicas are permitted to execute more than one task before the synchronisation barrier is triggered. When a task has been processed, the replica updates its model parameters according to the local gradient and moves on to the next task independent of the other replicas. Only when it is time to synchronise, replicas are required to exchange their current model state and incorporate the change in weight parameters from each other. By permitting relaxed synchronisation, we can further improve the hardware efficiency especially when training across multiple devices. The multiprocessors can spend more time on data processing and less time on waiting for data transfers that occur periodically.

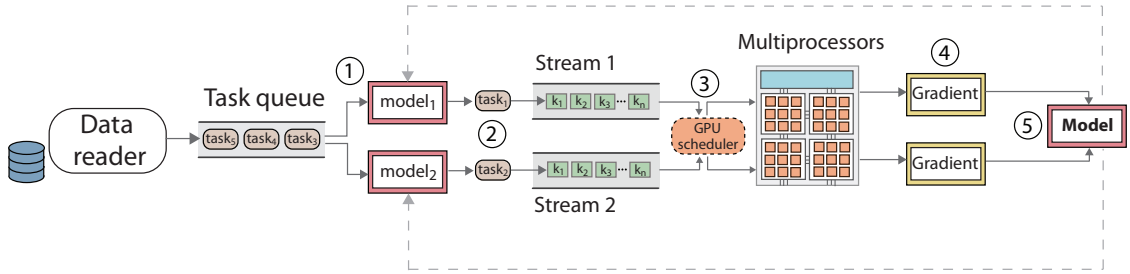


Figure 4.3: System architecture in Crossbow

4.4 System architecture of Crossbow

We present Crossbow, a high-performance DNN system that is designed to maximise the compute power of multiprocessors and efficiently train models on GPUs. We first summarise Crossbow's design goals in §4.4.1 and then give implementation details in §4.4.2.

4.4.1 Design goals

(1) Maximise the utilisation of processing resources. We want to achieve high hardware efficiency by minimising multiprocessor idle times and translate this into better convergence speeds. Fig. 4.3 shows how we make use of multi-stream concurrency to enable data-parallel training with multiple model replicas and thus achieve better GPU concurrency.

(2) Efficient memory usage. The parallel execution of multiple model replicas increases memory utilisation while maintaining model parameters and intermediate results during training. However, data stored in some memory allocations is not needed for the backward phase, allowing its allocated memory to be reused for storing other intermediate results. As a result, the total amount of memory allocation for a task decreases, which gives an opportunity to fit more model replicas for better data parallelism.

(3) Synchronise models across devices with low latency. Since the training process of DNN models can be further parallelised due to the compute capability offered by multiple GPUs, we do not want to waste multiprocessors waiting for the data transfers that occur during model synchronisation. To keep the latency between devices as low as possible, model replicas within the same device must merge their model parameters before they do so across devices.

4.4.2 Architecture and implementation

In this subsection, we describe the implementation of Crossbow. We first give an overview of the data processing workflow within Crossbow and explain how training with multiple replicas can be realised with multiple GPU streams. We then describe how Crossbow efficiently manages memory

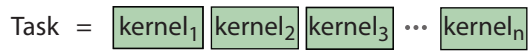


Figure 4.4: A task is a sequence of GPU kernels

usage during execution time. Finally, we detail how Crossbow performs model synchronisation and model updates across different replicas and devices to ensure overall convergence.

Overview of the data processing workflow

Crossbow makes use of GPU streams to supply multiple tasks in parallel to the multiprocessors. A task is a sequence of GPU kernels describing the computation that happens over a batch of training examples. As illustrated in Fig. 4.4, a task is composed of a series of different GPU kernels which are scheduled for execution serially. In Crossbow, a model replica takes one task at a time to execute, shown in step 1 in Fig. 4.3. With multiple GPU streams, several tasks can be issued to the device in parallel. In other words, Crossbow is capable of exploiting data parallelism by using multiple GPU streams to train multiple replicas on the same device while other systems such as TensorFlow [AAB+16] executes one replica per device.

In an attempt to minimise the delay in dispatching tasks to streams, Crossbow relies on a set of *task dispatchers* each of which takes a task from the *task queue*, associates it with a model replica, and dispatches to a corresponding stream (step 2). Since kernels within a stream are arranged in order, they are scheduled by the GPU schedulers one by one. With multiple streams, more than one kernel can be scheduled concurrently as long as there are available multiprocessors at scheduling time (step 3). The number of streams is predefined as part of the application and is set to at least as many as the number of model replicas.

When a task has finished, gradients are computed and used to update the corresponding model replica (step 4). If it is time for model synchronisation, the changes in model parameters in all the replicas are aggregated and incorporated into the *base model* that maintains the shared model parameters (step 5). Further details will be discussed in §4.4.2. Before the next set of tasks are issued to streams, replicas fetch the latest version of the model parameters from the base model and proceed with the next task. This workflow is carried out repeatedly until convergence is reached.

Tasks in Crossbow

A task describes the computation sequence within an iteration during model training. In Crossbow, the computation of a model is expressed as a directed acyclic graph (DAG) where each vertex represents a layer's computation and each edge represents the connection between two layers. Fig. 4.5 shows an example DAG for the computation of ConvNet (described in §5). A layer's computation is composed of at least one arithmetic operation which is executed on the device as a GPU kernel. In Crossbow, we

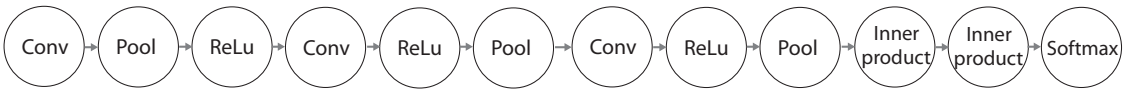


Figure 4.5: A DAG representing the ConvNet model

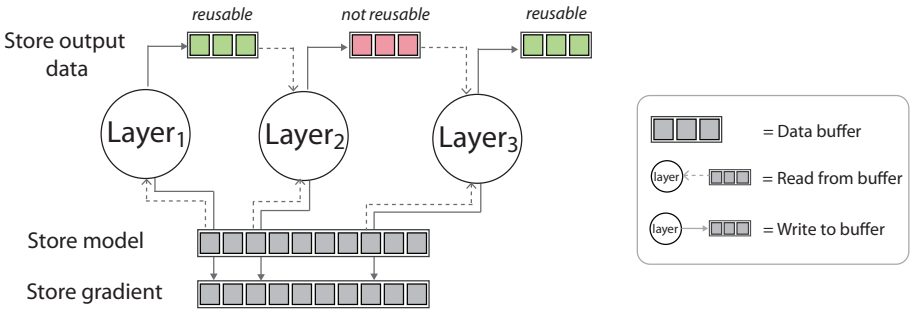


Figure 4.6: Memory utilisation of a task in Crossbow

make use of the cuDNN library [CWV+14]. It provides support for several common layers through CUDA function calls that encapsulate a set of GPU kernels. Before being dispatched to a stream, a DAG is transformed into a sequence of kernels by chaining all the kernels from all the layers together.

In many state-of-the-art DNN models such as ResNet and Inception-v3, the DAG is not a simple chain but rather contains a multi-branch structure whose purpose is to enhance the capability in data representation. In this case, such a complex DAG is transformed into a sequence of execution, by applying a *topological sort* to generate a new DAG with a simple chain. Later, a sequence of kernels is obtained and dispatched to a GPU stream.

Although the topological sort simplifies the task dispatching process to a GPU stream, the multi-branch structure introduces an opportunity to enhance GPU concurrency further. Since branches have no data dependencies between them, we have many sequences of kernels from different branches and dispatch each sequence to a different stream in parallel. By taking into account such branching structures, a task can be executed across multiple streams. A similar strategy has been employed in TensorFlow [AAB+16]: multiple streams and stream dependency primitives are used to parallelise the computation within a dataflow graph when parallel execution is possible.

According to the system design, the definition of a task is decoupled from that of a model replica and training data, although they must be associated with one another to enable execution. This is because the computation in iterations is the same throughout the training process while the state of model replicas and the batches of data constantly change: a task with a different model replica and training batch will produce different model gradients.

Layer ID	Output data buffer
0	new
1	new
2	1
3	1
4	new
5	0

Table 4.3: An example of a memory allocation plan

Memory allocation and management

The training process requires memory for two purposes: (i) maintaining the current state of model parameters and gradients, (ii) storing the output data of DNN layers. Crossbow allocates memory and holds data in *data buffers*. Fig. 4.6 illustrates how data buffers are used to store the model parameters, gradients, and output data. Within a task, each layer is associated with a data buffer that maintains its output data. Our implementation only allows a layer to produce a single output. However, a layer may read data from many data buffers that belong to the preceding layers. If the computation needs to read the model parameters or write the gradients, the layer will access the data buffers of the model replica that the task is associated with.

In §4.3.2, we discuss that a layer's output data takes up the majority of memory. We emphasised the opportunity to reduce the total amount of memory requirement by reusing the allocated data buffers. With Algorithm 4, we can construct an offline memory allocation plan. The plan gives information about whether a layer needs new memory allocated for its output data or can reuse an existing buffer created by a preceding layer.

In Table 4.3, we show an example of a memory allocation plan. There are six layers in the model, and each layer is assigned a unique index. If the plan indicates `new`, the layer needs a newly allocated memory for its output data buffer; otherwise, it reuses the buffer created by the layer whose the index is specified in the plan. For example, the layers with IDs of 2 and 3 can reuse the data buffer created by the layer with ID 1. The memory plan is static and used throughout the training until convergence.

While output data buffers are associated with layers in a task, the buffers holding the model parameters and gradients are coupled to the model replica. In Crossbow, we store model parameters in a single buffer, and the same applies to model gradients. With a single contiguous buffer, the model update operation can benefit from multi-threaded parallelism when the kernel of model updates is executed. When a layer wants to access a particular part of the model parameters, it must use the correct offset to specify the position in the buffer. Since it is known in advance which portion of the model parameters are needed by a layer, the buffer is read in bulk, given the offset and the size of the model parameters.

Model synchronisation

Crossbow trains a DNN by executing multiple model replicas in parallel on the different parts of the training dataset. While data parallelism can speed up the training process, the replicas must periodically synchronise the model parameters with one another to prevent model divergence. To facilitate the synchronisation procedure, Crossbow has a *base model* that maintains the shared model parameters accessed by all the replicas on the same device. The purpose of the base model is to serve as the synchronisation point. It is not associated with any batch of training data for data processing. Therefore, the base model only requires memory for storing the model parameters and gradients, not output data of layers. Crossbow supports a wide range of model synchronisation techniques via the base model, e.g. the gradient accumulation [DCM+12, HCC+13] and the model averaging [MMS+09, ZCL15].

To increase compute efficiency, Crossbow supports relaxed synchronisation under BSP by allowing replicas to execute more than one task in sequence before they synchronise. The number of tasks that each replica executes before the synchronisation barrier is called the *work per clock (WPC)*. The example in Fig. 4.7 shows three replicas that execute two tasks before they synchronise. This means that the work per clock is six because the barrier is triggered every six executed tasks.

In Crossbow, it is possible to train multiple replicas across multiple GPUs. To synchronise them across devices, we adopt tree aggregation to perform hierarchical model synchronisation: we synchronise the replicas within the same device via the base model first and then use the base models from different devices for the synchronisation across devices. We do not have an extra base model for the cross-device synchronisation but rather select one base model to perform the aggregation of model parameters from all the devices, e.g. the one on device ID 0. By minimising the amount of data transfer during the synchronisation across devices, we can minimise the latency to mitigate the synchronisation bottleneck.

Support for model training on CPUs

Crossbow also supports model training on the CPU. Some machine learning algorithms, including small-sized DNN models that have low computational workload, still benefit from CPU execution. With multiple CPU cores, we can take advantage of data parallelism by running concurrent tasks across cores. In our CPU implementation, multiple tasks can be associated with the same model replica but executed in parallel. In this case, multiple concurrent reads of the model parameters are permitted but updates to models are done sequentially. By doing this, we can effectively operate the training directly on the base model.

4.5 Evaluation

The goals of our experimental evaluation are: (i) to investigate Crossbow's convergence and scalability compared to existing DNN systems (§4.5.2); (ii) to study the impact of multiple model replicas

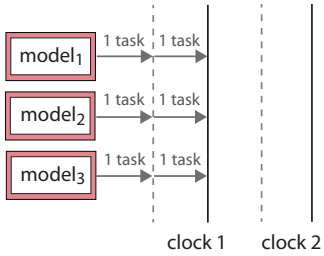


Figure 4.7: With WPC of two, each replica executes two tasks before triggering the synchronisation barrier

on the hardware and statistical efficiency (§4.5.3); (iii) to show how the number of replicas can be selected (§4.5.4); and (iv) to evaluate the effect of the memory reuse strategy in Crossbow (§4.5.5).

4.5.1 Experimental set-up

Training datasets and DNN models. We use three well-known datasets for image classification: (i) MNIST [LBBH98] contains 28×28-pixel grey-scale images of handwritten digits with 10 classes. The dataset has 60,000 training images, and 10,000 images for testing, (ii) Cifar [Kri09] has 32×32-pixel colour images of 10 different classes of objects and animals. The dataset has 50,000 training images and 10,000 images for testing. Each class contains 6,000 images and classes are mutually exclusive: there is no overlap between different classifications, (iii) ImageNet [RDS+15b] has over a million high-resolution images divided into 1000 classes, each with a ground truth label. We use the ImageNet dataset to investigate the effect of Crossbow's memory reuse technique.

We use the MNIST dataset to train the LeNet model which consists of 2 convolutional, 2 max-pooling, and 2 fully-connected layers with an `ReLu` and a `Dropout` layer. We use the same model configuration and learning hyperparameters as described in prior work [Ten17]. For this application, we train the model with the same number of epochs and use a batch size of 64.

With the Cifar dataset, we train ResNet applications with 32, 56, and 110 layers. For the model configurations and learning hyperparameters, we replicate the experimental set-up according to the original paper [HZRS15a] except that (i) we use the entire training dataset for the model training and do not perform the 45k/5k training/validation split; and (ii) we use a fixed learning rate of 0.1 in most of the experiments rather than applying a fixed learning rate decay scheme, which is often defined according to a particular batch size and the number of training epochs.

