Adaptable Asynchrony: Improving Adaptive Stochastic Optimisation in Deep Learning

De Sheng Royson Lee
Wolfson College

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

University of Cambridge
Department of Computer Science and Technology
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: dsrl2@cl.cam.ac.uk

June 6, 2018

Declaration

I De Sheng Royson Lee of Wolfson College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 11740

Signed:

Date:

This dissertation is copyright ©2018 De Sheng Royson Lee.
All trademarks used in this dissertation are hereby acknowledged.

Acknowledgement

I am extremely fortunate and thankful to Dr Richard Mortier and Dr Liang Wang for their constant support and advice on this project. I just can't thank them enough for always being accessible and going the extra mile to listen, guide, and provide invaluable feedback. I am also grateful towards Jianxin Zhao for taking the time off to answer my questions and give suggestions to help me debug and get started on the project.

Lastly, I would like to thank Prof Ross Anderson for his continuous encouragement throughout the course.

Abstract

Distributed training of deep learning models is typically trained using stochastic optimization in an asynchronous or synchronous environment. Increasing asynchrony is known to add noise introduced from stale gradient updates, whereas relying on synchrony may be inefficient due to stragglers. Although there has been a wide range of approaches to mitigate or even negate these weaknesses, little has been done to improve asynchronous adaptive stochastic gradient descent (SGD) optimization. In this report, I survey these approaches and propose a technique to better train these models. In addition, I empirically show that the technique works well with delay-tolerance adaptive SGD optimization algorithms, improving the rate of convergence, stability, and test accuracy. I also demonstrate that my approach performs consistently well in a dynamic environment in which the number of workers changes uniformly at random.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep learning has been the key approach for turning data into information and knowledge for a wide range of applications such as understanding phrases [1], detecting anomaly-based intrusion [2], recognising speech [3] and activity [4], amongst others. The rise of Big Data and the increasing demand for complex models to handle millions to billions of parameters has led to the distribution of these computations over a cluster of cores and machines [5]. Distributed and parallel deep learning techniques are used not only to accommodate large-scale models which are unable to fit into a single machine but also to speed up training while ensuring convergence.

The amalgamation of deep learning and distributed system design creates a new realm of possibilities for a wide range of design considerations, ranging from the type of model and data parallelism to the way that nodes compute and communicate with one another during training. Additionally, there is a range of server architectures and topologies, along with their algorithmic approaches, to accommodate and optimise training of these huge models. In this report, I focus on the parameter server topology [6]–[8] in which servers maintain and update the model and workers compute stochastic gradients.

Given the ubiquitous use of mini-batch stochastic gradient descent (SGD) and its adaptive SGD optimisation variants in a sequential setting, many popular training algorithms and recent work aim to adapt and optimise them in the parameter server setting. These stochastic optimisation algorithms are commonly trained in either a synchronous or an asynchronous environment, choosing either is a trade-off between consistency and speed. The former suffers from the straggler problem, which slows down the training due to the need to wait for the slowest worker in each iteration. The latter, on the other hand, suffers from the staleness problem, which allows potentially stale gradients to be used for computation.

I explore the various ways to deal with the straggler and staleness problem and propose a method, named Adaptable Asynchrony (AA), to manage asynchrony in adaptive SGD optimisation and mitigate the staleness problem for deep neural networks. Specifically, I exponentially increase the number of workers and tune the batch size and learning rate based on the difference in asynchrony during training. I showed empirically that applying AA to asynchronous Adagrad [9] (AsyncAdagrad [10]) and its delay-tolerant variants, such as AdaDelay [11], increases the convergence rate, avoids model divergence, and may improve test accuracy at a slight additional run-time cost and an additional hyper-parameter. Furthermore, I showed the consistency of AA by training the model in a dynamic setting in which workers constantly join and leave uniformly at random during training. My primary contributions are:

- Surveying solutions to the straggler and staleness problem faced by both synchronous and asynchronous approaches respectively when training a model using the parameter server approach.
- Observing the effects of applying delay-tolerance asynchronous adaptive SGD optimisation algorithms, which were initially proposed for high-delayed convex problems, to deep neural networks in a small cluster.
- Empirically demonstrating the relation between hyper-parameters and asynchrony for adaptive SGD optimisation.
- Proposing Adaptable Asynchrony to manage varying levels of asynchrony, and thus optimise training, in adaptive SGD optimisation.

The outline of the report is organised as follows. I discuss some of the optimisation and system design choices, problems, and, mitigations in Section 2. I give a brief summary of the popular frameworks and justify my choice to use a particular distributed data processing framework called Owl+Actor [12], [13] for my experiments in Section 2.7. I discuss the changes that I made to Owl+Actor in Section 3 in order to carry out my experiments and make further discussions in Section 4. Most importantly, I show the effectiveness and consistency of AA in Section 4.3 and conclude in Section 5.

Chapter 2

Background & Related Work

I first provide a concise summary on some of the popular sequential SGD optimisation methods. I then discuss the possible system design choices and how some of these sequential methods can be applied in a distributed setting, with an emphasis on Adagrad and its delay-tolerant variants in asynchronous environments. Although my primary goal is to manage asynchrony in adaptive SGD optimisation for deep neural networks in a centralised parameter server setting, I aim to provide some background summary on other possible design options and choices for training large-scale deep learning models and underline where my contributions lie.

2.1 Sequential SGD Optimisation

2.1.1 Preliminaries

In machine learning, the problem is often to estimate the parameter, θ, which minimises or maximises an objective function, $J(\theta)$.

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J_i(\theta)$$

where i refers to the i-th example in the data. The simplest form of parameter update to minimise $J(\theta)$ is as follows:

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)$$

where α is the learning rate, also known as step size, and $\nabla J(\theta_t)$ is the gradient at time step t. In most cases, a mini-batch, which consists of several training examples, is used.

$$\theta_{t+1} = \theta_t - \alpha \frac{1}{m} \sum_{i=1}^{m} \nabla J_i(\theta_t)$$

where m is the size of the mini-batch. The equation is known as stochastic gradient descent (SGD) when $m = 1$, and batch gradient descent when $m = n$. SGD is often used in non-convex optimisation as it helps to escape local minimums and saddle points and improve generalisation due to the noise it brings [14]. However, practitioners tend to use mini-batches as they help to speed up convergence while maintaining enough noise for each gradient update. Therefore, I assume the use of mini-batches for the term SGD in this report.

I focus more on the common SGD methods that are widely used and found in all the current popular deep learning libraries, leaving out computationally impractical second-order methods such as Newton's methods. I then discuss asynchronous variants of some of these methods in Section 2.4. I also leave out other variations of SGD that have been explored in an asynchronous setting. For example, variance-reduced SGD optimisation methods, also known as noise reduction methods, such as SVRG [15] and SAGA [16], have their own asynchronous variants [17], such as ASAGA [18] and KroMagnon [19] respectively. Interested readers who are keen to find out more about these methods may refer to a comprehensive review by Bottou et al. [20].

2.1.2 SGD Optimisation

Momentum

Momentum [21] is used to build up velocity in any direction and dampens oscillations by adding user-defined parameter, p, of the last time step to the update vector. Typical values of p are around 0.9.

$$\begin{aligned} v_t &= p v_{t-1} + \alpha \nabla J(\theta_t) \\ \theta_{t+1} &= \theta_t - v_t \end{aligned}$$

Nesterov Accelerated Gradient

Nesterov Accelerated Gradient (NAG), also known as Nesterov momentum [22], looks ahead and calculates the gradient based on future positions of the current parameters. In

practice, it is known to work slightly better than Momentum as it has stronger theoretical guarantees for convergence. In fact, for reasonably large α, NAG is more stable and allows the use of a larger p. Both Momentum and NAG are also proven to be equivalent when α is small [23].

$$v_t = pv_{t-1} + \alpha \nabla J(\theta_t - pv_{t-1})$$
$$\theta_{t+1} = \theta_t - v_t$$

2.1.3 Adaptive SGD Optimisation

Adagrad

Adagrad [9] is an adaptive learning rate that works well with sparse data. In each time step, it keeps track of the sum of squared gradients in G_t for each parameter where G_t is a diagonal matrix in time step t.

$$G_t = diag(\sum_{i=1}^{t} g_i g_i^T)$$

where

$$g_i = \nabla J(\theta_i)$$

In other words, $G_{t,jj}$ is the sum of squared gradients for parameter j up to time step t. The sum of squared gradients is then used to normalise the update. Note that the multiplication between G_t and g_t is a matrix-vector multiplication and the ϵ is there to prevent division by zero from happening.

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t} + \epsilon} g_t$$

This equation can also be written as

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\sum_{i=1}^{t} g_i^2} + \epsilon} g_t$$

Therefore, each parameter has its own learning rate, which increases when the gradient is smaller and decreases when the gradient is larger, placing more importance on sparse parameters. However, a common downside for Adagrad is that rapid diminishing of learning rate, which is caused by large initial gradients, may stop the learning too early.

Adadelta

Adadelta [24] extends Adagrad by restricting the window of accumulated gradients to size w, instead of size t in time step t, in an attempt to avoid the decay and sensitivity of learning rates in Adagrad. The authors pointed out that storing w previous gradients is inefficient so they implemented this accumulation as an exponentially decaying average of the squared gradients.

$$m_t = pm_{t-1} + (1-p)g_t^2$$

where p is a decay constant similar to momentum's.

The authors also removed the need for a global learning rate α by noting that the units in the parameter update $\Delta\theta$ and the parameter θ should match. Therefore, they assumed that the curvature is locally smooth and approximated $\Delta\theta$ by computing the RMS over a window of size w of previous parameter updates. Hence, apart from accumulating the gradient m_t, an accumulation of updates is computed after the update computation using the same decay constant p.

$$v_t = pv_{t-1} + (1-p)\Delta\theta_t^2$$

The resulting Adadelta method is as follows:

$$\theta_{t+1} = \theta_t - \frac{\sqrt{v_{t-1}} + \epsilon}{\sqrt{m_t} + \epsilon} g_t$$

RMSprop

RMSprop [25], like Adadelta, aims to reduce Adagrad's aggressive, monotonically decreasing learning rate. In fact, it is similar to Adadelta without the accumulation of updates. The authors recommended $p = 0.9$

$$m_t = pm_{t-1} + (1-p)g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{m_t} + \epsilon} g_t$$

Adam

Adam [26], which is essentially RMSprop with Momentum, is an algorithm that computes adaptive learning rates for each parameter using first-order gradient and little memory

requirement. It stores the running average of first, m_t, and second, v_t, moments of the gradient.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

The authors computed bias-correction terms, which was missing RMSProp, to offset the instability caused by setting m and v to 0.

$$\widehat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\widehat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

The update function is similar to RMSProp but using m_t instead of the raw gradient. The authors' proposed values: $\beta_1 : 0.9, \beta_2 : 0.999, \epsilon : 10^{-8}$.

$$\theta_{t+1} = \theta_t - \alpha \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon}$$

AdaMax

AdaMax, similar to Adam, scales the gradient inversely proportional to the L^∞ norm instead of the L^2 norm. Note that there is no bias correction for v_t.

$$\begin{aligned} v_t &= \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty) |g_t|^\infty \\ &= max(\beta_2 v_{t-1} + \epsilon, |g_t|) \end{aligned}$$

The update function is as follows with proposed values: $\beta_1 : 0.9, \beta_2 : 0.999, \alpha : 0.002$.

$$\theta_{t+1} = \theta_t - \alpha \frac{\widehat{m}_t}{v_t + \epsilon}$$

where $\widehat{m}_t$ is defined in Adam.

Nadam

Since NAG is better than Momentum [23], the authors [27] modified Adam to use NAG instead. The momentum term in Adam, $\widehat{w}_t$ is, therefore, modified to incorporate NAG

by taking the bias correction of the gradient.

$$\theta_{t+1} = \theta_t - \alpha \frac{\beta_1 \widehat{m}_t + \frac{(1-\beta_1)g_t}{1-\beta_1^t}}{\sqrt{\widehat{v}_t} + \epsilon}$$

2.2 Distributed System Design Choices

System design affects the scalability, fault tolerance, and the speed of training large-scale models. Although it is not my primary focus, I give a summary of its related work and highlight the choices that I use in my experiments. I briefly cover various components such as the distribution of workload (Section 2.2.1), the ways to communicate and update the model (Section 2.2.2) and the levels of asynchrony involved to bridge communication and computation (Section 2.2.3). Additional components such as GPU and CPU [28], [29] utilization and meta-learning for distributed optimisation [30], [31] are left out in this report.

2.2.1 Data & Model Parallelism

Often, large models need to be split into partitions as they are too big to fit into a single machine. Data partition is used when the data is too big and each worker trains the model with access to a subset of the entire data. The mini-batch is then split into multiple cores or workers and the sub-gradients are summed. Because of the high network overhead when exchanging these sub-gradients, a higher mini-batch size [32] or a reduction in the size of the exchange is often used [33].

Model partition, on the other hand, is used when the model parameters are too huge to fit into a single machine. Each worker computes the gradient for a subset of model parameters. Therefore, communication is needed and dependent on the barrier method used by these workers during the forward and backward passes. A model can be split among GPUs [34], CPUs, or workers where each model partition can be further parallelised across all available cores in each worker [7], [35]. In the context of deep learning, the model can also be split into layers [36] and wait-free backpropagation, a technique that distributes the gradients per layer during backpropagation, can be performed [5], [37].

Both data and model partition are often used in huge topologies. I do not use any data and model parallelism in my experiments as the model is not big enough to justify splitting and I work with a pure asynchronous environment where gradients from different workers update the model independently.

2.2.2 Topologies & Model Updating Techniques

In order to keep track of the model and its state throughout training, either a centralised or decentralised topology can be used. Based on the topology, various model update techniques can be used. In this report, I focus on the parameter server topology, which is described in one of the centralised topologies.

Decentralised

In a decentralised, also known as distributed, topology, workers have access to their set of the data and model. They communicate and distribute their gradients or model updates among themselves to synchronise the barrier (Section 2.2.3). One example is the Halton Sequence topology [5], [38], where workers only need to communicate with a subset of workers and messages can be combined, reducing network load. In the context of deep learning, Nikko [39] proposed a distributed topology where each worker stores their own local replica of a deep neural network. Each worker then updates any sub-gradients received, calculates their own sub-gradient from a mini-batch, and distributes that sub-gradient out. Ensuring that all copies of the deep neural network are synchronised is dependent on the commutative property of the update. Updates are also heavily compressed, utilizing sparsity and speeding up training without the need to transfer models consistently in a centralised setting. A similar example is by Watcharapichat et al. [40] who proposed that each worker constantly splits and sends a partition of the computed gradient in a synchronous manner, allowing the control of network latency. In another example, Vanli et al. [41] proposed an irreducible and aperiodic topology where each worker consists of a single layer feedforward neural network and data is processed in a sequential manner. Although the authors showed that the computational complexity of their algorithm scales linearly with the number of workers, their topology is not fault-tolerant and any node or link failures will deem it unusable.

Centralised

A centralised topology usually consists of a central server or a pool of central servers that hold and update the model parameters. For instance, the P2P engine in Owl+Actor [42] has a centralised server that holds the parameters and the workers coordinate among themselves to achieve barrier synchrony [42]. Similarly, workers can form groups which communicate among themselves before sending an aggregated gradient or model back to the server [7], [8], [43], [44].

The most popular centralised approach, which I focus on, is the parameter server tech-

nique which the server group stores the model parameters and the worker groups perform the intensive computations, which are the forward and backward propagation. The model parameters are either scheduled to or pulled by the worker groups and the gradients of those parameters are calculated and sent back depending on the barrier control. Unlike model averaging, each worker only computes the gradient and not the entire model. This parameter server technique has been widely adopted and used in many systems, such as Google's DistBelief [7], Microsoft's Project Adam [8], ParameterServer [6] in MXNet [45], Commodity Off-The-Shelf High Performance Computing (COTS HPC) systems [34], GeePS [46], B*ö*sen [47] in Poseidon [48], amongst others. Although the general technique is the same, they differ in design choices and advocate different barrier controls. For instance, Project Adam and DistBelief partitions the model into a small worker groups and uses these groups as model replicas. They both support the use of Hogwild!, a lock-free asynchronous approach which is described in Section 2.4.1. In contrast, the former waits for a subset of workers to prevent slow machines from stalling at points where synchrony is needed and the latter reports better results when Adagrad is used. ParameterServer supports both synchronous and asynchronous barrier control and B*ö*sen supports the Stale Synchronous Parallel [49] barrier control, which is described in Section 2.2.3. GeePS offer explicit GPU memory management support to train models that are bigger than the available GPU memory, which is not available in ParameterServer and B*ö*sen. Lastly, COTS HPC systems aim to train similar problems which DistBelief tackles with as little machines as possible. Using a cluster of GPU systems, which each GPU is used to train a partition of a model, and Infiniband interconnects, COTS HPC systems remove the bottleneck caused by the Ethernet network and reduce the computation and communication cost through optimised code practices.

Model averaging, also known as parameter averaging, is another technique that is often used in a centralised topology. In model averaging, each worker group trains its own model and sends back the model to the central server after n iterations. The parameter will then average the models and broadcast the new model to each worker group. However, averaging does not have theoretical guarantees for convergence and it will require further communication and synchronisation among the worker groups if SGD optimisation techniques are used. For instance, in the case of Adagrad, the summed of past gradients squared has to be stored and distributed or it may lead to incorrect behaviour as shown in Section (3.1.1). Nevertheless, model averaging works well in practice for SGD in map-reduce frameworks, such as Hadoop and Spark [50], and is implemented in SparkNet [51]. Additionally, it is also used for training networks with unbalanced and non-IID decentralized data, a technique that is known as Federated Learning [52]. Lastly, the parameter server engine, not to be confused with the parameter server topology, in Owl+Actor [42] uses a variation where each worker computes and sends back the differ-

ence in parameters between the model received from the server and the updated local model every n iterations instead.

2.2.3 Barrier Control

Barrier control, also known as synchronous parallel design, is essential to synchronise updates among the nodes in the topology, determining and coordinating workers' progress during training. The way in which updates are communicated is thus dependent on the barrier method used. In Section 2.3, I refer running SGD in the Bounded Synchronous Parallel (BSP) [53] as SSGD and the Asynchronous Parallel (ASP) [54] as ASGD/Hogwild!. Apart from these two commonly used barrier control methods, there exist other barrier control methods that lie within a spectrum of these two extremes, achieving partial synchrony. I reintroduce BSP and ASP in this section and discuss the other barrier control methods.

In BSP, also known as Bulk Synchronous Parallel, all workers perform a round of computation and communication in order and wait for one another before proceeding to the next iteration. The computed gradients can then be combined, modified, and/or applied individually and parameter updates can occur in a sequential manner. Therefore, BSP is accurate and sequentially correct. However, the main disadvantage of BSP is the need to wait for all workers to finish, causing the training to be as slow as the slowest worker. Therefore, stragglers, huge network delays or node crashes can result in poor performance and robustness.

ASP, on the other hand, allows workers to compute and communicate without waiting for one another, allowing speed-ups at a cost of accuracy. As a result, updates from each worker are usually applied to the model directly and staleness, which refers to gradients that are computed based on an old model, may occur. Unfortunately, in the presence of stragglers or high delays, staleness increases, introducing huge errors and noise to the model. Therefore, in an event where there is a high number of delayed updates (staleness), convergence is not guaranteed.

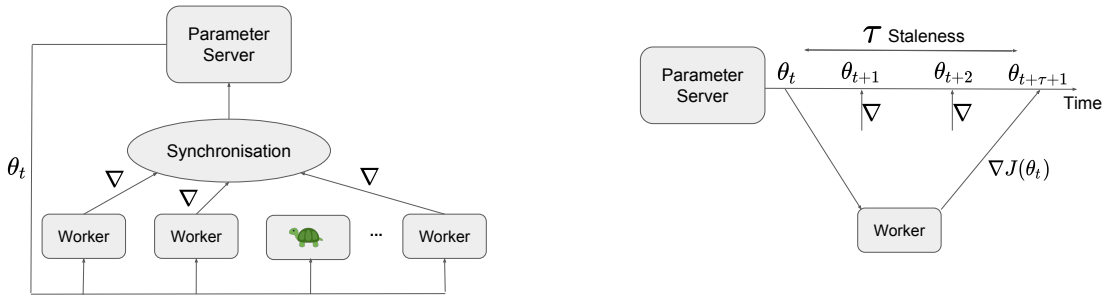
Stale Synchronous Parallel (SSP) [49] allows workers to compute and communicate asynchronously up to a certain threshold, which is known as the staleness threshold. If there are little delays and no stragglers, workers in the SSP setting behave exactly like they would in the ASP setting. However, if the staleness threshold is reached, the faster workers are forced to stop to wait for the slower ones. Doing so allows the speed-ups of ASP while ensuring that the model does not diverge under huge delays [49], [55]. As communication in both BSP and SSP usually occurs at the end of each barrier iteration, network communication usually happens in short bursts. Therefore, in order to utilise

the network, rate-limiting or prioritisation techniques can be used [5].

Probabilistic Synchronous Parallel (PSP) [42] introduces a random sampling primitive to BSP or SSP, forming pBSP or pSSP respectively. Instead of taking the global state into account when making barrier control decisions, PSP makes decisions based on sampled states. Through sampling, PSP is more resistant to stragglers as compared to the other barrier controls, improving robustness. Furthermore, PSP improves scalability as it does not require a view of the global state and it strikes a balance between efficiency and accuracy.

2.3 The Straggler and Staleness Problem

Two main extreme approaches of applying SGD in a parameter server setting are synchronous SGD (SSGD) [56]–[60], which is also known as training on the BSP barrier [53], and asynchronous SGD (ASGD) [7], [8], [61]–[63], which also known as training on the ASP barrier [54].