We use the ImageNet dataset to evaluate the memory reuse technique in Crossbow. We analyse the memory usage required to operate the training for the ResNet and Inception-v3 models. With the ResNet model, we use four variants with 18, 34, 50, and 101 layers with a batch size of 128. The model configurations are as described in [HZRS15a]. For Inception-v3, we use a batch size of 32 and the model configuration is based on the description in [SLJ+15].

System comparisons. We compare the performance of (i) Crossbow with multiple model replicas to (ii) MXNet and (iii) TensorFlow. For each DNN model, we make sure that all the model configurations and training hyperparameters are defined in the same way across different systems. While the other systems run one model per device, Crossbow executes multiple replicas on a device as long as the memory allows. If we run one replica on a single GPU with Crossbow, we effectively execute a basic sequential run.

To synchronise replicas in Crossbow, we use the model averaging approach proposed in [ZCL15] in our experiments. The reason is that we can preserve the original batch size used in the sequential execution i.e. we do not need to divide a batch of training data into several sub-batches before assigning to the replicas. In multi-device experiments, Crossbow's replicas from all the devices are synchronised under the same synchronisation approach. For MXNet and TensorFlow, the batch of data is split evenly across the available devices to ensure that the effective batch size remains unchanged.

Performance metrics. We validate DNN models based on the top-1 classification accuracy. We measure model performance in terms of convergence and observe training progress based on the misclassification error over time, on either training or testing data or both. To assess the statistical efficiency, we report convergence over epochs across the entire training session. To evaluate hardware efficiency, we observe the system throughput, which is measured as the total amount of training data in MB processed per second.

Cluster hardware. We use a machine with three NVIDIA Titan X Pascal GPUs with 12 GB of memory and a 10-core Xeon E5-2640 v4 CPUs at 2.40 Ghz with 64 GB of RAM to run all the experiments.

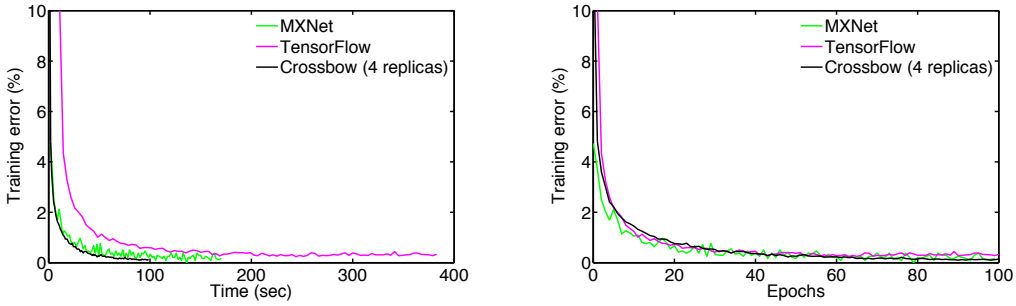
4.5.2 What is the convergence and scalability?

We evaluate the convergence speed of Crossbow in comparison to other systems when training the LeNet and ResNet (32 layers) models on the MNIST and Cifar datasets, respectively. We also explore the scalability by training the ResNet model on multiple GPUs.

LeNet model

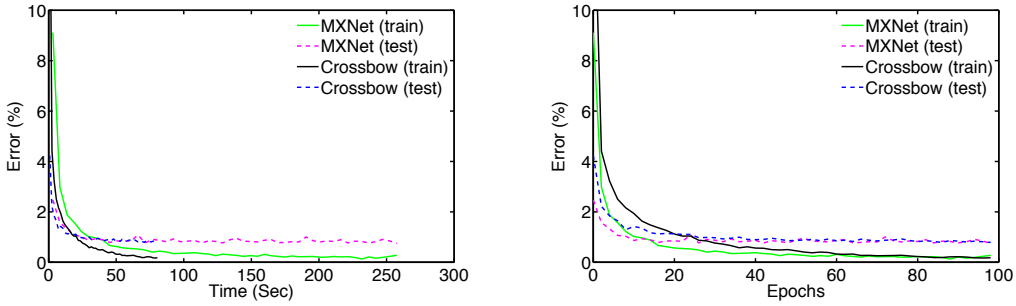
We train LeNet with the MNIST dataset across three systems including Crossbow, MXNet, and TensorFlow, on a single GPU. We fix the number of training epochs to 100 and report the convergence based on the training error over time. To compare the statistical efficiency among different systems, we also report convergence over epochs. While the other two systems execute training sequentially, Crossbow trains four model replicas in parallel in this experiment.

With the same number of epochs and training workload, Fig. 4.8(a) shows that Crossbow finishes the training the fastest: Crossbow takes 98 seconds whereas MXNet and TensorFlow take 170 and 382 seconds, respectively. In other words, Crossbow exhibits 1.7x and 3.9x speedup over MXNet and TensorFlow, respectively. To show that the same model configuration is used across the systems,



(a) Convergence over time (b) Convergence over epoch

Figure 4.8: LeNet model running on one GPU



(a) Convergence over time (b) Convergence over epoch

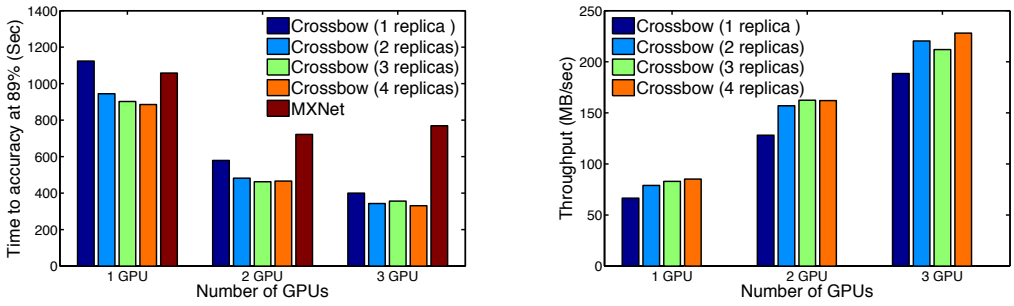
Figure 4.9: LeNet model running on two GPUs

we report the statistical efficiency. Fig. 4.8(b) confirms that the convergence per epoch behaves in a similar way. Crossbow finishes the training faster because it uses the available processing resource to do more computation. This is achieved by executing multiple model replicas with data parallelism.

Next, we run the model with the same training workload on Crossbow and MXNet with two GPUs. In this experiment, we observe the convergence based on both the training and testing error. Similar to the previous single-GPU experiment, we train 4 model replicas in total on Crossbow. Fig. 4.9(a) shows that it takes MXNet 260 seconds to finish the training while Crossbow takes 81 seconds, i.e. Crossbow achieves a 3.2x speedup compared to MXNet when running on multiple devices. In Fig. 4.9(b), the convergence from both training and testing reveals that the multi-replica approach requires extra epochs to reach the same error compared to sequential execution. However, since Crossbow can iterate over the training epochs faster due to better hardware efficiency, this improves statistical efficiency and eventually results in faster convergence.

ResNet model

Next we train the ResNet model with 32 layers, ResNet-32, on the Cifar dataset and vary the number of devices between one and three GPUs. In this experiment, we adopt the learning rate decay scheme as



(a) Time to reach testing accuracy at 89 %

(b) Throughput in Crossbow

Figure 4.10: ResNet-32 model running with Crossbow and MXNet

described in [HZRS15a]. We compare the performance of Crossbow to that of MXNet by measuring the training time required to reach a testing accuracy of 89 %. We collect the system throughput when the model is trained in Crossbow and report the average value in MB/sec. We also vary the number of model replicas per device between 1, 2, 3, and 4.

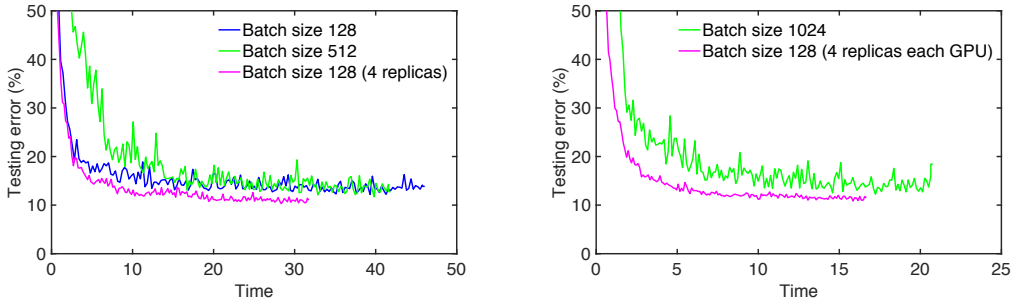
Fig. 4.10(a) shows that MXNet scales up to two GPUs while Crossbow scales to three GPUs regardless of the number of replicas. With multiple replicas, Crossbow takes a shorter time to reach the target accuracy compared to the training with one model replica. With three GPUs, Crossbow with 4 replicas achieves 2.3x speed-up compared to MXNet. In Fig. 4.10(b), for each number of GPUs, an execution with multiple replicas results in higher system throughput than that with a single replica. Also, we observe higher throughput when the number of GPUs increases, and this enables Crossbow to scale across multiple GPUs. These results altogether suggest that the improvement in hardware efficiency by training replicas in parallel is effectively translated into a better training speed-up.

While this experiment shows scalability across multiple devices within a single machine, higher system throughput can be obtained when training with multiple machines. Together with the low-latency hierarchical model synchronisation, good scalability should also be achieved in this setting.

4.5.3 What is the benefit of using multiple replicas over large batch sizes?

Since increasing training batch size has been a common approach to improve the GPU hardware efficiency, we study the convergence speed and system throughput when training a DNN model with multiple replicas in comparison to sequential training with large batch sizes.

We train the ResNet-32 model for 164 training epochs on one and two GPUs using Crossbow. With one GPU, we compare three different ways of training: (i) a baseline sequential training with a batch size of 128; (ii) a sequential training with a larger batch size of 512; and (iii) a training with 4 replicas running in parallel each of which runs on a batch size of 128. With two GPUs, we study two different types of execution: (i) a baseline sequential training with a batch size of 1024, i.e. each device processes 512 training data in parallel; and (ii) a parallel training with 4 model replicas on



(a) 1 GPU (b) 2 GPUs

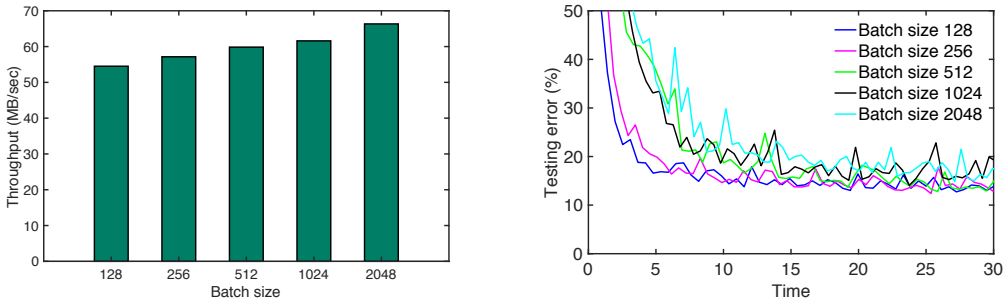
Figure 4.11: Benefit of using multiple replicas over large batch sizes

each device, each model executing batches of 128 training data items. We observe the convergence over time based on the testing accuracy and collect the averaged system throughput.

Fig. 4.11(a) shows that, on a single GPU, the training with 4 replicas yields the best convergence speed. The training completion time is significantly shorter than for both types of sequential training. By training with multiple replicas, we achieve a system throughput of 80 MB/sec; sequential runs with batch size of 512 and 128 give 60 and 54 MB/sec, respectively. Similar results are observed when training the model with two GPUs, as shown in Fig. 4.11(b). In this multi-GPU experiment, we achieve a system throughput at 150 MB/sec when training multiple replicas in parallel; a sequential run with the batch size of 1024 only results in 121 MB/sec. Under a sequential approach, training with a larger batch size reduces the training completion time compared to that with a smaller batch, suggesting that the compute hardware is utilised better. However, we observe in this model that the convergence quality becomes worse when using a larger batch size. This is because the number of model updates per epoch decreases when the batch size is increased.

To put the effect of batch size into perspective, we train the ResNet-32 model on a single GPU and evaluate the convergence speed with five different batch sizes: 128, 256, 512, 1024, and 2048. Fig. 4.12(a) confirms that, by increasing the number of training data items per batch, we can achieve better system throughput. This is because the multi-threaded concurrency of the GPU is used more efficiently when the execution involve large batches of data. However, Fig. 4.12(b) also shows that larger batch sizes result in slower progress in model convergence. Despite the gain in hardware efficiency, the negative impact on statistical efficiency can dominate when the batch size becomes larger.

Training with multiple replicas takes a different approach to maximise GPU compute capability. It performs training with a smaller batch size but executes multiple batches in parallel. As a result, it achieves high hardware efficiency while maintaining a convergence rate close to that of a sequential run with a small batch size. Furthermore, since it can pass over a dataset faster than sequential execution due to its data parallelism, the convergence speed based on wall clock time becomes faster.



(a) Throughput (b) Convergence

Figure 4.12: Training the ResNet-32 model with various batch sizes on a single GPU

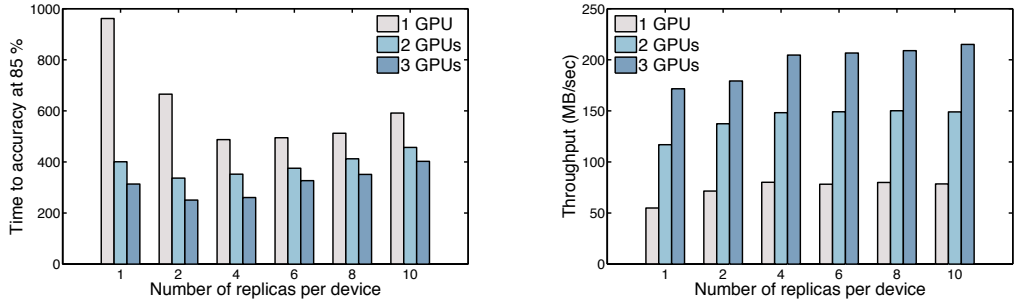
4.5.4 How is the number of model replicas selected?