(a) Straggler (b) Staleness

Figure 2.1: The Straggler and Staleness Problem

In ASGD, workers work independently, requesting new parameters and sending back their gradients, which are applied to the model. Doing so without any form of synchronization results in staleness, also known as a delayed update, delayed information, or simply delay, being applied to the model in exchange for a sub-linear speed-up in training. Specifically, staleness occurs when a worker reads the parameters at time step j to compute gradient j, which is used to update the parameters at time step i where $i > j$. This difference in time steps, $i - j$, is usually referred to as τ [11], [64]. Therefore, increasing asynchrony is known to increase the staleness, bringing additional deviation during training and leading to a poorer convergence. Another way to look at the pros and cons of asynchrony is that increasing asynchrony improves hardware efficiency as it takes a shorter time for each system to perform each iteration, and worsens statistical efficiency as it requires more

iterations to converge [43], [65]. Although the staleness problem is well-known, there has been little theoretical understanding of its behaviour for both convex optimisation [19], [61], [62], [66] and non-convex optimisation [67]. Moreover, to my knowledge, there is no theoretical backup for asynchronous adaptive SGD optimisation algorithms such as Adagrad [9] on non-convex optimisation.

In SSGD, sequential correctness is guaranteed [49], [59]. Each worker waits for one another before their computed gradients are aggregated and applied to the model. For instance, if 10 workers work on a mini-batch size of 10 and send back their gradients for every iteration, it is essentially the same as training the model sequentially with a mini-batch size of 100. This approach trades the staleness problem with the straggler problem, which occurs when every update to the model is as fast as the slowest worker in the pool. Therefore, increasing synchrony improves statistical efficiency but worsens hardware efficiency.

I discuss the approaches to mitigate staleness that are within my system design focus in Section 2.4 and 2.5 and give a summary of ways to mitigate both stragglers and staleness in Section 2.6.

2.4 Distributed Asynchronous SGD Optimisation

Apart from considering the design of distributed systems to support large-scale models, I also consider the approaches to the parameter server setting in an algorithmic approach. I start off by introducing how sequential SGD optimisation techniques are applied naively in a distributed setting and then discuss some of the additional tweaks and methods that have been developed to mitigate staleness. Unlike system design choices, algorithmic approaches tend to focus on the content of the communication and the update itself.

2.4.1 SGD Optimisation

Hogwild!

Hogwild! [61] is a scheme that enables the parallelization of SGD without any locking mechanisms, which is essentially what I refer to as ASGD in this report. The authors proved that the algorithm is mathematically efficient by removing previous assumptions that SGD is inherently sequential. Although the authors focused on only convex optimisation, they inspired similar lock-free approaches on non-convex optimisation. Hogwild!/ASGD has been deployed to solve different optimisation problems, such as deep learning [68] and PageRank approximations [69].

Batching Gradients (n-softsync)

Zhang et al. [64] suggested batching updates before applying the update to the model.

$$c = \lfloor \frac{M}{n} \rfloor$$

where c is the number of updates that are batched, M is the number of workers, and n is user-defined. The update is as follows:

$$\theta_{t+1} = \theta_t - \frac{1}{c} \sum_{l=1}^{c} \alpha_l g_l$$

where α_l depends on the staleness, τ_l, of the incoming gradient. The parameter server records the time step for each worker during the retrieval of the model. It then takes the difference between the recorded time step and the time step in which that update is applied to derive τ_l.

$$\alpha_l = \frac{\alpha_0}{\tau_l}$$

EASGD

Elastic Averaging SGD [70] allows workers to explore before communicating with the server to apply the update. The technique is similar to Model Averaging, which is described in Section 2.2.2, but it factors in the difference between the server's and each worker's parameters. In its asynchronous variant, each worker does its own training and updates its own parameters, x. The server stores its own set of parameters, $\tilde{x}$. Each worker performs n updates to its local model before requesting a value of $\tilde{x}$ from the server. After which, it computes the elastic difference $\alpha(x - \tilde{x})$ which is sent back to the server. The update for each worker and the server is as follows:

$$x \leftarrow x - \alpha(x - \tilde{x})$$
$$\tilde{x} \leftarrow \tilde{x} + \alpha(x - \tilde{x})$$

As the authors proposed the algorithm to be used with SGD, adapting it to include adaptive SGD optimisation might increase the network overhead as the sum of past gradients for each workers would need to be exchanged as well. Therefore, I did not attempt to try EASGD in my experiments.

2.4.2 Adaptive SGD Optimisation

DC-ASGD

Delay Compensated ASGD (DC-ASGD) [71] considers the difference in update time steps as the delay and leverages the Taylor expansion of the delayed gradient during update. I implemented the adaptive version of the algorithm (DC-ASGD-a) because it reportedly achieved the best results. Due to the high complexity of computation of the second order derivative, DC-ASGD uses the approximation of the Hessian matrix instead. The parameter server keeps track of the parameters, θ^{old}, that are pulled by each worker. The update is as follows:

$$z_t = mz_{t-1} + (1-m)g_t^2$$
$$\theta_{t+1} = \theta_t - \alpha(g_t + \frac{\lambda}{\sqrt{z_t} + \epsilon} g_t^2(\theta_t - \theta^{old}))$$

where λ is the variance control parameter. The authors proposed $\alpha = 0.5, \lambda = 2, m = 0.95$ on the CIFAR-10 dataset [72].

AsyncAdagrad

Instead of using a constant learning rate, I can apply Adagrad in an asynchronous setting naively. AsyncAdagrad [10] stores a shared matrix G which, similar to Adagrad, contains a diagonal sum of squared gradients. The authors proposed both a dual-averaging as well as a gradient descent variant of the algorithm and I refer the latter as AsyncAdagrad in my experiments. Since Adagrad uses a separate learning rate for each parameter, it can be easily implemented in situations where the model is split.

AdaptiveRevision

AdaptiveRevision [73] factors the difference in the updated gradients in the learning rate at an additional memory cost. The workers each received not only a subset of the parameters x_i, but also a previous sum of the gradients $\bar{g}_i$. Each worker would then calculate the subgradient for each coordinate i and send both the subgradient, g_i, and $\bar{g}_i$ back to the server if $g_i \neq 0$, essentially using the network to store $\bar{g}_i$.

The parameter server then approximates the delay and calculates a learning rate based on it. Let the values sent back by each worker, $\bar{g}_i$ be $\bar{g}^{old}$ and g_i be g. Let $\bar{g}$ be the current sum of gradients. Initial values are: $\bar{g}_0 = 0, z_0 = 1, z'_0 = 1$. The update at time step t is

as follows:

$$g^{bck} = \bar{g}_t - \bar{g}^{old}$$
$$\alpha^{old} = \frac{\alpha}{\sqrt{z'_{t-1}} + \epsilon}$$

The update is then executed by performing a step and then retracting steps based on the delay; an adaptive revision is applied based on previous delayed steps. Notice that if there is no delay $g^{bck} = 0$, the update is similar to that of AsyncAdagrad.

$$z_t = z_{t-1} + g_t^2 + 2g_t \cdot g^{bck}$$
$$z'_t = max(z_t, z'_{t-1})$$
$$\alpha = \frac{\alpha}{\sqrt{z'_t} + \epsilon}$$
$$\theta_{t+1} = \theta_t - \alpha g_t + (\alpha^{old} - \alpha) g^{bck}$$
$$\bar{g}_{t+1} = \bar{g}_t + g_t$$

The authors only focused on convex optimisation and I further observed its behaviour when applied on a deep neural network in Section 4.2.

AdaDelay

Instead of factoring in delay based on the difference in updated gradients, AdaDelay [11] factors in the difference in update time steps. The authors highlighted that AdaDelay, unlike AsyncAdagrad and AdaptiveRevision, does not assume monotonically decreasing step sizes.

The update at time step t is as follows:

$$z_t = z_{t-1} + \frac{t}{t+\tau} g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\frac{z_t(t+\tau)}{t}} + \epsilon}$$

where τ is the staleness value as described in the n-softsync algorithm. Notice when t is small, high-delay gradients are weighted less and when t grows, high-delay gradients are weighted more. Intuitively, infrequent updates cause smaller steps at the beginning but take bigger steps towards the end to reduce the initial bias. Similar to AdaptiveRevision, AdaDelay is the same as AsyncAdagrad when there is no delay ($\tau = 0$). Similarly, the authors indicated exploring AdaDelay for non-convex optimisation as future work and I

observed its behaviour in Section 4.2.

2.5 Effects of Hyper-parameter Tuning on Asynchrony

Although this section covers SGD optimisation features, I have decided to put it in a separate section to highlight the effects of tuning hyper-parameters on asynchrony for both during and throughout training.

2.5.1 Momentum

Mitliagkas et al. [74] showed that increasing the asynchrony in ASGD directly correlates to increasing the momentum. They used the terms "implicit momentum" for the momentum that was caused by asynchrony and "explicit momentum" for the momentum that was indicated by the user. They then proved that the implicit momentum is as follows:

$$1 - \frac{1}{M}$$

where M is the number of workers. Most importantly, they showed that a negative explicit momentum was needed to offset the high implicit momentum caused by a higher number of workers. However, adding momentum to Adagrad is not the norm and doing so in a distributed fashion did not seem useful as seen in my experiments in Section 4.3.1.

2.5.2 Adaptable Asynchrony

My goal for AA is to negate the negative effects of staleness in asynchronous adaptive SGD optimisation similar to the effects of tuning the momentum for ASGD. However, I did not manage to totally offset the penalty of asynchrony. Instead, I was able to stabilize the training and improve convergence for deep neural networks by slowly increasing the asynchrony. Unlike Chen et al.'s [37] approach to decay the learning rate after detecting a staleness threshold, I decided to tune the learning rate according to the change in the number of workers during training using a similar exponential decay equation used to anneal the learning rate:

$$\alpha = \alpha_0 * e^{-kt}$$

where α_0 is the initial learning rate and k is an additional hyper-parameter. However, instead of factoring in the iteration/epoch number, t, I factored in the change in the number of workers, d, as follows:

$$\alpha = \alpha_0 * e^{-kd}$$

Besides tuning the learning rate, I also tuned the batch size by the same factor as the learning rate as follows:

$$B = \frac{B_0}{e^{-kd}}$$

where B_0 is the initial batch size.

The intuition behind my approach stemmed from a couple of recent papers that advocated large batch training. Although these papers mostly considered the synchronous approach and did not have to factor in the impact of staleness, I was able to better speculate the effects of staleness given a similar scenario. For instance, Smith et al. [75] recommended increasing the batch size instead of decaying the learning rate to obtain the same learning curve at a faster rate. They also observed that increasing the batch size during training led to the gains as opposed to using a high batch size throughout training. Although they showed empirically that decaying the learning rate and increasing the batch size during training were equivalent, the technique was not shown to work on Adagrad. Even if it had worked, increasing the batch size alone in an asynchronous setting would have lead to instability during training, which might possibly cause the model to diverge or definitely cost additional iterations for the model to converge. I hypothesise that increasing the batch size will increase the staleness and the derivation of the update.

Other papers show that linearly increasing the learning rate along with the batch size will result in identical performance up to a certain batch size [76]–[78]. Other variations include linearly increasing the batch size with the learning rate and the training set size [79], increasing the learning rate by the square root of the mini-batch size [80], and increasing the batch size linearly with the number of workers in a synchronous setting [37]. However, increasing both the learning rate and batch size would further result in a greater spike and higher derivation in an asynchronous setting. Besides a linear scaling in both learning rate and batch size for SSGD, Goyal et al. [78] also recommended a gradual warm-up in the learning rate to avoid the initial spike. I speculate that the gradual warm-up technique would complement my work as I observed initial spikes in my experiments and reckoned these spikes might diverge the model if a large influx of workers were to join the training early on.