Crossbow speeds up DNN training by exploiting data parallelism through the means of multiple concurrent replicas. Given a DNN model and the memory capacity of a device, the choice of replica number can be selected from a range of discrete numbers, bounded by the maximum number of replicas fitting on the device. In this experiment, we investigate a heuristic that allows us to pick a reasonable numbers of replicas to obtain fast convergence.

We train three different DNN models, ResNet with 32, 56, and 110 layers, for 164 epochs on one to three GPUs. For each DNN, we vary the number of model replicas per device. With ResNet-32, we vary the number of replicas between 1 and 10 and, for the other ResNet models, we vary between 1 and 6 replicas. Our target testing accuracy is set at 85 %, and the training time required to reach the target is observed. We also collect the average system throughput during the first 100 seconds of each training session.

Fig. 4.13 shows results from the experiment with the ResNet-32 model. According to Fig. 4.13(a), with a single GPU, the shortest time to target is achieved by the training with 4 and 6 replicas. With two and three GPUs, using 2 and 4 replicas, respectively, yields the shortest time. With the corresponding system throughput shown in Fig. 4.13(b), we observe that throughput generally increases when more replicas are used—up to a point at which throughput becomes stable or improves only negligible. For example, training with 1 replica on three GPUs gives a throughput of 170 MB/sec while using 4 replicas results in around 200 MB/sec but we observe little throughput benefit beyond 4 replicas. A similar trend holds with one and two GPUs, e.g. the maximum throughput is reached when using 4 replicas per device. Since execution with 4 replicas also results in the shortest time to reach the target, we potentially can make use of the throughput trend as the heuristic to select the number of replicas.

We train the ResNet-56 and ResNet-110 models to confirm whether this heuristic is applicable to different DNNs. Fig. 4.14 shows that the throughput information can be used to help select the replica number in the ResNet-56 model. In Fig. 4.14(b), the throughput becomes stable when training with 3, 2, and 2 replicas on 1, 2, and 3 devices, respectively. Training with these numbers of replicas on the corresponding numbers of devices results in a short training time to the target accuracy, as shown



(a) Convergence (b) Throughput

Figure 4.13: Training ResNet-32 with various numbers of replicas on a single GPU and multiple GPUs

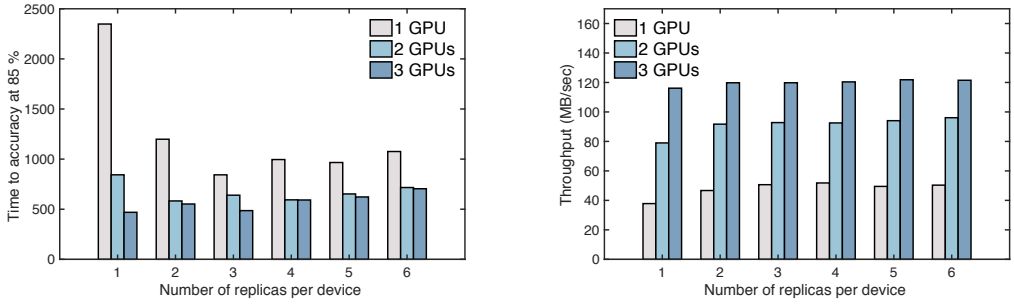
in Fig. 4.14(a). Results from the ResNet-110 model shown in Fig. 4.15 also validate our heuristic. Moreover, increasing the replica number without any throughput gain prolongs the training time. The missing bars indicate that the configuration cannot reach the target accuracy by the time the training ends. This is because executing more replicas on the device when hardware utilisation is already saturated not only help improve hardware efficiency but also introduces a higher chance of model divergence due to the increase in number of parallel replicas. By analysing the relationship between the system throughput and convergence speed for a given number of GPUs, we can find a reasonable replica number per device by incrementing the replica number and selecting the first value that exhibits a stable throughput trend.

4.5.5 What is the effect of the memory reuse technique?

To maximise the parallelism of the underlying GPU multiprocessors, one should ensure that memory does not become the system bottleneck. Crossbow employs a memory reuse technique, previously described in §4.3.2, to improve memory efficiency. Here we evaluate the effectiveness of the memory reuse technique in reducing the total memory usage when training a wide range of DNN models.

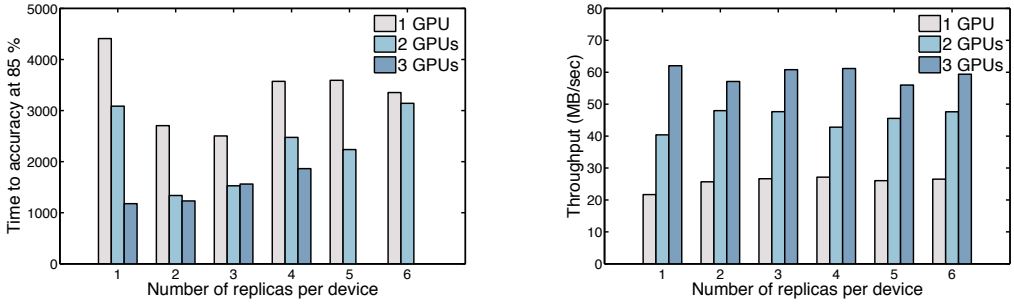
In this experiment, we use two datasets, Cifar and ImageNet, to train ten different models on a single GPU. For Cifar, we train five ResNet models, consisting of 20, 32, 44, 56, and 110 layers, with a batch size of 128. For ImageNet, we train four ResNet models, with 18, 34, 50, and 101 layers and with a batch size of 128, and an Inception-v3 model with a batch size of 32. We observe the amount of memory allocated required to perform the training and report the maximum number of model replicas that can fit on the device when models are trained, with and without our memory reuse technique.

With Cifar, Table 4.4 shows that, by using the memory reuse technique, the maximum number of replicas becomes at least twice as many as that without the technique. In some models such as ResNet-110, the technique allows 6 replicas to be trained concurrently on the same device rather than 2 replicas. Table 4.5 demonstrates that the technique allows the ImageNet DNN models which originally could not fit on a single device to execute. For example, only half of the ResNet-101 model



(a) Convergence (b) Throughput

Figure 4.14: Training ResNet-56 with various numbers of replicas on a single GPU and multiple GPUs



(a) Convergence (b) Throughput

Figure 4.15: Training ResNet-110 with various numbers of replicas on a single GPU and multiple GPUs

can run on the device and thus training must always be split across multiple GPUs. With the reuse technique, it has become possible to run the entire model on a GPU. The technique helps reduce the total amount of memory required to train each replica and therefore makes room for more replicas performing data parallelism.

4.6 Summary

GPUs are capable of fast arithmetic computation and thus have been widely adopted for DNN training. However, most state-of-the-art DNN models still take a significant amount of training time on GPUs. Therefore, higher compute efficiency of GPUs is indispensable because it gives an opportunity to further reduce the model training time.

We present Crossbow, a high-performance DNN system that is designed to maximise GPU compute power by permitting multiple model replicas per GPU when there is room to put more computation. Crossbow exploits multiple GPU streams for the execution of multiple models and thus enjoys the benefit of data parallelism. At the same time, it ensures efficient memory usage to minimise

Model	Number of replicas without memory reuse	Number of replicas with memory reuse
ResNet-20	15	31
ResNet-32	9	20
ResNet-44	7	15
ResNet-56	5	12
ResNet-110	2	6

Table 4.4: Effect of the memory reuse technique on the maximum number of replicas (Cifar)

Model	Number of replicas without memory reuse	Number of replicas with memory reuse
ResNet-18	1	2
ResNet-34	0.7	1
ResNet-50	0.89	1
ResNet-101	0.6	1
Inception-v3	1	2

Table 4.5: Effect of the memory reuse technique on the maximum number of replicas (ImageNet)

the chance of physical memory capacity being a constraint when using multiple replicas on the same device. Furthermore, the system ensures low-latency synchronisation across model replicas and thus allows system to scale across multiple GPUs. Finally, Crossbow demonstrates a 2.3x speedup over an existing DNN system when training a ResNet model on three GPUs.

Chapter 5

Exploring Model Configurations With Meta-dataflows

5.1 Overview

Given a DNN architecture to solve a recognition problem, the choice of training configurations and learning hyperparameters is crucial to the training because it has a direct impact on the convergence quality. Since training a complex model with a large dataset usually takes a considerable amount of training time, it is desirable to select learning hyperparameters that yield fast convergence.

Finding effective training configurations often involves experimentation and thus dataflow systems have been adopted to facilitate the search procedure [dat16], e.g. by carrying out an automatic hyperparameter exploration. However, these approaches are originally designed for general-purpose data processing and provide limited support for an efficient *exploratory workflow*: they often execute the workflow as multiple independent dataflow jobs, which is inefficient because (i) it is difficult to identify executions of poor configurations without completing all the jobs for global comparison; and (ii) the dataflow jobs in the exploratory workflow often have substantial overlaps in their intermediate results. By executing them as independent jobs, such intermediate data is recomputed repeatedly.

We introduce meta-dataflow (MDF), a new dataflow model that effectively expresses exploratory workflows and enables them to execute efficiently on a distributed dataflow system. MDF takes into account all the jobs of an exploratory workflow as a unified dataflow graph. This gives opportunities to apply optimisation techniques to the workflow execution. With MDFs, the system can explore model hyperparameter choices in a more efficient way by reusing intermediate results avoiding redundant computation. At runtime, poor model configurations can be terminated early so that cluster resources are released and used for more promising configurations. This reduces the time required to finish the entire exploratory workflow.

In this chapter, we first discuss the characteristics of exploratory workflows and emphasise the requirements for an efficient execution of such workflows. Then we describe MDF, our new abstraction for exploratory workflows. We give details of the general MDF model, optimisations for MDFs,

and the challenges related to the efficient execution of MDFs, including scheduling and memory management. After that, we describe system architecture and explain how MDF can be supported by existing distributed dataflow systems. The chapter finishes with experiments that demonstrate the effectiveness of MDFs in terms of improvements in the job completion time for a DNN workload and other data analytics applications.

5.2 Exploratory workflows

Since distributed dataflow systems have been adopted to facilitate the exploration of training configurations, we start this section by describing the general model of distributed dataflow systems. We then introduce exploratory workflows, explain how they are realised with existing systems, and emphasise why they are not well supported by those general-purpose dataflow systems. Later, we outline what the requirements are to achieve an efficient execution of exploratory workflows in dataflow systems.

5.2.1 Model for distributed dataflow systems

Dataflow graph

Distributed data processing systems, such as Apache Spark [Apa14b], express computation jobs as *dataflow graphs*, and execute them on a cluster of machines with data parallelism. A dataflow graph is a directed graph, $G = (V, E)$, in which vertices V are data processing *operators*, and edges E are data dependencies between them. A vertex $v \in V$ can have the pre- and post-sets denoted as $\bullet v$ and $v\bullet$ where $\bullet v = \{v' \mid (v', v) \in E\}$ and $v\bullet = \{v' \mid (v, v') \in E\}$. In a dataflow graph, if an operator v is a *source*, it has no pre-sets; $\bullet v = \varnothing$. If an operator v is a *sink*, it has no post-sets; $v\bullet = \varnothing$. Two operators v and v' are connected with a *path* denoted as $p(v, v')$, if there exist edges $e_1, \ldots, e_n \in E$ with $e_i = (v_i, v_{i+1})$, $v_1 = v$, and $v_{n+1} = v'$.

In most existing systems, dataflow graphs are a directed acyclic graph (DAG) without cycles. To support iterative computations, cycles can be either unrolled [Apa14b] or encapsulated by special operators [ABE+14]. In this work, we assume that dataflow graphs are DAGs and that iterative computation is unrolled.

Data model

We model the processed data as *finite datasets* of a domain D and do not impose any assumption on the structure of the data. However, we assume that we can concatenate two datasets d, d' and denote this operation as $d \oplus d' \in D$. Each operator in the dataflow graph applies a function over datasets: function $f_v : D^i \to D^o$ where $i = |\bullet v|$ and $o = |v\bullet|$ are the in- and out-degrees of the operator. Datasets can be partitioned across cluster nodes N on which the dataflow graph is executed. Each node, $n \in N$, has finite amount of memory, $mem(n) \in N_0$ and unbounded disk storage. Partitions of datasets can be stored either in main memory or on disk.

Execution model

Dataflow systems execute a dataflow graph as a *job* in which there are several instances of operators working on different data partitions in parallel on different cluster nodes. Most existing systems [Apa14b, IBY+07, CFMKP13] rely an execution model that uses a scheduler on a *master* node to break down a job into multiple smaller compute *tasks*. A task is essentially a pair of a data partition and an operator, which is executed by *worker* nodes.

Multiple operators can be grouped into a *stage*. Typically operators in the same stage can be executed at a worker in a pipeline. For a dataflow graph $G = (V, E)$, a stage is a set of operators, $T = \{v_1, \ldots, v_n\} \subseteq V$, that have only narrow dependencies [Apa14b]. Between operators $v \in V$ and $v' \in v\bullet$, there is a narrow dependency denoted by $v \twoheadrightarrow v'$, if each partition produced by f_v is used in at most one partition over which $f_{v'}$ is evaluated. For example, the *map* and *filter* functions have a narrow dependency. A dataflow graph can contain multiple stages and the execution order relies on the topological sort of the vertices in the graph. Based on this, we can also escalate the notions of the pre- and post-sets from the vertex level to the stage level. As a result, we use $\bullet T$ and $T\bullet$ to denote the sets of stages that must be executed before and after stage T, respectively.

When a worker node is about to execute a task, the respective data partition must be present at the worker and loaded into main memory. If there is insufficient memory available for the partition, the system makes an *eviction decision* regarding which dataset to store on disk. Modern systems such as Spark adopt the *least-recently used* (LRU) policy [ADU71], which evicts the dataset that has not been used for the longest time.

According to the above notions, we describe the execution of an MDF in terms of *states*. A state (D, δ, μ) is characterised by a set of datasets, and two functions: $\delta : N \times D \to \mathbb{N}_0$ assigns the size of a partition to a node and dataset; and μ: $N \to D$ assigns partitions that are kept in memory to nodes. A state needs to be *valid*, i.e. the total size of partitions kept in memory at a node must not exceed its memory limit: for all $n \in N$, it holds that $\sum_{d \in \mu(n)} \delta(n, d) \leq mem(n)$.

5.2.2 Support for exploratory workflows

To construct a data processing pipeline, users must choose appropriate algorithms and learning hyper-parameters for each individual step, e.g. methods for data pre-processing and model configurations when training a DNN. Although these choices may be simple in certain cases, e.g. the user is familiar with the problem or can apply prior experience to a new similar types of problem, many cases involve an *exploratory* process to make a decision. Here we demonstrate this process with the following example.

Example of dataflow for DNN training

We consider an example in which we want to build an image classifier by training a ConvNet model with an image dataset. Given the model architecture of ConvNet, we want to configure the learning

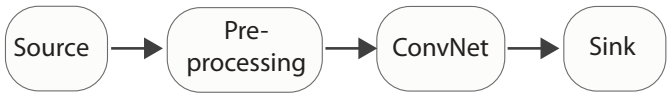


Figure 5.1: DNN training job described as a dafaflow graph

hyperparameters so that we obtain fast convergence. In this example, the learning hyperparameters of interest include the weight initialisation strategy and learning rate. Common data processing pipelines for DNN training usually start with input data pre-processing. The next step is to train the model with the selected learning configurations until convergence. Finally, the classification performance of the model is measured by using the testing data that is not used during the training process.

As shown in Fig. 5.1, a user can execute the above data processing pipeline by expressing it as a dataflow graph. A source operator reads the input data before the second operator applies data standardisation to pre-process the image data. After that, we have an operator that trains ConvNet, whose model parameters are initialised with the Gaussian distribution, and the learning rate is set to 0.01.

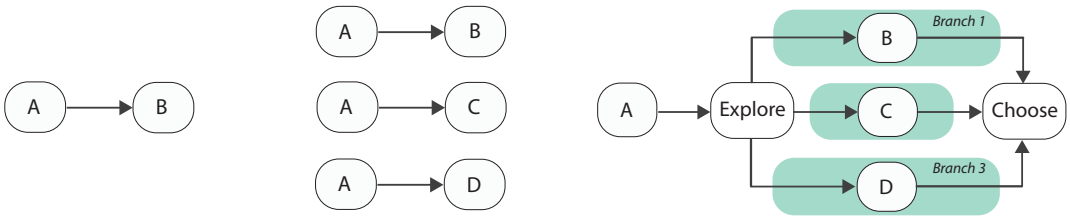
This dataflow graph involves a set of *explorables*, i.e. the choice of learning hyperparameters. A user may want to compare the outcomes obtained from different configurations of the weight initialisation method and learning rate value. In practice, the user would select the model configuration that yields the best result quality—in our case, the best convergence speed.

The process of exploring the choices of explorables in a dataflow graph is referred to as an *exploratory workflow*. In such workflow, a user modifies the dataflow graph by changing the vertices, i.e. using different functions in the operators. Based on the previous discussion in §5.2.1, an exploratory workflow is realised by the execution of multiple related dataflow graphs that are submitted as independent jobs to a dataflow system. This, however, leads to several disadvantages: (i) the user has to ensure the coordination of the entire workflow; (ii) the outcome of each job can be assessed only after the job is completed; and (iii) the best outcome can only be selected when all the jobs have finished. Moreover, an exploratory workflow can easily become cumbersome when there is a large number of explorables in the workflow.

Requirements for an efficient execution of exploratory workflow

When a workflow involves several explorables, the number of possible dataflow jobs can quickly become large. Here we identify four requirements to improve system efficiency and reduce the overall job completion time.

Avoidance of unnecessary computation. The system should not spend time on processing jobs with inefficient or unwanted choices of explorables. Unnecessary computation may arise in an exploratory workflow for two reasons: (i) the quality of the intermediate results may reveal an inferior choice for



(a) A traditional dataflow (b) Set of traditional dataflows (c) Meda-dataflow with three branches

Figure 5.2: Exploratory workflows and meta-dataflow

an explorable; and (ii) the results obtained from other explorables suggest that certain choices of an explorable may not be applicable.

Reuse of intermediate results. Different jobs should reuse intermediate results when possible. When exploring different choices of explorables, many intermediate results can be reused. For example, a task that is common to multiple jobs should be executed only once.

Early discarding of datasets. The intermediate datasets should be discarded as soon as they are no longer needed. This is because a dataflow system requires resource to maintain datasets for an efficient execution. To avoid the memory pressure during the workflow execution, datasets that are no longer useful should be discarded and give room for other datasets that are necessary for the subsequent execution.

Workflow-aware memory management. The management of memory in the cluster should take into account the data access patterns in exploratory workflows. Since exploratory workflows access the datasets in a predictable fashion, a dataflow system can decide which datasets to keep in main memory or evict to disk to reduce the access cost to the intermediate datasets.

These requirements suggest the need for a better integration of multiple related dataflow jobs as well as scheduling and memory management techniques for an efficient execution of exploratory workflows. In the next section, we describe a new dataflow model to address these requirements.

5.3 Meta-dataflows

In this section, we describe a new abstraction for exploratory workflows called *meta-dataflows* (MDFs). First we introduce the general MDF model and give details on its operators. We then discuss how different types of exploratory workflows can benefit from MDFs. Towards the end, we discuss the challenges related to the efficient execution of MDFs.

5.3.1 Meta-dataflow model

The main idea of a meta-dataflow is to integrate a set of related dataflow graphs that have different configurations for the explorables into a single graph. By doing this, the execution of an exploratory workflow can be achieved by submitting a single dataflow job. The meta-dataflow model extends a common dataflow model by adding support for dataflow actions at the *meta*-level. In MDFs, the choices of explorables are represented through `explore` operators and they are selected by `choose` operators, as shown in Fig. 5.2. We refer to the paths between an `explore` and a `choose` operator as *branches*, which represent different values of an explorable. While `explore` represents the beginning of a branch, `choose` controls which intermediate results from branches are used for further computation.

In MDFs, an explorable requires an `explore`. The operators that succeed the `explore` operator execute choices of configurations or learning hyperparameters. The computation within the operators that precede an `explore` and succeed a `choose` are independent of the explorables. By doing this, the MDF model can address the requirement on the reuse of intermediate results since intermediate results are produced once and reused rather than being generated multiple times in separate jobs.

The quality of intermediate results generated from each branch is assessed by a `choose` operator. This is done based on an evaluation function defined within the operator, which selects a subset of intermediate results for further processing. Given this, it is possible for MDFs to avoid unnecessary computations on results of underperforming branches. Since the result of each branch can often be evaluated independently from that of the other branches, a `choose` operator can execute in an *incremental* manner as soon as a branch finishes its computation. This gives an opportunity to discard those intermediate results that are not promising early. Furthermore, the incremental execution of `choose` operators may be used to indicate that some branches are not needed to be executed at all. For example, some choices of an explorable give poorer results compared to the branches that have been executed previously. In this case, these choices are superfluous, and this type of unnecessary computation can be avoided in the MDF.

Formal definition

MDF. A meta-dataflow (MDF) is a dataflow graph, $G = (V, E)$, where $V_< \subseteq V$ represents a set of `explore` operators, and $V_> \subseteq V$ represents a set of `choose` operators. Specifically, for each $v \in V_<$, it holds that $|\bullet v| = 1$ and $|v \bullet| > 1$ and, for each $v \in V_>$, it holds that $|\bullet v| > 1$ and $|v \bullet| = 1$. A path $p(v, v')$ between two operators $v, v' \in V$ is referred as *branch* if $v \in V_<$ and $v' \in V_>$.

The execution semantics of an MDF extends the standard one of dataflow graph. An ordinary operator $v \in V \setminus (V_< \cup V_>)$ can be executed if all preceding operators $v' \in \bullet v$ have been executed. When v is executed, the operator function f_v is applied to the input dataset.

Explore semantics. The input dataset for an `explore` operator is used to compute each branch. Therefore, we can define the execution semantics of `explore` as follows. We represent $G = (V, E)$ as an MDF. The semantics of an `explore` operator $v \in V_<$, $\bullet v = \{v'\}$ is defined such that (i) its operator

function $f_v : D \to D^o$ with $o = |v \bullet|$ and $f_v(d) \mapsto d^o$, i.e. `explore` simply forwards the datasets; and (ii) v can be executed if v' has been executed.

Choose semantics. A `choose` operator selects the datasets that are generated by branches. The semantics of `choose` is determined by (i) an *evaluator* function that calculates a quality score over the resulting dataset of each branch and (ii) a *selection* function that keeps the datasets generated by a subset of branches according to their scores and discards the others. Formally, we can define the execution semantics of `choose` as follows: Let $G = (V, E)$ be an MDF. The semantics of a `choose` $v \in V_>$ is defined by its operator function $f_v : D^i \to D$ with $i = |\bullet v|$, which is $f_v(d_1, \ldots, d_i) \mapsto \rho_v((d_1, \phi_v(d_1)), \ldots, (d_i, \phi_v(d_i)))$ where $\phi_v : D \to \mathbb{R}$ is an *evaluator* function that calculates a score over the intermediate result generated by each branch, and $\rho_v : (D \times \mathbb{R})^i \to D$ is a *selection* function that keeps the datasets from certain branches based on their scores.

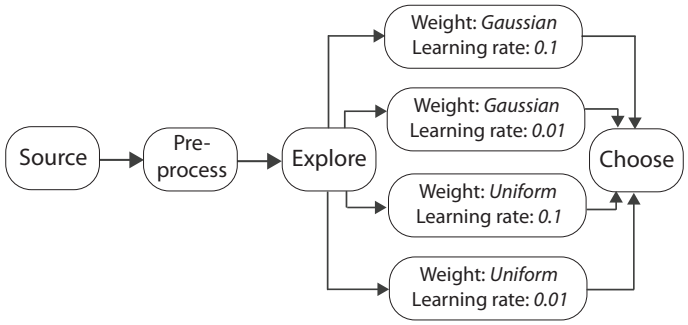
Different types of evaluator and selection functions are supported in MDFs. An evaluator function computes an evaluation score over the values of a resulting dataset. For the selection function, a typical function is *top-k*, which selects the intermediate datasets from k branches with the best quality scores. Other common functions include *min* or *max*, and predicates that check if the score is above or below certain threshold (*threshold*) or whether it falls within a certain interval (*interval*). Furthermore, a selection function can be based on the first-k scores that satisfy thresholds (*k-threshold*) or intervals (*k-interval*).

Example of an MDF for a DNN application

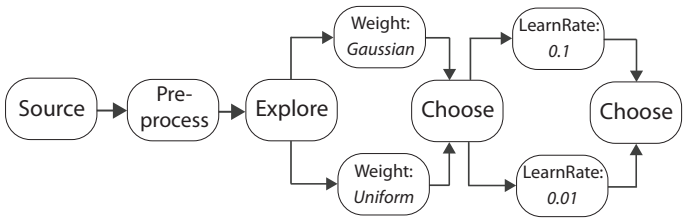
To illustrate how an exploratory workflow can be expressed with an MDF, we present a concrete example using our ConvNet application. Here we want to find the choice of model learning configuration that results in the fastest convergence. Without the exploration, we simply describe the workflow of this job as previously described in Fig. 5.1. Suppose that we are particularly interested in the impact of different weight initialisation methods (e.g. a Gaussian distribution or a uniform distribution) and values of learning rate (e.g. between 0.01 and 0.1) on the convergence speed. In Fig. 5.3(a), we demonstrate an MDF with 4 branches between the `explore` and `choose` operators. The `choose` operator uses an evaluator function that computes the classification accuracy for the quality score. It uses top-1 as the selection function and thus only the datasets generated from the branch that yields the fastest convergence speed is returned as the outcome of the exploration. With MDF, the hyperparameter exploration is simply defined as a single dataflow job.

Optimisations in MDFs

It is possible to apply optimisations during the execution of MDFs based on the combination of different evaluator and selection functions within `choose` operators. An evaluator function may exhibit certain properties, e.g. the function may be monotonic or convex over the choices of the explorable. Similarly, a selection function may be associative or non-exhaustive. In other words, the intermediate



(a) MDF exploring weight initialisation methods and learning rates



(b) MDF that selects datasets at intermediate stage

Figure 5.3: Sample MDFs for a DNN job

results from a subset of branches can be selected independently from those at the other branches. These properties can be adopted to achieve an efficient execution of MDFs. Since a choose operator can perform an incremental evaluation, when a selection function is associative, the resulting datasets from the underperforming branches can be discarded as soon as it becomes clear that they should not be used for further computation. Moreover, with monotonic or convex evaluator functions, we can reason whether the branches that are not yet executed would yield worse results than the already executed ones. If it is the case, an MDF can simply carry on the execution of the next stage without having to complete the execution of all the branches of the explorable. In certain cases, when the evaluator function is neither monotonic nor convex but the selection function is based on the first-k scores, there is no need to execute the remaining branches as long as we have already obtained k results that satisfy the evaluator function.

Common patterns for MDFs

Here we describe some common patterns when exploratory workflows are expressed as MDFs.

Push-down of choose operators. It is beneficial to place choose operators as early as possible in the MDF so that underperforming branches are terminated early. According to this, the user

should therefore consider where to introduce `explore` and `choose` operators in an MDF so the best performance is delivered.

Evaluation of iterative computation. A support for iteration is required for dataflow jobs that perform a fixpoint computation, such as a dataflow graph whose goal is to solve a classification problem. In this case, the user may want to try several learning hyperparameters to solve the given classification task. Each hyperparameter is considered an explorable in the MDF. For each explorable, the MDF must execute the model training with several iterations until convergence. Naively, each branch would execute until completion before the branch quality is assessed by using the evaluator function in the `choose` operator. However, it is possible to incorporate the `choose` operator as part of the iteration itself to avoid the full execution of the branches. By doing this, the `choose` operator can terminate the branches early that do not show signs of convergence.