In contrast, there are recent papers that advocate the use of mini-batches, showing that the use of large batches causes less stable training and worse generalisation performance [81], [82]. However, these papers do not factor in staleness and it is not clear if the noise from an extent of staleness is able to offset the decreasing generalisation caused by increasing the batch size. Hoffer et al. [80] also suggested that performing more update iterations would improve generalisation but I felt that doing so to a full extent would compromise the speed-up of asynchronous training.

Therefore, I decide to explore techniques used to mitigate the impact of label noise. As

the impact of increased label noise and increased staleness is similar, I used the intuition that both label noise and staleness led to additional deviation during training, affecting the rate of convergence and test accuracy. Rolnick et al. [83] suggested increasing the batch size and decreasing the learning rate to train deep neural networks with very noisy training labels. They showed empirically that increasing noise reduced the effective batch size, which could be mitigated by their approach. Although I could not find any theoretical backup to correlate the relationship between staleness and noise in a non-convex setting, I showed empirically that using a similar approach, which I termed AA, helped to improve the rate of convergence, training stability, and might lead the model to converge to a better minimum in Section 4.3.

2.6 Mitigations to the Straggler and Staleness Problem

Both the staleness and the straggler problems have been approached in various ways. For instance, Chen et al. [37], introduced backup workers to alleviate the straggler problem and showed that SSGD outperformed ASGD in convergence and accuracy. Tandon et al. [84] mitigated the straggler problem by getting each worker to compute more than one gradient, allowing servers to realize the need to acquire the required gradients without waiting for the stragglers via small feedback control messages.

The staleness problem, on the other hand, has been approached via algorithmic optimisations that factor in the actual delay [11], [73] or an approximation of the delay [71] during the model update. Apart from taking the delay into account, there have been methods that introduce partial synchronization, such as bounding the staleness [49] or sampling a subset of the gradients [42]. Another way of tuning asynchrony is to introduce worker groups, where workers in each group synchronously communicate with one another and each group asynchronously communicate with the parameter server or other groups [43]. Additionally, there have been optimisations that mitigate the drawbacks that staleness attributed by making changes to the way the gradients are computed [70], updated [64], and communicated [5], [37] in order to reduce network overhead and increase the scalability of the topology. In relation to hyper-parameters, asynchrony in ASGD has been shown to introduce momentum and can be controlled by explicitly offsetting the momentum [74]. However, the impact of adding momentum to Adagrad is not exactly clear and not standard practice, motivating the use of AA to manage asynchrony instead. AA extends the empirical observations made by Chen et al. [37]. In their work on ASGD, Chen et al. reported that even by slowly increasing asynchrony and decaying the learning rate, the model still occasionally diverged. However, they used a static staleness threshold to reduce the learning rate and did not tune the batch size. I showed that by

tuning the batch size and learning rate based on the number of workers, I was able to prevent spikes and ensured a smooth rate of convergence apart from the initial spikes during training. Nevertheless, I hypothesise that the gradual warm-up technique in the learning rate proposed by Goyal et al. [78] complements and mitigates these spikes. A summary of the hardware-agnostic ways to tackle the straggler and staleness problem can be found in Table 2.1. Note that most of the techniques from the different headers and some of the techniques within the same headers are complementary. For instance, deploying gradual warm-up and tuning momentum while training the model with DC-ASGD on the Stale Synchronous Parallel barrier can further mitigate the effects of staleness. Similarity, Adaptable Asynchrony can be used with any partial synchronization method and adaptive SGD optimisation algorithm such as AdaptiveRevision and AdaDelay.

	Straggler	**Staleness**	
		SGD	**Adagrad**
Partial Synchronization			
Stale Synchronous Parallel [49]	✓	✓	✓
Probabilistic Synchronous Parallel [42]	✓	✓	✓
Grouping Workers [43]	✓	✓	✓
Backup Workers [37]	✓		
Distributed SGD Optimisation			
Gradient Coding [84]	✓		
EASGD [70]		✓	
AdaptiveRevision [73]			✓
AdaDelay [11]			✓
Batching Gradients (n-softsync) [64]		✓	?
Delay Compensated ASGD [71]		✓	
Hyperparameter Tuning			
Momentum [74]		✓	
Gradual Warmup [78]		?	?
Adaptable Asynchrony		?	✓

✓- the method has been theoretically or empirically shown to mitigate the problem.
? - the method has not been shown but I hypothesise that it helps

Table 2.1: Ways to Mitigate the Straggler and Staleness Problem

2.7 Frameworks

There are numerous frameworks that are used for not only deep learning but also machine learning and scientific computing in general. These frameworks differ in many aspects such as the way computation graphs are constructed, flexibility, modularity, performance, support for heterogeneous platforms, amongst others. As most of these platforms are

changing frequently, making a comprehensive comparison at every aspect is probably worth a report on its own [85]. In fact, two popular frameworks, PyTorch [86] and Caffe2, which were previously built on Caffe [87], are rapidly changing and merging into one at the time of writing[1]. Therefore, I focus on comparing the distributed and parallel design and features for deep neural networks.

Framework	**Model Update Technique***	**Barrier Control**
SparkNet [51]	MA	BSP
DL4J [88]	MA, PS[1]	BSP, ASP
Tensorflow [89], [90]	PS	BSP, ASP
MXNet [45]	PS	BSP, ASP
SINGA [44]	PS, P2P	BSP, ASP
Keras [91]	MA, PS	ASP
Owl+Actor [12]	MA[2], P2P	BSP, ASP, SSP, PSP

* MA = Model Averaging, PS = Parameter Server, P2P = Peer-to-peer
[1] Additional set-up is provided separately
[2] A variant of MA that is similar to PS (See Section 3.2)

Table 2.2: Default Parallelism Support in Deep Learning Frameworks

Apart from the frameworks listed in Table 2.2, PyTorch supports distributed message passing in both asynchronous and synchronous mode. However, it allows only tensors to be sent across and does not support a PS by default. Caffe2, on the other hand, supports only parallelised synchronous SGD over GPUs. Both frameworks have limited support and, thus, not included in the table.

In my experiments, I created the parameter server framework in Owl+Actor to show my results. A general purpose numerical library developed in OCaml, Owl proves to be better in performance [12] compared to other popular libraries such as Numpy [92] and Scipy [93]. Together with Actor, the frameworks provide both numerical and distribution functionalities. Although Tensorflow allows us to code in just a single system, avoiding the burden of maintaining dependencies and abstraction, modifying the default involves handling C++/python wrappers. Owl+Actor, on the other hand, has a, subjectively, cleaner and more intuitive modular system that is easier to modify or build upon. Furthermore, Owl+Actor, by default, has all the functionality that I need for all possible directions that I previously considered heading towards to at the start of the project. Therefore, Owl+Actor's modular system is highly flexible to cater to my needs.

[1] https://github.com/caffe2/caffe2/issues/2439

Chapter 3

Design and Implementation

In this section, I detail the design considerations and implementation, highlighting parts which need to be refactored to fit into the modular design of Owl+Actor. I first discuss my changes to the parameter server engine, which supports a variant of model averaging, in Owl+Actor to support the parameter server topology that is described in Section 2.2.2. After which, I lay out the groundwork for my experiments in Section 3.2 before discussing implementation changes to accommodate some of the delay-tolerance adaptive SGD optimisation algorithms in Section 3.3 and the changes to include AA in Section 3.4.

3.1 Parameter Server in Owl+Actor

3.1.1 Limitations

The default parameter server engine in Owl+Actor uses a variant of the model averaging technique, which is described in Section 2.2.2 but sends the gradients instead by subtracting the updated local model with the received model. It consists of a single server, which interacts with its workers to train a model. Although it does not support model parallelism, it is enough to attain my goals as my input for my experiments is relatively small. The default parameter server consists of the following interfaces:

- *schedule:* handles scheduling of model parameters to workers depending on the *barrier* control method.
- *pull:* handles the updates of the model parameters and adds them to the centralized model.
- *push:* sends the model updates from workers to the server.

- *stop (Not implemented):* determines when the model finishes training.

The default parameter server works as follows:

1. The client specifies the data, barrier control, and model, along with the training configurations, such as the learning rate, the number of epochs, stopping criteria, amongst others, for each worker.
2. The specified model, along with its data, is individually executed on each worker.
3. Each worker trains the model and applies the updates to their own model. They then take the update in model parameters and send it back via the *push* interface to the server, which then applies it to the centralized model via the *pull* interface.
4. The server then sends the updated centralized model to available workers via the *schedule* interface.
5. Step 3-4 is repeated with the updated model until each worker meets its stopping criteria or finishes training.

The design is similar to the design proposed by Zinkevich et al. [56] for paralleling SGD and can easily be extended to support EASGD [70]. However, there are a couple of design limitations which can cause either incorrect behaviour or a high network overhead when adaptive SGD optimisation like Adagrad is used.

First, as each worker determines when to stop training, there is no indication to the server that the task is completed. Although I initially modified the *push* interface to send a *finish* message when training stopped, a crashed worker process would go unnoticed till the next *heartbeat* message. Additionally, the number of epochs would be reduced every time a worker process crashed, resulting in incorrect intended behaviour. For instance, if a worker process crashed upon initialization, the training would lose $\frac{1}{n}$ batches given n workers.

Each worker also has to initialize its own training state, perform its own weight update, and execute features such as gradient clipping and momentum. Furthermore, each worker sends back the difference in model parameters, an outcome which the server has to average, modify, or simply add back again to the centralized model. Doing so creates more redundancy as compared to the parameter server topology and adopting the latter can improve scalability.

More importantly, training configurations that rely on previously applied gradients such as adaptive learning rates and momentum, are incorrectly executed. Each worker stores and refers to its own cache of past gradients during the weight update, a situation that is semantically incorrect as algorithms such as Adagrad require a global view of the past applied gradients. Moreover, synchronising the cache so that every worker maintains a

global view seems highly inefficient and redundant. Therefore, I decided to implement the parameter server topology into Owl+Actor to support my experiments.

3.1.2 Changes

I constructed the parameter server using the example of its model averaging variant, which is the parameter server engine in Owl+Actor, as a baseline. I then made adjustments to accommodate some of the common features in ParameterServer [6]. First and foremost, the model parameters, state, and caches are only initialized once on the server, saving time and improving scalability.

Each worker is stateless and computes only the intensive part of training: the forward and backward propagation. Therefore, I modified the *push* interface to compute a gradient and the loss given a set of model parameters. I then altered the *pull* interface to perform the weight update given the computed gradient (Listing 1). I also modified Owl's generic optimisation library (*owl_optimise_generic.ml*), breaking the *minimise_network* into two separate functions: *update_network_server* for model updating and *calculate_gradient_worker* for gradient calculation (Listing 2). Each worker sends back not only the gradient but also the loss so that the server does not have to perform the additional forward pass. Since my experiments are done in a small cluster without simulated delay, I figured that it would be a reasonable approach to save computation time. However, in the presence of stragglers, this approach does not work and the server has to make the additional forward pass to get an accurate loss. Apart from the model parameters, the iteration number, t, is also sent to each worker so each worker can work on a different mini-batch.