Fig. 5.3(b) shows another variant of a MDF for our DNN job. In this variant, the MDF avoids unnecessary computation as part of the job execution when a weight initialisation method results in slow convergence speed: the initial `choose` operator selects the model whose weights are initialised in a way that leads to better convergence speed. The `choose` operator evaluates the quality of the resulting models which have executed a certain number of iterations. After a promising weight initialisation method is identified, the MDF carries on the computation to the downstream operators where the value of the learning rate is explored and eventually the full model training is carried out.

5.3.2 Challenges in MDF scheduling

To support MDFs, a distributed dataflow system must schedule the execution of regular operator as well as `explore` and `choose` operators. Scheduling an MDF in an efficient manner is more challenging than scheduling a regular dataflow job because it involves various runtime decisions: for example, which dataset partitions should be maintained in main memory at a particular point in time or which branches would produce better intermediate datasets and later be selected by the `choose` operator. Next we describe the notion of an MDF schedule, which defines how the execution of a MDF progresses by choosing the next stage to execute. We then discuss the requirements for obtaining an efficient scheduling algorithm for MDFs.

MDF schedule. Let $G = (V, E)$ be an MDF. A schedule for an MDF is a sequence of stages $\langle T_1, \ldots, T_k \rangle$, such that (i) it is complete, i.e. there is a stage T_i, $1 \leq i \leq k$, for each sink $v \in V$ of the MDF and (ii) it respects data dependencies: for each stage T_i, $1 \leq i \leq k$ where $\bullet T_i \neq \varnothing$, the required input datasets have already been generated, i.e. $\bullet T_j \subseteq \bigcup_{1 \leq j \leq i} T_j$.

When a schedule $\langle T_1, \ldots, T_k \rangle$ is executed, it results in a sequence of valid states $S = \langle s_0, \ldots, s_k \rangle$. State $s_i = (D_i, \delta_i, \mu_i)$ represents the datasets, their partition sizes at nodes in the cluster, and their storage locations after T_i is executed. Here we have an assumption that a schedule is feasible, i.e. that the dataset partitions $d \in D_{i-1}$ that are required as the input for stage T_i are loaded into the memory. For each node in the cluster $n \in N$, it holds that $d \in \mu_{i-1}(n)$.

Scheduling cost for MDFs. A transition from a state $s_{i-1} = (D_{i-1}, \delta_{i-1}, \mu_{i-1})$ to a state $s_i = (D_i, \delta_i, \mu_i)$ has a *cost* which is the amount of time required to load the input datasets into memory plus the execution time of stage T_i. However, the time required to evaluate the operator functions of each stage does not depend on the order of stages. As a result, the cost of executing stage T_i is the sum of the sizes of dataset partitions that need to be loaded across all the nodes: $c(s_{i-1}, s_i) = \sum_{n \in N} \sum_{d \in \mu_i \setminus \mu_{i-1}} \delta(n, d)$. The cost of an entire schedule is effectively the sum of the cost of all the transitions according to the sequence of states in the schedule $S = \langle s_0, \ldots, s_k \rangle$, i.e. $c(S) = \sum_{i=0}^{k-1} c(s_i, s_{i+1})$.

Requirements for efficient scheduling. An MDF *scheduler* tries to create a schedule with minimal cost. In MDFs, the cost depends on the states that are actually visited during the execution of the schedule. Therefore, it can only be assessed retrospectively after the schedule has been executed. For a given state, it is difficult for a scheduler to estimate correctly which dataset partitions should be spilled to disk when there is not enough cluster memory available. It is also hard to know in advance which datasets should be maintained for future computation given that some branches of the MDF may not get executed at all due to the `choose` operators. Due to the lack of the above information, MDF scheduling must be carried out in an online fashion. This calls for *stage scheduling*: upon the completion of a stage, the next stage to be executed must be determined. As discussed previously, the input datasets must be loaded into the memory so that a stage can be executed. But when the cluster runs out of memory, some intermediate datasets must be evicted to disk. In this case, we need a *memory management policy* to determine which dataset partitions to spill to disk. In the next section, we give details of how we deal with the stage scheduling and memory management in the MDF model.

5.3.3 Branch-aware scheduling

We introduce a branch-aware scheduling algorithm for MDFs that schedules `choose` operators early to avoid computing ineffective branches as soon as possible and also increases the reuse of intermediate datasets during the execution.

The scheduling strategy used by existing dataflow systems is based on breadth-first search (BFS). Under BFS, the cluster resources are used to execute the initial stages to completion before subsequent stages are scheduled. This is, however, not efficient for MDFs: when an `explore` operator is scheduled, each branch will be scheduled and executed to completion before the corresponding `choose` operator is reached. This introduces two drawbacks: (i) since branches generate their intermediate datasets, this makes the execution memory-intensive; and (ii) all branches are executed until completion before a `choose` operator is called, which increases the overall job completion time linearly with the number of branches. Since `choose` operators are not scheduled early, we lose the opportunity to avoid unnecessary computation.

We introduce *branch-aware scheduling* (BAS) which permits the `choose` operator to execute right after each branch so that an early evaluation can be done. BAS adopts depth-first traversal between an `explore` operator and its corresponding `choose`. With BAS, the evaluator and selection functions

in `choose` operators are handled in different ways: the evaluator is applied directly to the resulting datasets by worker nodes, while the selection function is executed during the scheduling decision done by the master node. Since the `choose` operators have an opportunity to evaluate the quality of the executed branches as soon as they are completed, it is possible to abandon the computation of the remaining branches if the already executed ones have satisfied the selection function. Meanwhile, if the quality of the result generated by the current branch does not meet the evaluator's criteria, the memory allocated for the intermediate result can be freed immediately.

With the selection function in `choose` operators, the order in which BAS schedules branches determines efficiency and job completion time. Based on the semantics of `choose` operators, the execution of a branch may make it clear that some remaining branches are unnecessary. To benefit from such optimisations, BAS takes into account hints on the scheduling order of the branches and explorables. For example, a hint may express the priority of the different choices of an explorable. Hints may come from the user's domain knowledge or be derived from the properties of the `choose` operators themselves.

Regardless of the branch selection order, executing MDFs with BAS is superior to BFS-based scheduling strategies as used by current distributed dataflow systems. As previously discussed in §5.3.2, the cost of a schedule can be defined in terms of the sizes of dataset partitions that all the workers have to load into memory during MDF execution. Reducing the number of datasets to be stored decreases the cost of a schedule. This can be achieved by BAS, which employs the depth-first traversal between `explore` and `choose` operators rather than BFS.

5.3.4 Anticipatory memory management

When the memory of a cluster is exhausted, datasets must be evicted to disk. Since loading the evicted datasets back to memory incurs a cost in the schedule, a *memory management policy* plays an important role in determining the efficiency of the system as it decides which datasets should be evicted. A policy is considered optimal if it evicts datasets that will not be used for the longest time [RS94]. Existing distributed dataflow systems such as Spark make use of a least-recently used (LRU) policy. They keep track of information about the last time that datasets were accessed and those which are not accessed for the longest are spilled to disk.

Operators in MDFs such as `explore` have a high out-degree: the input dataset is used as input for a number of explored branches. Such high fan-out graph pattern is uncommon in traditional dataflow graphs and, thus, is not taken into account by existing systems as part of their memory management policy. For MDFs, LRU-based policies result in an inefficient eviction because, with a high fan-out degree of `explore` operators, datasets that have not been recently used may still be needed as input for future downstream operators, and should not be evicted. This inefficiency becomes more prominent when the explored branches are deep.

The structure of the dataflow graphs in MDFs provides useful information on which datasets are accessed during the execution. For example, the dataset generated by an upstream operator of

an `explore` is likely to be accessed many times by the downstream branches that originate from the `explore`; datasets generated by the branches may not be accessed again beyond the `choose` operator due to the evaluation and selection functions. In addition to the information of dataset access patterns, an efficient memory management policy depends on the cost of reloading datasets from disk back to memory. Such cost is directly determined by the sizes of the datasets. Therefore, an efficient eviction policy must take into account both data access patterns and the sizes of dataset partitions.

Here we propose *anticipatory memory management* (AMM) for MDFs that orders datasets by a *preference* to be maintained in the memory based on (i) how often a dataset will be accessed; and (ii) the cost of loading it from disk. When the system exhausts memory, the dataset with the lowest preference will be evicted to disk.

We describe how AMM works as follows. When the cluster memory is exhausted, AMM considers the datasets that are currently maintained in memory. It first computes how often each of these datasets will still be used as input of operators in the graph. This is done by finding out the operator that produces the dataset and then consider the successors of that operator that have not yet been executed. By doing this, it is possible to derive the number of potential future accesses. In the next step, AMM assigns a preference value to each dataset that is currently kept in memory in each node. The preference determines how important it is to keep dataset in memory. It can be derived by computing the product of three factors: (i) the number of future accesses; (ii) the size of the dataset partition on each node; and (iii) a disk/memory cost ratio. The ratio is hardware-specific and calculated as $w_d \cdot r_m / w_m \cdot r_d$ where w_d, w_m, r_d, and r_m are the amount of time to write a fixed amount of data to disk and to memory and to read it from disk and from memory, respectively. Finally, the algorithm returns the dataset with the lowest value of preference and, as a result, the partitions of that dataset on all nodes are selected for eviction when memory is exhausted.

5.4 System architecture

In this section, we describe how MDF can be supported in existing distributed dataflow systems, which typically adopt a master/worker architecture. The master node runs a *scheduler* which gives instructions to the workers about which stages of the dataflow graph to execute next. The worker nodes have *memory allocator*s which manage memory according to the eviction policy.

Scheduler

Typically, a scheduler assigns all the cluster resources to tasks of a single stage. To collect the results from the workers back to the master node, functions called *actions* are triggered. Upon the arrival of the results, the master node applies computation on them. To enable `choose` operators, we can implement a new type of action: the result of the evaluator function in the `choose` operator is sent to the master node, which executes the selection function. After that, the master continues to schedule the remaining branches or the next stage of the computation.

In a schedule, it is straightforward to identify branches because they are always preceded by an `explore` operator. Using a scheduling queue, only stages of one branch are submitted for scheduling. In the meantime, other branches are maintained in a pending branch queue, which will be accessed later by the `choose` operator for the selection procedure. Branches that are selected by a `choose` will remain active: the output datasets of these branches are used for the tasks in the next stages. Specifically, (i) the scheduler identifies the execution of a `choose` in the schedule; (ii) the master retrieves the evaluator results from the worker nodes through an action; (iii) the master executes the selection function; (iv) the master decides which branches to remain to continue to the next stage; and (v) the chosen branches are taken from the pending branch queue and the scheduler schedules the next tasks according to these chosen ones.

Memory allocator

Each node in the system has a memory allocator whose purpose is to load datasets into memory and spill them to disk when the memory is exhausted. In Spark, the functionality is done by the *block manager* at workers. To support the AMM algorithm, the memory allocator can be extended as follows: (i) the master node must implement a policy and establish a mechanism that allows workers to exchange information about memory management with the master node; and (ii) each worker enforces the dataset eviction policy specified by the master node.

When the memory is exhausted, workers notify the master node about the datasets currently maintained in memory and the available memory. The master makes use of this information to execute the AMM policy and generate a list of datasets ordered by the computed preference value. The new scheduling decision is sent to the workers together with the preference list. The workers then enforces the eviction policy accordingly.

5.5 Evaluation

We experimentally evaluate the use of MDFs for exploratory workflows. We use common exploratory workflows from three application domains: (i) a deep learning job that trains a convolutional neural network model, (ii) an analysis job that cleans and mines time series data, and (iii) a data profiling job that uses kernel-density estimation to learn a distribution. In addition, we use (iv) a synthetic job to investigate the performance of MDFs compared to existing distributed dataflow systems.

5.5.1 Experimental set-up

We have added MDF support to SEEP, an open-source distributed dataflow system [CFMKP13, FMKP14]. The architecture of SEEP is representative of that of other systems such as Apache Spark [ZCD+12] or Flink [ABE+14], and its scheduler is similar to Dryad's [IBY+07]. Although MDF is system-agnostic, we use SEEP and Apache Spark to run the experiments due to our local expertise and prior experience. We also compare SEEP's performance to that of Spark.

We run all the experiments on a cluster with 1 master node and 8 worker nodes. Each node has quad-core Intel Xeon E3-1220 CPU with 16 GB of RAM and a local disk. All nodes are connected with a 1 Gbps Ethernet network. We use Ubuntu Linux 14.04.3 with the Linux kernel 3.1 and OpenJDK JVM 8.

Exploratory workflows

We express exploratory workflows as MDFs in three application domains:

(1) DNN job. Exploratory workflows for training DNN models can include explorables for hyperparameters, with the goal of finding the best configuration that results in the highest classification accuracy. We create an MDF that contains three stages: pre-processing of training data, DNN model training, and model validation. In the training stage, the MDF explores: (i) 8 weight initialisation strategies based on either Gaussian or uniform distributions (W); (ii) 4 learning rates (R=$\{0.0001, 0.001, 0.005, 0.01\}$); and (iii) 4 momentum values (M=$\{0.25, 0.5, 0.75, 0.9\}$). With all possible values of initialisation strategies and hyperparameters, the number of paths to explore becomes $|W \times R \times M|$ = 128. The MDF trains a convolutional neural network with the same model architecture throughout the experiment. We use the Cifar dataset that contains RGB image data commonly used for benchmarking machine learning jobs [Kri09]. After an epoch of training, we measure the classification accuracy using the validation data.

(2) Time series analysis job. A common task in time series analysis is to determine which data point, or groups of data points, are of interest for further analysis. A typical data processing pipeline has three stages: (i) *masking* data points in the series based on the range of values within a sliding window, (ii) *marking* discrete events that indicate substantial changes in the series, and (iii) *detecting* sequences of discrete events, each indicating a change of a particular magnitude. For this job, we use a real-world dataset of approximately 350,000 sensor measurements from oil wells [HCCI08].