As the server stores the global state, adaptive SGD optimisation algorithms can be correctly applied. Furthermore, since it also controls the training, the *stop* interface can be trivially implemented (Listing 1). The changes also prevent the loss of training iterations in an event where workers crashed. For instance, if a worker process were to crash during training, the training would proceed with the remaining workers, increasing their workload and the total computation time.

To test the differences, which is shown in Figure 3.1, between the default parameter server engine and the proposed changes, I ran asynchronous mini-batch GD with Adagrad. The experiments were done with the following configuration: a mini-batch size of 100 with replacement, 50 epochs, and $\alpha = 0.001$. I then ran the experiment on the default parameter server engine on 25 workers and called it *AsyncAdagrad-MA*. Lastly, I ran the experiment on my changes with the same number of workers, a situation that is essentially running *AsyncAdagrad* and called it *AsyncAdagrad-PS*.

```
  let local_model task =
    try E.get task.sid |> fst
    with Not_found -> (
      Owl_log.debug "set up first model";
      M.init task.model;
      E.set task.sid task.model;
      E.get task.sid |> fst;
    )

  let exit_condition task_id =
    try E.get (string_of_int task_id ^ "finish") |> fst
    with Not_found -> false

  let push task id vars =
    ...
    let params = task.client_params in
    let x = task.train_x in
    let y = task.train_y in
    let grad, loss = M.calculate_gradient ~params ~init_model:false model
                     x y t in
    ...

  let pull task address vars =
    ...
    let gradient, loss = v in
    let model = local_model task in
    let state = match task.state with
      | Some state -> M.update_network ~state ~params ~init_model:false model
            gradient loss xs
      | None       -> M.update_network ~params ~init_model:false model
             gradient loss xs
      in
    E.set task.sid task.model;
    task.state <- Some state;
    E.set (string_of_int task.sid ^ "finish") Checkpoint.(state.stop);

  let stop task context = exit_condition task.sid
```

Listing 1: Initial Changes to Parameter Server Interfaces

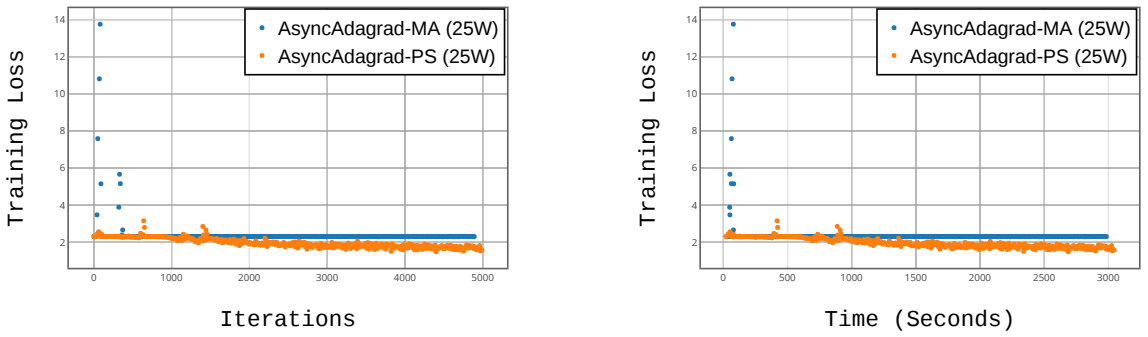
```
let calculate_gradient_worker params forward backward x y t =
  ...
  let xt, yt = bach_fun x y t in
  let yt', ws = forward xt in
  let loss = loss_fun yt yt' in
  (* take the mean of the loss *)
  let loss = Maths.(loss / (F (Mat.row_num yt |> float_of_int))) in
  (* add regularisation term if necessary *)
  let reg = match params.regularisation <> Regularisation.None with
    | true  -> Owl_utils.aarr_fold (fun a w -> Maths.(a + regl_fun w))
                 (F 0.) ws
    | false -> F 0.
  in
  let loss = Maths.(loss + reg) in
  let _, gs' = backward loss in
  let loss = primal' loss in
  (* clip the gradient if necessary *)
  let gs' = Owl_utils.aarr_map clip_fun gs' in
  Gc.minor ();
  (* return gradient and loss *)
  gs', loss

let update_network_server ?state params weights gradient loss update save
                          x_size =
  ...

  (* init new or continue previous state of optimisation process *)
  ...

  (* check if the stopping criterion is met *)
  Checkpoint.(state.stop <- stop_fun (unpack_flt loss));
  (* checkpoint of the optimisation if necessary *)
  chkp_fun save Checkpoint.(state.current_batch) loss state;
  (* print out the current state of optimisation *)
  if params.verbosity = true then Checkpoint.print_state_info state;
  (* calculate gradient descent *)
  let ps' = Checkpoint.(Owl_utils.aarr_map4 (grad_fun (fun a -> a))
            weights state.gs
            state.ps gradient) in
  (* update gcache if necessary *)
  Checkpoint.(state.ch <- Owl_utils.aarr_map2 upch_fun gradient state.ch);
  (* adjust direction based on learning_rate *)
  let us' = Checkpoint.(
      Owl_utils.aarr_map3 (fun p' g' c ->
        Maths.(p' * rate_fun state.current_batch g' c)
      ) ps' gradient state.ch
    )
      in
  (* adjust direction based on momentum *)
  let us' = Owl_utils.aarr_map2 momt_fun Checkpoint.(state.us) us' in
  (* update the weight *)
  let ws' = Owl_utils.aarr_map2 (fun w u -> Maths.(w + u))
            weights us' in
  update ws';
  ...
```

Listing 2: Initial Changes to Owl's Optimisation Library



(a) Iterations to Converge

(b) Time to Converge

Figure 3.1: AsyncAdagrad on Parameter Server

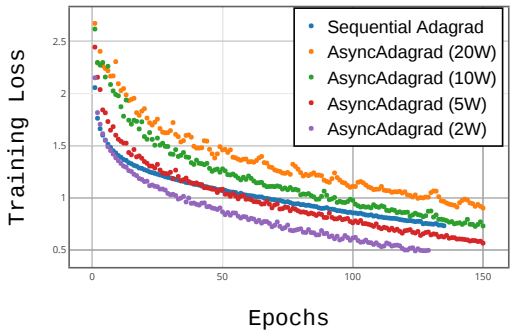
From Figure 3.1, both the default parameter engine (*AsyncAdagrad-MA*) and the new changes (*AsyncAdagrad-PS*) took around the same time. However, as each worker had to spend additional time to initialize and perform the weight updates individually in *AsyncAdagrad-MA*, the additional computation led to a physical machine overload, causing two workers to crash during the last few moments of training, leading to a slightly lower iteration count. Most importantly, *AsyncAdagrad-MA* did not seem to converge as well since it relied on the local caches of each individual worker, which was clearly wrong. In short, I showed that my implementation of the parameter server topology resulted in correct behaviour and could be used as an engine to conduct further experiments with. I further verified my parameter server in Section 3.1.3.

Although changes were made to improve the parameter server, I would like to cover the limitations of the proposed changes. Unsurprisingly, my parameter server cannot accommodate a model that is too large to be fit into a single machine. Changes will be required to support model parallelism, allowing the client to specify the partitions. My parameter server also relies on a single server and will greatly benefit from having additional replications, improving redundancy and fault-tolerance.

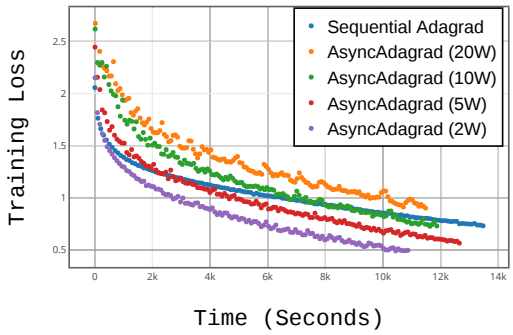
3.1.3 Verification

In order to further verify my parameter server implementation, I illustrated the effects of asynchrony on the rate of convergence and test accuracy. Increasing asynchrony has been known to lead to a slower convergence rate [7], [49], [70], [71]. Additionally, the converged model might not reach the same minimum as that of training in a sequential setting. Therefore, I reproduced the known effects of asynchrony as an indication that my changes in Section 3.1.2 worked as expected.

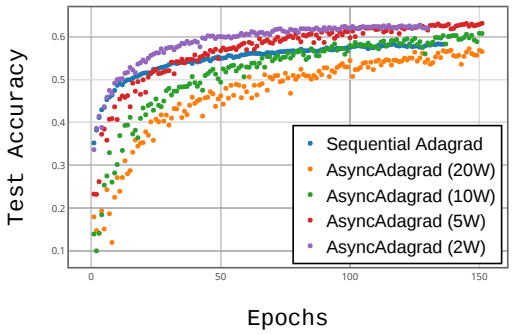
I ran AsyncAdagrad with a varying number of workers $(2, 5, 10, 20)$. I also used a mini-batch size of 128 without replacement, unlike my earlier experiment (Figure 3.1), as it is known to lead to a faster convergence [94]. Additionally, I randomly split the training data into two to include a validation set (80:20) for early stopping to prevent the model from overfitting (See Section 3.2). I used a patience value of 30, $\alpha = 0.001$ and a max epoch size of 150. Moreover, I ran Adagrad sequentially as a baseline for comparison. I plotted, as shown in Figure 3.2, the rate of convergence, the time taken, and the test accuracy for every epoch.



(a) Epochs to Converge



(b) Time to Converge



(c) Test Accuracy

Figure 3.2: Effects of Asynchrony

Through my experiments (Figures 3.2), I could observe that increasing asynchrony led to a speed up during training. However, doing so decreased the rate of convergence and test accuracy due to an increase in staleness. Additionally, I could see that increasing the number of workers caused the model to converge to a different minimum. Notice that the run with 2 workers stopped early as overfitting was detected, resulting in a fewer iteration count. Therefore, my results were aligned to established observations in the field, asserting my implementation of the parameter server. Another interesting observation that I found was that a slight extent of staleness resulted in a better test accuracy. I

hypothesise that the small additional deviation helped to improve the generalisability of the model. In other words, perhaps a little staleness can act as a form of regularization to improve generalisation.

3.2 Groundwork

Before I began running my experiments, I had to make changes to track results for further analysis. All changes were made in the parameter server file or as separate functions in the existing Owl+Actor framework. Therefore, my changes do not affect existing functionality. Although some of my changes are common generic features and are undoubtedly better to be included in the framework, I decided to code them separately for the time being for experimental purposes and integrate them into framework later if results were as expected.

First and foremost, in order to detect overfitting, I used early stopping (Listing 3). I split the training data randomly (8:2) to include a validation set. Unlike the default parameter server engine, I only allocated the data that was needed for both the client and server. Specifically, the workers had access to the training data and the server had access to both the validation and test data. The server would perform a single forward pass to compute the validation loss every epoch and decide if the training should continue or not.

In order to track test accuracy for every epoch, I modified the existing test function for MNIST dataset in Owl. However, parsing in the entire test set for CIFAR-10 would require a lot of memory footprint. Therefore, I split it into batches, calling garbage collection for every batch and running the function on every epoch in the *pull* interface (Listing 4).

I also made some changes that were not used but might be useful for future experiments. For instance, I intended to plot training loss against time using Owl+Actor existing plot functionalities for every iteration in order to see the training visually in real time. However, doing so triggered a bug that led to a segmentation fault, possibly due to a misconfiguration on my local Ubuntu machine [1]. Therefore, I wrote the outputs to files instead and used Python to generate the figures separately.

Besides plotting, I initially altered the framework to track record on the average network time per worker and the total wall time taken during training. In order to determine the average network time, I recorded the time (*response*) taken for every worker to send back a gradient for every model scheduled. I also recorded the time (*computation*) each worker

[1] https://github.com/owlbarn/owl/issues/189

```
let validate_model task i =
  let x = task.val_x in
  let y = task.val_y in
  let model = M.copy task.model in
  let xt, yt = (Batch.run task.server_params.batch) x y i in
  let yt', _ = (M.forward model) xt  in
  let loss = (Loss.run task.server_params.loss) yt yt' in
  (* take the mean of the loss *)
  let loss = Maths.(loss / (F (Mat.row_num yt |> float_of_int))) in
  unpack_flt loss

let pull task address vars =
  (* Calculate Validation loss every epoch *)
  let _ = match Checkpoint.(state.current_batch mod
                            (state.batches_per_epoch) = 0) with
    | false ->  ()
    | true  ->  let vl = validate_model task
                (Checkpoint.(state.current_batch /
                 (state.batches_per_epoch)) - 1) in
                match task.lowest_val_loss <> 0.
                      && vl >= task.lowest_val_loss with
                  | true  ->  task.patience <- task.patience + 1
                  | false ->  task.lowest_val_loss <- vl;
                              task.patience <- 0
  in

  (* Determine if training ends *)
  let _ = match task.patience >= 30 with
  | false -> ()
  | true  -> Owl_log.info "Early stopping..";
             Checkpoint.(state.stop <- true)
  in

let train_generic ?params nn x y tx ty jid url =
  (* prepare params and make task *)
  let params = match params with
    | Some p -> p
    | None   -> Params.default ()
  in
  let sid = Owl_stats.uniform_int_rvs ~a:0 ~b:max_int in
  let cid = Owl_stats.uniform_int_rvs ~a:0 ~b:max_int in

  (* Split training and validation data to 80:20 *)
  let r = Array.init (Owl_dense_ndarray.S.nth_dim x 0) (fun i -> i) in
  let r = Owl_stats.shuffle r in

  (* Validation data *)
  let v_rows = Array.sub r 0 2000 in
  let vx = Arr (Owl_dense_ndarray.S.get_fancy [L (Array.to_list v_rows)] x) in
  let vy = Arr (Owl_dense_ndarray.S.rows y v_rows) in

  (* Training data *)
  let t_rows = Array.sub r 2000 8000 in
  let x = Arr (Owl_dense_ndarray.S.get_fancy [L (Array.to_list t_rows)] x) in
  let y = Arr (Owl_dense_ndarray.S.rows y t_rows) in

  let server_task = make_server_task sid params nn x vx vy tx ty in
  let client_task = make_client_task cid params x y in
  (* register sched/push/pull/stop/barrier *)
  E.register_schedule (schedule server_task);
  E.register_pull (pull server_task);
  E.register_push (push client_task);
  E.register_stop (stop server_task);
  E.start ~barrier:E.ASP jid url
```

Listing 3: Early Stopping in Parameter Server

```
let test_network task =
  let model = M.copy task.model in
  let open Owl_dense in
  let imgs, labels = unpack_arr task.test_x, unpack_arr task.test_y in

  let mat2num x s = Matrix.S.of_array (
      x |> Matrix.Generic.max_rows
        |> Array.map (fun (_,_,num) -> float_of_int num)
    ) 1 s
  in

  let s = [
  [ [0;499] ] ; [ [500;999] ] ; [ [1000;1499] ] ; [ [1500;1999] ]
  ; [ [2000;2499] ] ; [ [2500;2999] ] ; [ [3000;3499] ] ; [ [3500;3999] ]
  ; [ [4000;4499] ] ; [ [4500;4999] ] ; [ [5000;5499] ] ; [ [5500;5999] ]
  ; [ [6000;6499] ] ; [ [6500;6999] ] ; [ [7000;7499] ] ; [ [7500;7999] ]
  ; [ [8000;8499] ] ; [ [8500;8999] ] ; [ [9000;9499] ] ; [ [9500;9999] ]
  ]
  in
  let calc_accu s1 =
    let imgs1 = Ndarray.S.get_slice s1 imgs in
    let m = Ndarray.S.nth_dim imgs1 0 in
    let label1 = Ndarray.S.get_slice s1 labels in
    let fact1 = mat2num label1 m in
    let pred1 = mat2num (M.model model imgs1) m in
    let accu1 = Matrix.S.(elt_equal pred1 fact1 |> sum') in
    Gc.minor ();
    accu1
  in
  let accu = List.map calc_accu s |> List.fold_left (+.) 0. in
  let m = Ndarray.S.nth_dim labels 0 in
  let res = (accu /. (float_of_int (m))) in
  Owl_log.info "Accuracy on test set: %f" res;;
```

Listing 4: Computing Test Accuracy in Parameter Server

took to compute the gradient. The average network time was determined as follows:

$$\frac{\sum_i^p (response_i - computation_i)}{n}$$

where n is the number of workers and p is the number of schedules and gradients computed. However, as my experiments in this report did not involve delay simulation or include additional load on the network, the average network time was similar in all the experiments I conducted and I decided to omit it. However, it might be useful for future experiments that might use the network as storage or simulate delay by introducing stragglers.

3.3 Delay-Tolerant Adaptive SGD Optimisation

Owl comes with its own modular library to accommodate not only sequential SGD optimisations but also other optimisation methods such as conjugate gradient and coordinate descent. The library contains its generic functions to minimise a function or a network in an abstract order such as the *update_network_server* function in Listing 2. However, delay-tolerant adaptive SGD optimisation requires more steps and caches. For instance, AdaptiveRevision requires the learning rate to be calculated once before and after a cache update and an extra revision step factor in the delay. Therefore, changing the library to accommodate these extra steps would result in a huge change in other parts of the program. Therefore, each algorithm used the existing functionality in the sequential optimisation library for its generic steps and had its extra steps coded directly in the *update_network_server* function. In order to make these steps generic in the future, these distributed optimisations should be coded in a separate distributed optimisation library as they required more inputs than local optimisation and should not be used sequentially.

I added AdaptiveRevision, AdaDelay, and Delay-Compensation ASGD (DC-ASGD) into the library (Listings 5 to 8) although only the first two were experimented with. I also kept track of the additional information needed as a key-value pair for each of these algorithms in the parameter server. For instance, I kept track of the gradient, iteration number, and model sent to each worker in the *schedule* interface for AdaptiveRevision, AdaDelay, and DC-ASGD respectively. I then calculated the delay depending on the algorithm and passed it into the *update_network_server* function before updating the required caches (Listings 9 and 10). As a result, AdaptiveRevision would require more memory than AdaDelay. Although McMahan and Streeter [73] used the network to store the additional information, I decided to store the information in the parameter server as Sra et al. [11] did, trading-off additional network overhead with additional server memory consumption. More details on these algorithms can be found in Section 2.4.2.

```
module Learning_Rate = struct

  type typ =
    ...
    | AdaptiveRev  of float (* Distributed training only *)
    | AdaDelay     of float (* Distributed training only *)
    | DelayComp    of float * float * float (* Distributed training only *)

  let run = function
    ...
    | AdaptiveRev a       -> fun _ _ c -> Maths.(F a /
                             sqrt (c.(1) + F 1e-32))
    | AdaDelay a          -> fun i _ c ->
                               Maths.(sqrt (c.(0) / (F (float_of_int i))))
    | DelayComp (_, v, _) -> fun _ _ c -> Maths.(F v /
                             sqrt (c.(0) + F 1e-7))

  let update_ch typ g c = match typ with
    | DelayComp (_, _, m) -> [|Maths.((F m * c.(0))
                             + (F 1. - F m) * g * g); c.(1)|]


  ...
```

Listing 5: Delay-Tolerant Adaptive SGD Optimisation in Owl's Learning Rate Module

```
let update_network_server ?state params weights gradients delay loss
                          update save x_size =
  ...
  let gradient, gradient_back = gradients in
  ...
  let us' = match params.learning_rate with
  | AdaptiveRev _ ->
    (* calculate old learning rate *)
    let old_l = Checkpoint.(
      Owl_utils.aarr_map2 (fun g' c ->
        Maths.(rate_fun state.current_batch g' c)
      ) gradient state.ch
    )
    in
    (* update gcache *)
    Checkpoint.(state.ch <- Owl_utils.aarr_map3 (fun g gb c ->
      let z = Maths.((c.(0) + (g * g)) + (F 2. * g * gb)) in
      [|z; max z c.(1)|]
    ) gradient gradient_back state.ch);
    (* calculate new learning rate *)
    let new_l = Checkpoint.(
      Owl_utils.aarr_map2 (fun g' c ->
        Maths.(rate_fun state.current_batch g' c)
      ) gradient state.ch
    ) in
    (* adjust direction based on old and new learning_rate *)
    Checkpoint.(
      Owl_utils.aarr_map4 (fun p' l l' gb ->
        Maths.((p' * l) + ((l' - l) * gb))
      ) ps' new_l old_l gradient_back
    )
      ...
```

Listing 6: AdaptiveRevision in Server Update

```
let update_network_server ?state params weights gradients delay loss
                          update save x_size =
  ...
  let gradient, gradient_back = gradients in
  ...
  let us' = match params.learning_rate with
  | AdaDelay a ->
    (* update gcache *)
    Checkpoint.(state.ch <- Owl_utils.aarr_map2 (fun g c ->
      let i = F (float_of_int state.current_batch) in
      let z = Maths.(c.(0) + ((i / (i + (F (float_of_int delay))))
              * (g * g)))
      in
      [|z; c.(1)|]
    ) gradient state.ch);

    (* calculate delayed learning rate. c_j *)
    let delay_l = Checkpoint.(
      Owl_utils.aarr_map2 (fun g' c ->
        Maths.(rate_fun state.current_batch g' c)
      ) gradient state.ch
    ) in

    (* take smaller/larger steps depending on delay *)
    let mod_l = Checkpoint.(
      Owl_utils.aarr_map (fun dl ->
        Maths.(F a /
              ((dl * sqrt ((F (float_of_int state.current_batch))
              + (F (float_of_int delay))))
              + F 1e-32)
              )
      ) delay_l
    ) in

    (* adjust direction based on learning_rate *)
    Checkpoint.(
      Owl_utils.aarr_map2 (fun p' l->
        Maths.(p' * l)
      ) ps' mod_l
    )
      ...
```