For the data processing pipeline, we create an MDF with two explorables: masking may use (i) different window lengths (W=$\{2, \ldots, 9\}$), and (ii) thresholds for the permitted difference between the largest and smallest data points in the window (T=$\{1.0001, \ldots, 1.5\}$). We consider different granularities of both parameters, and explore all combinations of the two at each granularity, resulting in between 8 and 64 branches. The obtained intermediate result is then evaluated in terms of the aggressiveness of masking: the MDF evaluates the number of resulting data points, and the ratio of masked data points should not exceed a threshold.

(3) Data profiling job. We use kernel-density estimation (KDE) to create an MDF for data profiling where the probability density function of random variables is estimated. It processes a synthetic dataset with 100 million normally distributed random values. For this job, the MDF has multiple explorables: (i) the data pre-processing method between normalisation and standardisation (N), (ii) the kernel function for the distribution estimation (K=$\{$biweight, triweight, $\ldots\}$), and (iii) the kernel bandwidth (B=$\{0.1, 0.2, 0.3\}$). To evaluate the effectiveness of branches, the MDF uses hold-out samples (1% of

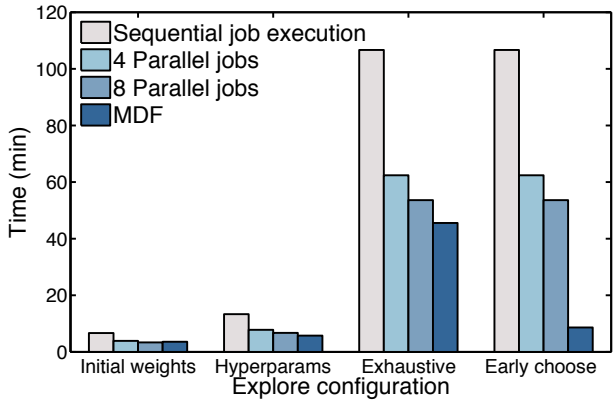


Figure 5.4: Completion times for MDFs in comparison to sequential and parallel execution (DNN job)

the dataset) and computes the log likelihood of the probability density function values of the hold-out samples.

(4) Synthetic job. Finally, we create a synthetic MDF that enables us to have control over the branch structure, and computational workload. The MDF processes string-integer pairs, and uses two nested `explores`, B_1 and B_2. Each `explore` performs an algebraic operation on each branch and updates the integer value in tuples accordingly. The algebraic operation is performed a configurable number of times per data tuple, which allows us to tune the processing workload.

5.5.2 How do MDFs affect completion time?

We study the job completion times of MDFs and compare them against three baseline approaches: (i) sequential executes separate jobs for each explorable setting in sequence. Each job utilises the entire cluster, and once it is finished, the next job is scheduled and executed; (ii) 4-parallel submits four jobs in parallel to the cluster until all the jobs have finished; and (iii) 8-parallel executes eight parallel jobs. The deployment uses 8 worker nodes, and the memory resource on each node is equally shared among workers in the parallel deployments.

Fig. 5.4 shows the completion times for the DNN job with different explorables. The first set of bars shows the time to explore just the *initial weights* W; the second set of bars reflects the exploration of all combinations of *hyperparameters*, $R \times M$. In the first configuration, the differences between all the approaches are small, and completion times are short; in the second one, parallel executions (4-parallel and 8-parallel) begins to show slight speed-ups compared to sequential execution. MDF offers further improvement because it loads and pre-processes the image dataset only once and reuses it across all explored paths.

The final two sets of bars consider both the initial weights and the hyperparameters, thus exploring the full set of combinations. The first option, *exhaustive* exploration, considers all combinations of weights with all combinations of hyperparameters. This leads to the exploration with a number

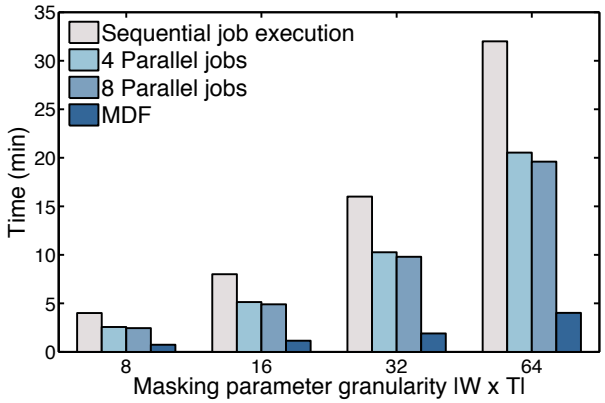


Figure 5.5: Completion times for MDFs in comparison to sequential and parallel execution (Time series analysis job)

of paths equal to the cardinality of the cross product of each set, $|W \times R \times M|$. For the DNN job, however, it is possible to explore the initial weights first, then choose the best result for the subsequent exploration of the hyperparameters within a single job. The last set of bars therefore considers this *early choose* approach, which brings down the number of explored paths to $|W| + |R \times M|$.

Under the *exhaustive* exploration, MDF reduces completion time by 60% compared to sequential. The parallel execution fully utilises the cluster, thus achieving better performance. MDF gives further improvement, reducing completion times by 28% and 15% compared to 4-parallel and 8-parallel, respectively. This is because MDF does not repeatedly pre-process the training data across explored paths.

MDF with the *early choose* exploration reduces completion time by 85% compared to 8-parallel, which represents the best performing baseline approach. Here MDF can avoid training the model with poorly initialised weights W, thus only exploring the promising combinations of the other parameters $R \times M$. Compared to the exhaustive exploration, this requires fewer intermediate datasets to be kept in memory.

Fig. 5.5 shows the results for the time series analysis job. As expected, the completion time of sequential grows linearly with the number of explored combinations $W \times T$. With parallel execution, the completion time increases more slowly due to its better cluster utilisation. All three of the baseline approaches are significantly slower than MDF. With MDF, the choose after the marking operator selects only a subset of intermediate results for further processing (event marking and sequence detection) and, therefore, terminates underperforming branches. This reduces the completion time between 75% and 88% over parallel and sequential execution, respectively.

Fig. 5.6 shows the completion times for the data profiling job. MDF consistently finishes faster than the alternatives—it reduces completion time by 70%, on average, when exploring KDE configurations compared to sequential. This is because the output of the data pre-processing operator is reused for the subsequent different explorations of the kernel functions and bandwidths. The benefit of MDF depends on the input size: the normalisation process is inexpensive, but it requires a linear scan over

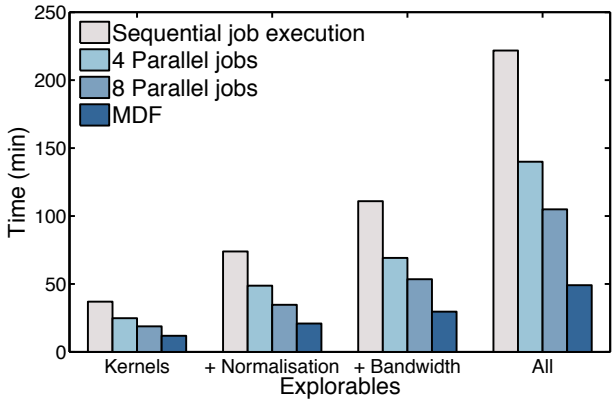


Figure 5.6: Completion times for MDFs in comparison to sequential and parallel execution (Data profiling job)

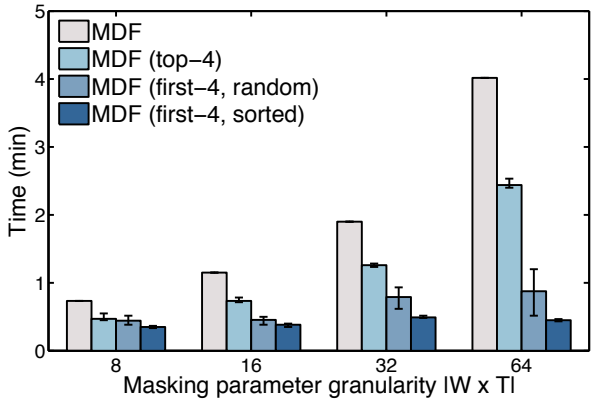


Figure 5.7: Impact of different choose functions

the entire dataset and therefore has increased cost as the dataset size grows. With MDF, this input data is read only once.

Among the baseline approaches, the completion time of parallel execution is shorter than that of sequential, and 8-parallel is faster than 4-parallel. In this job, the computation and I/O operations can be overlapped among the parallel jobs. A higher degree of parallelism, however, increases memory pressure, which limits performance.

5.5.3 What is the impact of different choose functions?

Next we evaluate the effect of the optimisations of the MDF model for different choose functions. We execute the time series analysis job with 4 different functions: (i) choose all data points whose value is above a predefined threshold; (ii) choose the top-4 data points; (iii) choose only the first 4 data points whose value satisfies the evaluation function; and (iv) choose the first 4 while the user has domain knowledge and thus provide hints on the scheduling order. For each choose function, we report the

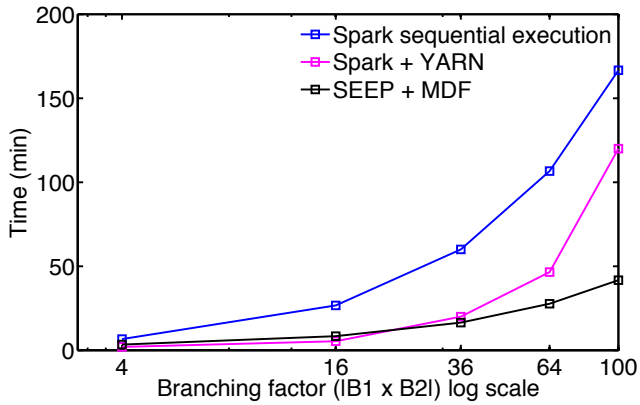


Figure 5.8: Completion times of MDF compared to jobs in Spark

average results over 12 runs and include the minimum and maximum as the error bars. Since the variance between runs is small, the error bars may not be visible.

In Fig. 5.7, the MDF bars are the same as those in Fig. 5.5, exploring all branches to completion. By selecting only the top-4 results at the choose operator, as shown by the MDF (top-4) bar, the system can reduce the completion time by 34%–39%. When the MDF can select any 4 results that meet a threshold rather than the top-4, even larger savings can be achieved: in this experiment, half of the results meet the threshold. We first execute the MDF without hints on the scheduling order, which executes the branches in a random order, as shown by the MDF (first-4, random) bar. Here there is a wide variation in completion times depending on the order. The maximum is always less than that of MDF (top-4), showing an 85% improvement in the best case. When the user provides hints about which branches are likely to satisfy the threshold first, as shown by the MDF (first-4, sorted) bar, the system consistently exhibits shorter job completion time than first-4 without hints.

5.5.4 How does MDF perform compared to other existing systems?

Finally, we put the completion times achieved by MDF into perspective by executing the synthetic job and comparing against another distributed dataflow system, Apache Spark [Apa14b]. We compare MDF against two types of job execution: (i) the equivalent sequential jobs executed by Spark; and (ii) parallel jobs executed by Spark together with YARN [VMD+13], which is a cluster resource manager that can run multiple Spark jobs in parallel. Given the simple nature of computation in the synthetic job, the implementation and workload are the same in SEEP and Spark.

Fig. 5.8 shows the completion time when the branching factors $|B_1|$ and $|B_2|$ vary. In this experiment, we use the same branching factor in the inner and outer explores, i.e. $|B_1|$=$|B_2|$. With the sequential execution under Spark, jobs are queued, and a job can only be started when the previous one has completed. As a result, we observe the that the completion time increases quadratically. With the execution of parallel jobs using Spark with YARN, the cluster utilisation improves as jobs execute as soon as resources become available.

The MDF jobs, however, exhibit significantly shorter completion times compared to Spark/YARN as the branching factor grows—they avoid the redundant execution of stages and can keep data relevant to the current branch in memory. As a result with $|B_1|=|B_2|=10$, which corresponds to 100 explorable configurations, MDF shows a reduction of the completion time by 65% over the execution under Spark/YARN. With lower branching factors, the differences are less significant, especially considering that Spark/YARN does not have the overhead of executing a `choose` operator to select a final output.

5.6 Related work

Scientific workflow management. There are several systems for managing workflows that involve multiple dataflow jobs. Oozie [IHB+12], Azkaban [SKS13] and Luigi [Spo17] orchestrate the execution of jobs, but they do not support data sharing or optimised memory allocation. Pegasus [DVJ+15] is a scientific workflow manager that supports complex workflows on clusters, but lacks support for exploratory workflows: it is concerned with the orchestration of arbitrary dataflow jobs. Ideas from MDF, such as the branch-aware scheduling, could be applied to Pegasus.

Work on *provenance* in scientific workflows [DF08] has focused on how data and results of jobs can be shared among users over time. MDFs efficiently execute exploratory workflows in a distributed dataflow system, whereas existing work on provenance aims at cataloging, sharing and providing access to datasets, which could be a result of an exploratory workflow.

Data sharing in jobs. Prior work has proposed approaches for sharing datasets among different dataflow jobs. Tachyon [LGZ+14] uses a distributed storage layer to cache recently-accessed in-memory data, following an LRU policy. Nectar [GRT+10] is a central service that manages the storage, caching and eviction of datasets. It can share intermediate datasets generated by sub-computations of jobs.

In contrast, MDFs do not only target dataset caching but also schedule computation efficiently for exploratory workflows. MDFs do not require the deployment of a caching layer, but are implemented as part of a dataflow system. Sharing opportunities for intermediate datasets are made explicit in MDFs through the `explore` and `choose` operators, which permits more sophisticated memory management policies that take future dataset usage into account.

Hyperparameter optimisation for ML jobs. Due to the large number of hyperparameters, exploratory jobs are particularly prevalent for machine learning algorithms. A traditional way to perform hyperparameter optimisation is grid search. Since the method relies on an exhaustive search which is computationally expensive, alternative methods are proposed such as random search [BB12]. Given the same computational budget, random search can effectively find better hyperparameter configurations by sampling hyperparameter with a fixed number of times. This is due to the fact that not all hyperparameters have a significant effect on the model performance. Such optimisation methods can be employed together with MDFs for an efficient execution of exploratory jobs: superfluous hyperpa-

rameter choices can be avoided at runtime and hyperparameter exploration can finish immediately after the selection function is satisfied.

5.7 Summary

We presented Meta-dataflow (MDF), a new dataflow model that effectively expresses exploratory workflows and enables them to execute efficiently on distributed dataflow systems. The idea behind MDFs is to capture the expertise of users who want to investigate a range of related dataflow jobs with different algorithms or parameters. MDFs specify such exploratory workflows as a unified dataflow graph and thus give a distributed dataflow system new opportunities for optimising execution. With MDFs, the system can efficiently explore options by reusing intermediate results to avoid redundant computation. At runtime, the ineffective learning configurations can be terminated early according to predefined quality functions, which reduce the total number of possibilities to explore and thus the overall job completion time. We demonstrated the benefits of the MDF model with exploratory workflows in a range of application domains, including a DNN training application.

Chapter 6

Conclusion

Deep neural networks (DNNs) have brought significant advancements in machine learning over the recent years. With a deep structure and flexible model parameterisation, they achieve state-of-the-art accuracy for many challenging tasks such as image recognition [RDS+15a, LBBH98, TYRW14]. While offering powerful recognition and predictive performance, complex DNN models have been known to be difficult to train [HO06, GB10, GBB11, IS15]. This is because their classification performance is directly influenced by the availability of training data and how the learning hyperparameters are configured.

DNN training is not only data-intensive but the process is also compute-intensive, as the computation involves expensive matrix computation, making it time-consuming to obtain converged models. Dataflow systems are used to parallelise computation and accelerate the training process. A common scalable approach is to use a parameter server framework to perform a distributed training [LAP+14]. Since the goal of distributed DNN training is to speed up a long training process, it is necessary that the underlying data processing system performs the training in an efficient way.

While we can accelerate the model training process with a scalable approach such as a parameter server framework, this disseration highlighted three challenges that directly determine the training completion time: (i) how to perform frequent model synchronisation while balancing the compute and network resources in a cluster to achieve the best convergence speed without suffering from system bottlenecks; (ii) how to achieve high compute efficiency in worker nodes to maximise the speed-up gain; and (iii) how to efficiently explore model configurations, given that the hyperparameter space is large, finding the effective configurations that yield fast learning. We described our proposed ideas as part of practical techniques to address each of these challenges.

Model synchronisation. While a parameter server framework has been proposed to address large-scale model training, decisions how to split cluster resource into workers and servers introduce a complexity in deploying distributed training. Since sub-optimal resource splits will result in system bottlenecks or resource underutilisation, a decision on cluster configuration must be made to ensure good scalability.

To fully utilise the CPUs in a cluster for model training and the cluster network bandwidth for model synchronisation, we introduce Ako which is a decentralised DNN system that results in the best convergence speed without the need for finding the optimal cluster resource split to avoid system bottlenecks. Ako scales to large deployments due to a new scalable synchronisation approach called *partial gradient exchange* in which gradient updates propagate to all workers but only using constant bandwidth. Our experiments showed that an implementation of Ako with asynchronous compute and network tasks has better performance on a fixed-sized cluster than one with parameter servers.

This work effectively demonstrates that distributed model training can be achieved without global shared model parameters, which impose the need for decisions on resource allocation in the parameter server framework. By removing the complexity of cluster configuration, Ako offers a practical and scalable way of solving distributed machine learning problems.

Compute efficiency of workers. At a worker node, it is important to ensure high compute efficiency as training complex DNN models tends to be computationally intensive. Since model training always requires a number of passes over the training data, the compute efficiency directly determines the amount of time spent on each iteration and thus training completion time. Many existing machine learning systems [CLL+15, AAB+16] accelerate computation with multiple GPU devices. However, little attention is paid to system efficiency, especially when the compute workload is distributed across multiple devices. Since each device is assigned a fraction of computation in the iteration, the workload per device decreases as more devices are introduced in the system. As a result, the overall system efficiency tends to decrease due to the decline in hardware utilisation on each device.

To maximise the compute utilisation of a worker node, we present Crossbow, a high-performance GPU-based system that is designed to maximise GPU compute power by permitting multiple model replicas per GPU. Crossbow exploits several GPU streams to execute multiple models and thus is able to further leverage GPU concurrency through data parallelism. At the same time, Crossbow ensures efficient memory usage to minimise the chance of physical memory being the system bottleneck. Furthermore, model synchronisation in Crossbow is performed in a low-latency manner to minimise processor idle time and thus enables the training to scale across multiple GPUs. Our experiments showed that Crossbow delivers a 2.3x speed-up over an existing GPU-based system when training a state-of-the-art DNN, ResNet [HZRS15a], on GPUs.

This work demonstrates that the processing power from idle multiprocessors can effectively be translated into better system throughput and eventually convergence speed. By making an effort to maximise all the available hardware resources, we can bring down the training time with the same resource budget. Moreover, an increase in system efficiency further benefits multi-GPU systems as they scale across a larger number of GPUs.

Finding effective choices for model configurations. Solving recognition problems often involves hyperparameter tuning. To find the effective hyperparameter configurations, distributed dataflow systems are adopted to accelerate the process of hyperparameter exploration. However, most systems execute an exploratory workflow as multiple independent dataflow jobs, which can be inefficient.

For an exploratory workflow to find effective learning hyperparameters, we introduce Meta-dataflow (MDF) which is a new dataflow model that can express complex exploratory workflows and execute them efficiently on distributed dataflow systems. MDFs capture the expertise of users who explore a range of related dataflow jobs with different learning configurations. MDFs specify such exploratory workflows as a single integrated dataflow graph, and this gives a distributed dataflow system new opportunities for optimising their execution. With MDFs, the system can explore options while avoiding redundant and unnecessary computation. As a consequence, the cluster resources are used more efficiently, reducing overall job completion time. Our experiments showed the benefits of the MDF model with exploratory workflows for a range of application domains, including DNN training and common data analytics applications.

This work demonstrates that, by expressing exploratory workflows with MDFs, superfluous or underperforming hyperparameter configurations can be avoided and dynamically pruned at runtime. Therefore, we do not waste compute time on unwanted configuration choices. Without MDFs, the execution of exploratory dataflow graphs is static: the outcome of the exploration can be assessed only after the job is complete. Furthermore, MDF is complementary to hyperparameter optimisation techniques, which aim to find a set of optimal hyperparameters for a learning algorithm under given time and compute budgets.

The success of DNNs has been made possible not only by algorithmic breakthroughs but also the improved capabilities of data processing systems. With scalable systems that efficiently train DNN models, we can obtain converged models in a manageable amount of time. Thus training complex DNN models with large amounts of data becomes practical for solving real-world problems.

6.1 Future work

In this section, we extend our discussion by touching on some open problems for future work.

Ako with peer-to-peer transfers between GPUs. We have shown the effectiveness of partial gradient exchange in a distributed setting: we experimented on a cluster of machines with multi-core CPUs that synchronise model parameters in a peer-to-peer fashion over network links. Since GPUs are capable of fast matrix computation, it is interesting to investigate the convergence and scalability when training DNN models with this technique in a multi-GPU setting.

New generations of hardware offer GPU-to-GPU connectivity among devices [NVI16], allowing direct data transfers without accessing main memory. As a result, routing transfers through system memory can be avoided, significantly relieving pressure on the PCIe uplink to CPU. With such direct inter-GPU communication, exchanging gradients during model training should result in low latency. Therefore, the staleness of model parameters can be maintained at a low level, permitting fast convergence speed.

Layer-aware model synchronisation in distributed DNN training. Since DNNs learn hierarchical data representations, it is interesting to study whether different representational levels require different synchronisation schemes and whether different model parts can be trained under different degrees

of staleness. These questions come from three observations: (i) lower layers extract lower-level features from the raw data, while higher layers reason about high-level conceptual knowledge; (ii) the lower layers often constitute a small fraction of model parameters due to invariant learning, while the higher layers involve dense connectivities; and (iii) different layers have different degrees of generality and specificity with respect to the classification task [YCBL14]. By being aware of the DNN architecture, it may become possible to optimise the communication among data parallel workers. Such an approach could potentially result in higher communication efficiency, and help increase training speed-up without introducing a significant negative impact on model convergence.

Novel algorithmic techniques for parallel execution. There have been a number of algorithmic techniques proposed to speed up the convergence rate of DNN training, e.g. momentum [Pol64, PJ92, Nes09, WKH94] and batch normalisation [IS15]. These techniques lead to strong statistical efficiency and their effectiveness has been proven under traditional sequential training.

At the same time, large datasets play a crucial role in achieving high model performance. However, training with sequential execution inevitably incurs synchronisation bottlenecks, making it difficult to scale the training across many nodes. Due to such limited scalability, training acceleration becomes highly dependent on the processing power of individual hardware device and is bounded by the current hardware technology. As a research direction, we would like to explore novel algorithmic techniques for fast training that take the parallel execution into account. With such techniques, we can benefit from hardware efficiency and scalability with minimal negative impact on the statistical efficiency.

Automatic exploration of DNN architecture. Today, most DNN architectures are hand-designed, and this makes automatic methods of finding optimised DNN architectures for a given task appealing. Prior work [MLM+17] proposes to employ an existing neuroevolution technique namely NEAT [SM02]. Since a DNN architecture is expressed as a DAG, the graph topology can be modified and augmented in a straightforward way. With the adoption of genetic algorithms, the model architecture can evolve: a number of variant architectures can be explored until optimal architecture is found. However, executing a DNN model is computationally demanding and thus training all possible models to completion is infeasible. Open problems are (i) how to quantify robustly the performance of a model when it is not fully trained during the evolutionary process; and (ii) whether the evolution can proceed according to hints within the model architecture directly rather than the model classification performance.

Distributed training with model consensus. The volume of data in many domains continues to grow and data tends to be stored at multiple geolocations e.g. data centers in different continents. To maximise the performance of a DNN model for solving a given task, one would want to train the model with as much data as possible. However, in this case, it can be costly and impractical to move the data in order to perform centralised model training. Within this context, we find it interesting to explore the idea of consensus learning [GH14] in which there are multiple DNN models trained at different locations, and they carry out consensus communication algorithm to consolidate the global learning process.

Bibliography

[AAB+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. In *Arxiv preprint arXiv:1603.04467*, 2016.

[AAGNS12] Mohamed Aly Amr Ahmed, Joseph Gonzalezand, Shravan Narayanamurthy, and Alexander Smola. Scalable inference in latent variable models. In *ACM international conference on Web search and data mining (WSDM)*, 2012.

[ABC+16] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[ABE+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere Platform for Big Data Analytics. In *Conference on Very Large Data Bases (VLDB)*, 2014.

[ACDL14] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A Reliable Effective Terascale Linear Learning System. *Journal of Machine Learning Research*, 15(1):1111–1133, 2014.

[AD11] Alekh Agarwal and John C Duchi. Distributed Delayed Stochastic Optimization. In *Advances in Neural Information Processing Systems (NIPS)*, 2011.

[ADU71] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. Principles of Optimal Page Replacement. *Journal of the ACM (JACM)*, 1971.

[Apa11] Apache. Hadoop. https://hadoop.apache.org/, 2011.

[Apa14a] Apache. Mahout. https://mahout.apache.org/, 2014.

[Apa14b] Apache. Spark. https://spark.apache.org/, 2014.

[Apa16] Apache. MLlib. https://spark.apache.org/mllib/, 2016.

[BB12] James Bergstra and Yoshua Bengio. Random Search for Hyper-parameter Optimization. *Journal of Machine Learning Research (JMLR)*, 2012.

[BDFF10] Alexander Berg, Jia Deng, and Li Fei-Fei. Large Scale Visual Recognition Challenge 2010 (ILSVRC). http://image-net.org/challenges/LSVRC/2010/, 2010.

[Ben12] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Arxiv preprint arXiv:1206.5533*, 2012.

[CBD+90] Yan Le Cun, Bernhard Boser, John Denker, R. E. Howard, Wayne Habbard, Lawrence D. Jackel, and Donnie Henderson. Handwritten Digit Recognition with a Back-propagation Network. In *Advances in Neural Information Processing Systems (NIPS)*, 1990.

[CCH+14] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2014.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[CFMKP13] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management. In *ACM International Conference Management of Data (SIGMOD)*, 2013.

[CHK+13] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the Straggler Problem with Bounded Staleness. In *Workshop on Hot Topics in Operating Systems*. USENIX, 2013.

[CHW+13] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with COTS HPC systems. In *International Conference on Machine Learning (ICML)*, 2013.

[CKL+06] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for Machine Learning on Multicore. In *Advances in Neural Information Processing Systems (NIPS)*, 2006.

[CLL+15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *Arxiv preprint arXiv:1512.01274*, 2015.

[CLN11] Adam Coates, Honglak Lee, and Andrew Y. Ng. An analysis of single-layer networks in unsupervised feature learning. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011.

[CMBJ16] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Józefowicz. Revisiting Distributed Synchronous SGD. In *Arxiv preprint arXiv:1604.00981*, 2016.

[CMGS10] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition. In *ArXiv preprint arXiv:1003.0358*, 2010.

[CPS06] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. *International Conference on Frontiers in Handwriting Recognition (ICFHR)*, 2006.

[CSAK14] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[CWV+14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. In *Arxiv preprint arXiv:1410.0759*, 2014.

[dat16] databricks. Deep Learning with Apache Spark and TensorFlow. https://databricks.com/blog/2016/01/25/deep-learning-with-apache-spark-and-tensorflow.html, 2016.

[DCM+12] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[DCML08] Christopher Dyer, Aaron Cordova, Alex Mont, and Jimmy Lin. Fast, Easy, and Cheap: Construction of Statistical Machine Translation Models with MapReduce. In *Proceedings of the Third Workshop on Statistical Machine Translation (StatMT)*, 2008.

[DF08] Susan B. Davidson and Juliana Freire. Provenance and Scientific Workflows: Challenges and Opportunities. In *ACM International Conference Management of Data (SIGMOD)*, 2008.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *ACM Communication*, 51(1), 2008.