Listing 7: AdaDelay in Server Update

```
let update_network_server ?state params weights gradients delay loss
                          update save x_size =
  ...
  let gradient, gradient_back = gradients in
  ...
  let us' = match params.learning_rate with
  | DelayComp (a, _, _) ->
    (* meansquare for adaptive variance control *)
    Checkpoint.(state.ch <- Owl_utils.aarr_map2 upch_fun gradient state.ch);

    (* calculate variance control *)
    let vc = Checkpoint.(
      Owl_utils.aarr_map2 (fun g' c ->
        Maths.(rate_fun state.current_batch g' c)
      ) gradient state.ch
    ) in

    (* calculate delay revision step *)
    let delay_l =  Checkpoint.(
      Owl_utils.aarr_map3 (fun vc' g' gb ->
        Maths.(vc' * g' * g' * gb)
      ) vc gradient gradient_back
    ) in

    Checkpoint.(
      Owl_utils.aarr_map2 (fun p' l' ->
        Maths.((F a) * (p' - l'))
      ) ps' delay_l
    )

  | _ ->
    (* update gcache if necessary *)
    Checkpoint.(state.ch <- Owl_utils.aarr_map2 upch_fun gradient state.ch);
    (* adjust direction based on learning_rate *)
    Checkpoint.(
      Owl_utils.aarr_map3 (fun p' g' c ->
        Maths.(p' * rate_fun state.current_batch g' c)
      ) ps' gradient state.ch
    )
  in
  ...
```

Listing 8: DC-ASGD in Server Update

```
(* retrieve the number of update iterations at
parameter server for AdaDelay *)
let local_iteration task =
  let k = (string_of_int task.sid ^ "iter") in
  try E.get k |> fst
  with Not_found -> (
    E.set k 0;
    E.get k |> fst;
  )

(* retrieve the total gradient for Adaptive Revision *)
let total_gradient task =
  let k = (string_of_int task.sid ^ "total_grad") in
  try E.get k |> fst
  with Not_found -> (
    E.set k (Owl_utils.aarr_map (fun _ -> F 0.) (M.mkpar task.model));
    E.get k |> fst;
  )

...

let schedule task workers =
  let params = task.server_params in
  (* get model, if none then init locally *)
  let model = local_model task in
  let batch_no = task.schedule_no in
  (* If AdaptiveRevision, record total gradient for worker before schedule.
     If AdaDelay, record current iteration
     If DC-ASGD, record current model *)
  let tasks = List.mapi (fun i x ->
    let _ = match params.learning_rate with
    | AdaptiveRev _   -> let total_gs = total_gradient task in
                         E.set (x ^ "gradient") total_gs
    | AdaDelay _      -> let iter = local_iteration task in
                         E.set (x ^ "iter") iter
    | DelayComp _     -> E.set (x ^ "model") (M.mkpar model)
    | _               -> () in
    (x, [(task.sid, (model, batch_no + i))])
  ) workers
  in
  task.schedule_no <- batch_no + (List.length tasks);
  tasks

...
```

Listing 9: Additional Storage and *schedule* Interface for Delay-Tolerance Algorithms

```
let pull task address vars =
  ...
  (* Calculate delay for revised learning rate *)
  let delay = match params.learning_rate with
  | AdaDelay _ -> let iter = local_iteration task in
                  let prev_iter = E.get (address ^ "iter") |> fst in
                  E.set (string_of_int task.sid ^ "iter") (iter + 1);
                  iter - prev_iter
  | _          -> 0
  in
  (* Calculate gradient for revision step *)
  let gradient_back = match params.learning_rate with
  | AdaptiveRev _ -> let gradient_old = E.get (address ^ "gradient")
                     |> fst in
                     let total_gs = total_gradient task in
                     Owl_utils.aarr_map2 (fun w u -> Maths.(w - u)) total_gs
                                         gradient_old
  | DelayComp _   -> let model_old = E.get (address ^ "model") |> fst in
                     Owl_utils.aarr_map2 (fun w u -> Maths.(w - u))
                     (M.mkpar model) model_old
  | _             -> Owl_utils.aarr_map (fun _ -> F 0.) gradient
  in
  let gradients = gradient, gradient_back in
  let state = match task.state with
    | Some state -> M.update_network ~state ~params ~init_model:false model
                    gradients delay loss xs
    | None       -> M.update_network ~params ~init_model:false model
                    gradients
                    delay loss xs
  in
  ...
  (* Update total gradient in AdaptiveRevision *)
  let _ = match params.learning_rate with
  | AdaptiveRev _ -> let total_gs = total_gradient task in
                   E.set (string_of_int task.sid ^ "total_grad")
                   (Owl_utils.aarr_map2 (fun w u -> Maths.(w + u))
                   total_gs gradient)
  | _             -> ()
  in
  ...
```

Listing 10: *pull* Interface for Delay-Tolerance Algorithms

```
type param_context = {
  ...
  mutable progressive : int; (* number of active workers allowed *)
  ...
}

(* Param barrier: Progressive Asynchronous parallel *)
(* Ignores context_step as it is not needed in async *)
let param_pasp _context =
  let filled = ref 0 in
  let l = Hashtbl.fold (fun w b l ->
    match (b = 1) with
    | true -> filled := !filled + 1;
              l
    | false ->
      (match (((List.length l) + !filled) < !_context.progressive) with
      | true  -> l @ [ w ]
      | false -> l)
  ) !_context.worker_busy []
  in (!_context.step, l)
```

Listing 11: Barrier Control in Actor

3.4 Adaptable Asynchrony

To allow the control and simulation of an arbitrary number of workers working asynchronously, I made changes to Actor and added a new barrier control, which I named *Progressive ASP* (PASP). I also added the number of workers, called *progressive*, that were allowed to be scheduled tasks to at any given point in time (Listing 11). Additionally, I added functions to retrieve, add, and remove the number of workers from the pool (Listing 12), which required a minimum number of 2 workers and a maximum number of available workers specified.

Adding and removing workers in the parameter server is as easy as simply calling these functions (Listing 13). I did my experiments in two modes: progressive and dynamic. As described in Section 2.5.2, the former slowly increased the staleness so that the training would converge smoothly to a good minimum before speeding up again. The latter tested the robustness and consistency in dealing with an arbitrary number of workers joining and leaving uniformly at random during training. I added or removed workers from the pool at random every couple of iterations and showed that the model could still converge smoothly (Section 4.3.3).

To tune the momentum, learning rate, and/or batch size with regards to the difference in the number of workers, I stored the base number of workers and base values for each component. For momentum, I calculated the total momentum as a sum of the implicit and explicit momentum as defined in Section 2.5.1. I then tuned the explicit momentum every time there was a change in the number of workers such that the total momentum remained the same throughout training (Listing 14).

```
(* actor_param.ml *)
let progressive_num () =
  match Actor_paramserver.(!_context.job_id) = "" with
  | true  -> failwith "actor_param:progressive_num"
  | false -> Actor_paramserver.(!_context.progressive)

let add_workers i =
  match Actor_paramserver.(!_context.job_id) = "" with
    | true  -> failwith "actor_param:add_workers"
    | false -> Actor_paramserver.(add_workers i)

let remove_workers i =
  match Actor_paramserver.(!_context.job_id) = "" with
    | true  -> failwith "actor_param:remove_workers"
    | false -> Actor_paramserver.(remove_workers i)

(* actor_paramserver.ml *)
let add_workers i =
  let num_worker = StrMap.cardinal !_context.workers in
  let res = !_context.progressive <> num_worker in
  let _ = match (!_context.progressive + i) > num_worker with
  | true  -> !_context.progressive <- num_worker
  | false -> !_context.progressive <- !_context.progressive + i
  in
  res

let remove_workers i =
  (* Minimum of 2 workers required *)
  let res = !_context.progressive <> 2 in
  let _ = match (!_context.progressive - i) <= 1 with
  | true  -> !_context.progressive <- 2
  | false -> !_context.progressive <- !_context.progressive - i
  in
  res
```

Listing 12: Progressive ASP in Actor's Parameter Server Module

```
  let pull task address vars =
    ...
    let current_progression = E.progressive_num () in
      (* Add/Remove workers for PASP barrier every 5 epochs *)
      let workers_changed = match Checkpoint.(state.current_batch mod
                            (state.batches_per_epoch * 5) = 0) with
        | false -> false
        (* Progressive mode: slowly increase asynchrony *)
        | true  -> E.add_workers current_progression
        (* Dynamic mode: simulating random workers *)
     (* | true  -> let b  = Owl_stats.uniform_int_rvs ~a:0 ~b:1 in
                   let aw = Owl_stats.uniform_int_rvs ~a:1 ~b:8 in
                   let lw = Owl_stats.uniform_int_rvs ~a:1 ~b:8 in
                   match b with
                   | 1 -> Owl_log.debug "%i workers attempting to join." aw;
                          E.add_workers aw
                   | _ -> Owl_log.debug "%i workers attempting to leave." lw;
                          E.remove_workers lw *)

        in
    ...
```

Listing 13: Simulating Workers in Parameter Server

```
let calc_implicit_momentum num_of_workers =
  unpack_flt Maths.(F 1. - (F 1. / F (float_of_int num_of_workers)))

(* retrieve total_momentum for PASP *)
let total_momentum task =
  let params = task.server_params in
  let k = (string_of_int task.sid ^ "total_momentum") in
  try E.get k |> fst
  with Not_found -> (
    let implicit_momentum = E.progressive_num () |> calc_implicit_momentum in
    let explicit_momentum = match params.momentum with
    | Standard m -> m
    | Nesterov m -> m
    | None       -> 0.0
    in
    E.set k (explicit_momentum +. implicit_momentum);
    E.get k |> fst;
  )
...
let pull task address vars =
  ...
  (* Detect if workers changed *)
  let _ = match workers_changed with
    | false ->  ()
    | true  ->  let w = E.progressive_num () in
                let tm = total_momentum task in
                let im = calc_implicit_momentum w in
                let em = (tm -. im) in

                match params.momentum with
                  | Standard _ -> params.momentum <- Momentum.Standard em
                  | Nesterov _ -> params.momentum <- Momentum.Nesterov em
                  | None       -> params.momentum <- Momentum.Standard em
...
```

Listing 14: Tuning Momentum in Parameter Server

Likewise, I stored the initial and current learning rate and batch size as key-value pairs in order to tune them as described in Section 2.5.2 (Listing 15). However, there was the need to propagate the new batch size to the workers. Therefore, I needed to retrieve the new batch size into the *schedule* interface and modify each worker to calculate the gradient based on the modified batch size (Listing 17). In addition, I prevented workers from being added if the training had already reached a certain number of update iterations (Listing 16).

```
  (* base batch_size for PASP *)
  let base_bs task =
    let params = task.server_params in
    let k = (string_of_int task.sid ^ "base_bs") in
    try E.get k |> fst
    with Not_found -> (
      let bs = match params.batch with
                              | Sample b           -> b
                              | Mini b             -> b
                              | Stochastic         -> 1
                              | Full               -> 0
      in
      E.set k bs;
      E.get k |> fst;
    )

  (* current batch_size for PASP *)
  let current_batch_size task =
    let params = task.server_params in
    let k = (string_of_int task.sid ^ "current_bs") in
    try E.get k |> fst
    with Not_found -> (
      E.set k params.batch;
      E.get k |> fst;
    )


  (* base learning_rate for PASP *)
  let base_lr task =
    let params = task.server_params in
    let k = (string_of_int task.sid ^ "base_lr") in
    try E.get k |> fst
    with Not_found -> (
      let a = match params.learning_rate with
                              | Adagrad a          -> a
                              | Const a            -> a
                              | AdaptiveRev a      -> a
                              | AdaDelay a         -> a
                              | DelayComp (a, v, m)-> a
                              | _                  -> 0.
      in
      E.set k a;
      E.get k |> fst;
    )

  let base_workers task =
    let k = (string_of_int task.sid ^ "base_workers") in
    try E.get k |> fst
    with Not_found -> (
      E.set k (E.progressive_num ());
      E.get k |> fst;
    )

  ...
```