[DKW+14] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth A. Gibson, and Eric P. Xing. High-Performance Distributed ML at Scale through Parameter Server Consistency Models. In *Arxiv preprint arXiv:1410.8043*, 2014.

[DKW+15] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric Xing. High-Performance Distributed ML at Scale through Parameter Server Consistency Models. In *Conference on Artificial Intelligence (AAAI)*, 2015.

[DVJ+15] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus, A Workflow Management System for Science Automation. *Future Generation Computer Systems (FGCS)*, 2015.

[DXS+15] Sergey Dudoladov, Chen Xu, Sebastian Schelter, Asterios Katsifodimos, Stephan Ewen, Kostas Tzoumas, and Volker Markl. Optimistic Recovery for Iterative Dataflows in Action. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

[EGW+10] Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. The Pascal Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision*, 88(2):303–338, 2010.

[FHJ+14] Seide Frank, Fu Hao, Droppo Jasha, Li Gang, and Yu Dong. On parallelizability of stochastic gradient descent for speech DNNs. In *IEEE Acoustics, Speech and Signal Processing (ICASSP)*, 2014.

[FMKP14] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Making State Explicit for Imperative Big Data Processing. In *USENIX Annual Technical Conference (ATC)*, 2014.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep Sparse Rectifier Neural Networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*,

volume 15, pages 315–323. Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011.

[GH14] Leonidas Georgopoulos and Martin Hasler. Distributed Machine Learning in Networks by Consensus. *Neurocomputation*, 124:2–12, 2014.

[Goo11] Google. Quantifying comedy on YouTube: why the number of o's in your LOL matter. https://research.googleblog.com/2012/02/quantifying-comedy-on-youtube-why.html, 2011.

[Goo15] Google. TensorFlow: Assign variables to parameter servers. https://www.tensorflow.org/versions/r0.8/api_docs/python/train.html#replica_device_setter, 2015.

[GRT+10] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[Haw04] Douglas M Hawkins. The problem of overfitting. *Journal of Chemical Information and Computer Sciences*, 44(1):1–12, 2004.

[HCC+13] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Advances in Neural Information Processing Systems (NIPS)*, 2013.

[HCCI08] Matthew Hill, Murray Campbell, Yuan-Chi Chang, and Vijay Iyengar. Event Detection in Sensor Networks for Modern Oil Fields. In *Conference on Distributed Event-Based Systems, (DEBS)*, 2008.

[HDY+12] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition. *Signal Processing Magazine*, 2012.

[Hea08] Jeff Heaton. *Introduction to Neural Networks for Java*. Heaton Research, Inc., 2008.

[HGM10] Keith B. Hall, Scott Gilpin, and Gideon Mann. MapReduce/Bigtable for Distributed Optimization. In *Neural Information Processing Systems Workshop on Leaning on Cores, Clusters, and Clouds*, 2010.

[HNP09] Alon Halevy, Peter Norvig, and Fernando Pereira. The Unreasonable Effectiveness of Data. *IEEE Intelligent Systems*, 24(2), 2009.

[HO06] Geoffrey E. Hinton and Simon Osindero. A fast learning algorithm for deep belief nets. *Neural Computation*, 2006.

[How13] Andrew G. Howard. Some Improvements on Deep Convolutional Neural Network Based Image Classification. In *Arxiv preprint arXiv:1312.5402*, 2013.

[HZMR16] Stefan Hadjis, Ce Zhang, Ioannis Mitliagkas, and Christopher Ré. Omnivore: An Optimizer for Multi-device Deep Learning on CPUs and GPUs. In *Arxiv preprint arXiv:1606.04487*, 2016.

[HZRS15a] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *arXiv preprint arXiv:1512.03385*, 2015.

[HZRS15b] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *arXiv preprint arXiv:1502.01852*, 2015.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2007.

[IHB+12] Mohammad Islam, Angelo K. Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, and Andreas Neumann. Oozie: Towards a Scalable Workflow Management System for Hadoop. In *SWEET@SIGMOD*, 2012.

[IS15] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ArXiv preprint arXiv:1502.03167*, 2015.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *ACM International Conference on Multimedia*, 2014.

[KDK11] Noam Koenigstein, Gideon Dror, and Yehuda Koren. Yahoo! Music Recommendations: Modeling Music Ratings with Temporal Dynamics and Item Taxonomy. In *ACM Conference on Recommender Systems (RecSys)*, 2011.

[KH92] Anders Krogh and John A. Hertz. A Simple Weight Decay Can Improve Generalization. In *Advances in Neural Information Processing Systems (NIPS)*, 1992.

[Kri09] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[KTD+13] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. MLbase: A Distributed Machine-learning System. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[LAP+14] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[LASY14] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[LBBH98] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998.

[LBOM98] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient BackProp. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, 1998.

[LGZ+14] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *ACM Symposium on Cloud Computing (SoCC)*, 2014.

[LK09] Percy Liang and Dan Klein. Online EM for Unsupervised Models. In *Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. Association for Computational Linguistics, 2009.

[LKKU15] Hao Li, Asim Kadav, Erik Kruus, and Cristian Ungureanu. MALT: Distributed Data-parallelism for Existing ML Applications. In *ACM European Conference on Computer Systems (EuroSys)*, 2015.

[LMB+14] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: Common Objects in Context. In *Arxiv preprint arXiv:1405.0312*, 2014.

[LNC+11] Quoc V. Le, Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, and Andrew Y. Ng. On optimization methods for deep learning. In *International Conference on Machine Learning (ICML)*, 2011.

[MHM10] Ryan McDonald, Keith Hall, and Gideon Mann. Distributed training strategies for the structured perceptron. In *Annual Conference of the North American Chapter of the Association for Computational Linguistics*, 2010.

[Mit97] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[MLM+17] Risto Miikkulainen, Jason Zhi Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving Deep Neural Networks. In *Arxiv preprint arXiv:1703.00548*, 2017.

[MM15] Dmytro Mishkin and Jiri Matas. All you need is a good init. In *ArXiv preprint arXiv:1511.06422*, 2015.

[MMS+09] Ryan Mcdonald, Mehryar Mohri, Nathan Silberman, Dan Walker, and Gideon S Mann. Efficient large-scale distributed training of conditional maximum entropy models. In *Advances in Neural Information Processing Systems (NIPS)*, 2009.

[MNSJ15] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I. Jordan. SparkNet: Training Deep Networks in Spark. In *Arxiv preprint arXiv:1511.06051*, 2015.

[MZHR16] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Ré. Asynchrony begets momentum, with an application to deep learning. In *ArXiv preprint arXiv:1605.09774*, 2016.

[NBB00] Angelia Nedic, Dimitri Bertsekas, and Vivek Borkar. Distributed Asynchronous Incremental Subgradient Methods, 2000.

[Nes09] Yurii Nesterov. Primal-dual subgradient methods for convex problems. *Mathematical Programming*, 120(1):221–259, 2009.

[Ng04] Andrew Y. Ng. Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance. In *International Conference on Machine Learning (ICML)*, 2004.

[NRRW11] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems (NIPS)*, 2011.

[NVI15a] NVIDIA. GeForce Titan X. https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications, 2015.

[NVI15b] NVIDIA. GPU-Based Deep Learning Inference: A Performance and Power Analysis. https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf, 2015.

[NVI16] NVIDIA. GP100 Pascal White paper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[NVI17] NVIDIA. System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface, 2017.

[OTW+15] Beng Chin Ooi, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao, Zhaojing Luo, Anthony K.H. Tung, Yuan Wang, Zhongle Xie, Meihui Zhang, and Kaiping Zheng. SINGA: A Distributed Deep Learning Platform. In *ACM International Conference on Multimedia (MM)*. ACM, 2015.

[PDDC09] Nicolas Pinto, David Doukhan, James J DiCarlo, and David D Cox. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PLOS Computational Biology*, 2009.

[PJ92] Boris T. Polyak and Anatoli B. Juditsky. Acceleration of Stochastic Approximation by Averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, 1992.

[PL10] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[Pol64] Boris T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[PVG+11] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research (JMLR)*, 12:2825–2830, 2011.

[RDS+15a] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[RDS+15b] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive Modeling*, 1986.

[RLRC13] Romain Reuillon, Mathieu Leclaire, and Sebastien Rey-Coyrehourcq. OpenMOLE, a workflow engine specifically tailored for the distributed exploration of simulation models. *Future Generation Computer Systems (FGCS)*, 29(8):1981–1990, 2013.

[RMN09] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale Deep Unsupervised Learning Using Graphics Processors. In *International Conference on Machine Learning (ICML)*, 2009.

[RS94] Krithi Ramamritham and John Stankovic. Scheduling Algorithms and OS Support for Real-time Systems. *Proceedings of the IEEE*, 1994.

[SFD+14] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In *INTERSPEECH*, 2014.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[SIV16] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *Arxiv preprint arXiv:1602.07261*, 2016.

[SKS13] Roshan Sumbaly, Jay Kreps, and Sam Shah. The Big Data Ecosystem at LinkedIn. In *ACM International Conference Management of Data (SIGMOD)*, 2013.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[SM02] Kenneth O. Stanley and Risto Miikkulainen. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[SMDH13] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning (ICML)*, 2013.

[SN10] Alexander Smola and Shravan Narayanamurthy. An Architecture for Parallel Topic Models. *Proc. VLDB Endow.*, 2010.

[Spa15] Apache Spark. ML Pipelines. https://spark.apache.org/docs/latest/ml-pipeline.html, 2015.

[Spo17] Spotify. Luigi. https://github.com/spotify/luigi, 2017.

[SSP03] Patrice Y. Simard, David Steinkraus, and John C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *International Conference on Document Analysis and Recognition (ICDAR)*, 2003.

[SZ14] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ArXiv preprint arXiv:1409.1556*, 2014.

[Ten16a] TensorFlow. Code example 1. https://github.com/tensorflow/tensorflow/blob/r0.8/tensorflow/models/image/mnist/convolutional.py, 2016.

[Ten16b] TensorFlow. Code example 2. https://github.com/dsindex/tensorflow/blob/master/mlp_mnist_dist.py, 2016.

[Ten16c] TensorFlow. Code example 3. https://github.com/tensorflow/models/blob/master/autoencoder/autoencoder_models/Autoencoder.py, 2016.

[Ten17] TensorFlow. Deep MNIST for Experts. https://www.tensorflow.org/get_started/mnist/pros, 2017.

[Ter13] Doug Terry. Replicated Data Consistency Explained Through Baseball. *Communications of the ACM*, 2013.

[TYRW14] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, 2014.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[VMD+13] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *ACM Symposium on Cloud Computing (SoCC)*, 2013.

[WCC+16] Wei Wang, Gang Chen, Haibo Chen, Tien Tuan Anh Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, and Sheng Wang. Deep Learning At Scale and At Ease. In *Arxiv preprint arXiv:1603.07846*, 2016.

[WCD+15] Wei Wang, Gang Chen, Anh Tien Tuan Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, and Sheng Wang. SINGA: Putting Deep Learning in the Hands of Multimedia Users. In *ACM International Conference on Multimedia*, 2015.

[WDQ+15] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Managed Communication and Consistency for Fast Data-parallel Iterative Analytics. In *ACM Symposium on Cloud Computing (SoCC)*, 2015.

[WGSM16] Sebastien C. Wong, Adam Gatt, Victor Stamatescu, and Mark D. McDonnell. Understanding data augmentation for classification: when to warp? In *Arxiv preprint arXiv:1609.08764*, 2016.

[WHK08] Jason Wolfe, Aria Haghighi, and Dan Klein. Fully distributed EM for very large datasets. In *International Conference Proceeding Series (ICML)*. ACM, 2008.

[Wik17] Wikipedia. List of Nvidia graphics processing units. https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units, 2017.

[WKH94] Wim Wiegerinck, Andrzej Komoda, and Tom Heskes. Stochastic dynamics of learning with momentum in neural networks. *Journal of Physics A: Mathematical and General*, 27(13):4425, 1994.

[WYS+15] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep Image: Scaling up Image Recognition. In *Arxiv preprint arXiv:1501.02876*, 2015.

[WZGZ14] Minjie Wang, Hucheng Zhou, Minyi Guo, and Zheng Zhang. A Scalable and Topology Configurable Protocol for Distributed Parameter Synchronization. In *Asia-Pacific Workshop on Systems (APSys)*, 2014.

[WZZ+13] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L. Cun, and Rob Fergus. Regularization of Neural Networks using DropConnect. In *International Conference on Machine Learning (ICML)*, pages 1058–1066. JMLR Workshop and Conference Proceedings, 2013.

[XHD+15] Eric P. Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A New Platform for Distributed Machine Learning on Big Data. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 2015.

[XKZ+15] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric P. Xing. Distributed Machine Learning via Sufficient Factor Broadcasting. In *Arxiv preprint arXiv:1511.08486*, 2015.

[YCBL14] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Arxiv preprint arXiv:1411.1792*, 2014.

[YRHC15] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul Chilimbi. Performance Modeling and Scalability Optimization of Distributed Deep Learning Systems. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2015.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[ZCL15] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging SGD. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[ZF13a] Matthew D. Zeiler and Rob Fergus. Stochastic pooling for regularization of deep convolutional neural networks. In *ArXiv preprint arXiv:1301.3557*, 2013.

[ZF13b] Matthew D. Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. In *Arxiv preprint arXiv:1311.2901*, 2013.

[ZHW+15] Hao Zhang, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, and Eric Xing. Poseidon: A System Architecture for Efficient GPU-based Deep Learning on Multiple Machines. In *Arxiv preprint arXiv:1512.06216*, 2015.

[ZJL+14] Yongqiang Zou, Xing Jin, Yi Li, Zhimao Guo, Eryu Wang, and Bin Xiao. Mariana: Tencent Deep Learning Platform and Its Applications. *Conference on Very Large Data Bases (VLDB)*, 2014.

[ZLS09] Martin Zinkevich, John Langford, and Alex Smola. Slow Learners are Fast. In *Advances in Neural Information Processing Systems (NIPS)*, 2009.

[ZZY+13] Shanshan Zhang, Ce Zhang, Zhao You, Rong Zheng, and Bo Xu. Asynchronous stochastic gradient descent for DNN training. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.