Listing 15: Storing Initial and Current Learning Rate and Batch Size For Tuning

```
let pull task address vars =
  ...
  let _ = match workers_changed with
      | false ->  ()
      | true  ->
            (* Increase batch size *)
            let bs = base_bs task |> float_of_int in
            let lr = base_lr task in
            let w = base_workers task in
            let w' = E.progressive_num () in
            let d = (w' - w) in
            Owl_log.debug "Worker count changed to %i" w';
            let d = float_of_int d in
            let nlr = lr *. (exp (-0.15 *. d)) in
            let nbs = bs *. (lr /. nlr) |> int_of_float in
            Owl_log.debug "New Batch Size %i" nbs;
            (* Check if new batch size affects training *)
            let batches_per_epoch = match params.batch with
                  | Full       -> 1
                  | Mini _     -> xs / nbs
                  | Sample _   -> xs / nbs
                  | Stochastic -> xs
                  in
            let batches = (float_of_int batches_per_epoch) *. params.epochs
                          |> int_of_float in
            Owl_log.debug "Iterations: %i" batches;
            match Checkpoint.(state.current_batch) >= batches with
            | true  -> Owl_log.debug "Batch size too big.
                                      Removing recently added workers..";
                       let _ = E.remove_workers (w' - current_progression) in
                       ()
            | false ->
              let _ = match params.batch with
              | Sample _    -> params.batch <- Sample nbs
              | Mini _      -> params.batch <- Mini nbs
              | Stochastic  -> params.batch <- Mini nbs
              | Full        -> ()
              in
              E.set (string_of_int task.sid ^ "current_bs") params.batch;
              Owl_log.debug "New Learning Rate: %f" nlr;
              match params.learning_rate with
              | Adagrad _          -> params.learning_rate <- Adagrad nlr
              | Const _            -> params.learning_rate <- Const nlr
              | AdaptiveRev _      -> params.learning_rate <- AdaptiveRev nlr
              | AdaDelay _         -> params.learning_rate <- AdaDelay nlr
              | DelayComp (_, v, m)-> params.learning_rate
                                        <- DelayComp (nlr, v, m)
              | _                  -> ()
  in
  ...
```

Listing 16: Tuning Learning Rate and Batch Size in *pull* Interface

```
let schedule task workers =
  ...
  let cb = current_batch_size task in
  ...
    (x, [(task.sid, (model, batch_no + i, cb))])
  ) workers
  in

...

let push task id vars =
  ...
  task.client_params.batch <- bs;
  let params = task.client_params in
  let x = task.train_x in
  let y = task.train_y in
  let grad, loss = M.calculate_gradient ~params ~init_model:false model x y t in
  let result = (grad, loss) in
  ...
in
...
```

Listing 17: Scheduling Task with New Batch Size

Chapter 4

Results

4.1 Experimental Setup

I conducted my experiments using Owl+Actor. All my experiments were done on a single machine that contained 6 Intel Xeon Processor E5-2430L v2 (15M Cache, 2.40 GHz) and over 64 GB of memory. I did not simulate delay or introduce stragglers and could only support a small number of workers. Therefore, the setting in which I ran my experiments was unlike those in huge cloud computing networks. As using higher batch size required a large memory footprint, I had to reduce the number of workers for some experiments with Adaptable Asynchrony.

Unless mentioned otherwise, all experiments were done on the CIFAR-10 dataset [72], which I now refer to as CIFAR for the rest of my writing, using a small VGG-like neural network with the following layers:

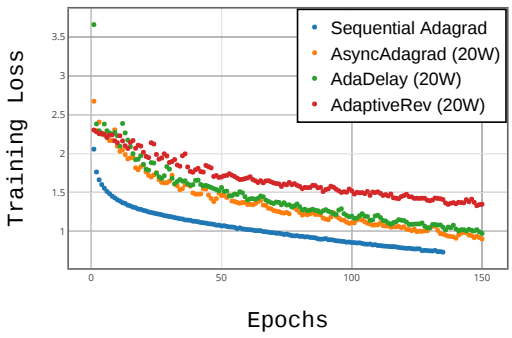
1. Normalisation with 0.9 decay
2. Convolution using 32 filters
3. Convolution using 32 filters + Max pooling
4. Convolution using 64 filters
5. Convolution using 64 filters + Max pooling
6. Fully connected with 512 nodes with ReLU activation
7. Output layer with Softmax activation

Note that all convolution layers used a 3∗3 convolution window and all max-pooling layers used a $2 * 2$ pool size. I also used a dropout rate of 0.1 after every max pooling layer. I do not aim to achieve optimal results but to study the effects of varying asynchrony

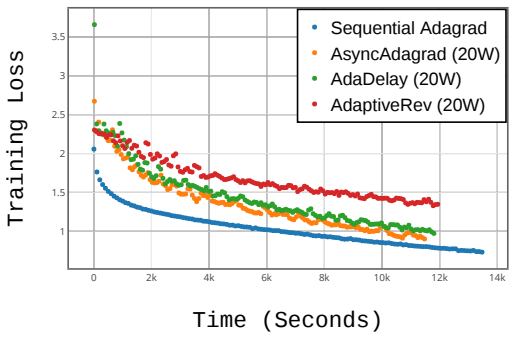
using adaptive SGD optimisation algorithms in a small cluster. All figures are generated using Plot.ly[1].

4.2 Delay-Tolerant Adaptive Optimisation

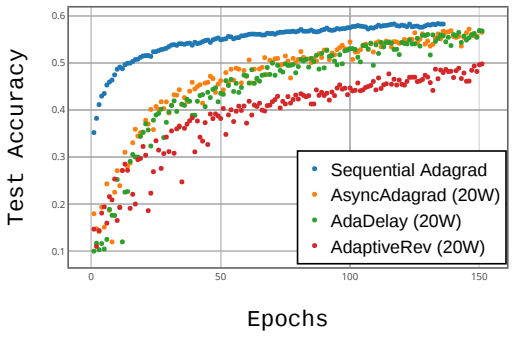
Some of the state-of-the-art adaptive SGD optimisation algorithms were designed to mitigate the negative impact of the huge delays caused by stragglers in convex optimisation. I decided to implement some of these algorithms such as AdaptiveRevision and AdaDelay in order to observe the behaviour of these delay-tolerant algorithms in a non-convex setting without delay stimulation. I tried $\alpha = [0.001, 0.003, 0.005]$ and found that $\alpha = 0.001$ resulted in a better rate of convergence and test accuracy for AsyncAdagrad and AdaDelay and $\alpha = 0.003$ for AdaptiveRevision. In practice, tuning hyper-parameters is challenging because stability is influenced by the asynchronous nature and dependent on the behaviour of the systems [64].



(a) Epochs to Converge



(b) Time to Converge



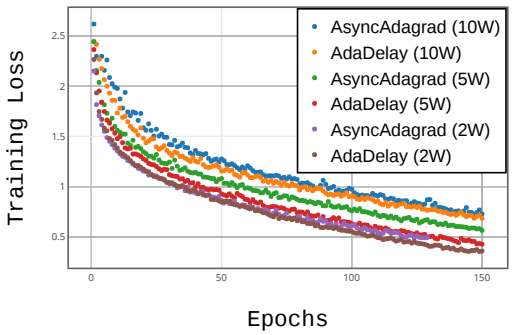
(c) Test Accuracy

Figure 4.1: Delay Tolerance Adaptive SGD Optimisation Algorithms

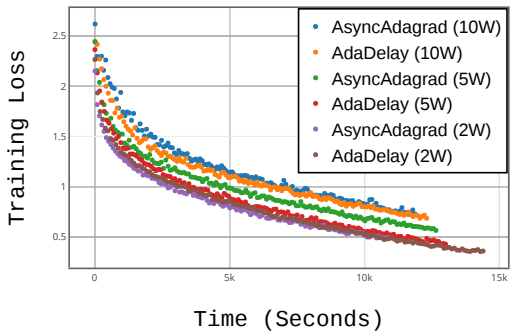
[1]https://plot.ly/

In Figure 4.1, I averaged the loss of update iterations in every epoch and plotted the rate of convergence, time to converge, and test accuracy. Both AdaptiveRevision and AdaDelay required more local computation time as extra steps were taken to calculate and include the delay in the model update. Although I could not find any convergence analysis of these algorithms in a non-convex setting, my intuition was that AdaDelay would perform better than AdaptiveRevision as AdaDelay does not assume worst-case bounds and is not monotonically decreasing whereas AdaptiveRevision relied on a pessimistic worst-case delay.

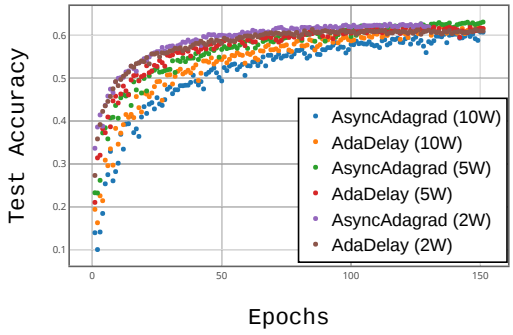
My experiments showed that AdaptiveRevision's approach to take smaller steps might cause the model to converge to a worse minimum in a non-convex setting. Therefore, I decided to omit AdaptiveRevision in future experiments and advocate testing it again in a high-delayed environment. On the other hand, even though staleness was small in my cluster, AdaDelay performed either similarly or slightly better than AsyncAdagrad.



(a) Epochs to Converge



(b) Time to Converge



(c) Test Accuracy

Figure 4.2: AsyncAdagrad and AdaDelay

I repeated the experiments with a varying number of workers (Figure 4.2) and found that AdaDelay consistently performed slightly better than AsyncAdagrad. Although designed

for high-delay situations, AdaDelay's non-monotonically decreasing step sizes showed to be great at mitigating the effects of staleness in a deep neural network. I hypothesise that if I were to scale the number of workers up or simulate delay, AdaDelay would be superior to AsyncAdagrad.

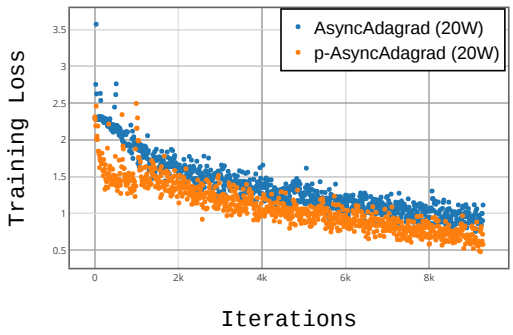
4.3 Adaptable Asynchrony

My motivation for AA stemmed from not only a combination of recent papers, as discussed in Section 2.5.2, but also the empirical observations of applying these techniques in my situation. I detail the empirical motivations that led to AA in Section 4.3.1. After which, I show that using AA improved the rate of convergence, stability, and test accuracy when used with AdaDelay in Section 4.3.2. Lastly, I demonstrate that AA maintains training stability in a dynamic environment by simulating the addition and removal of workers during training.

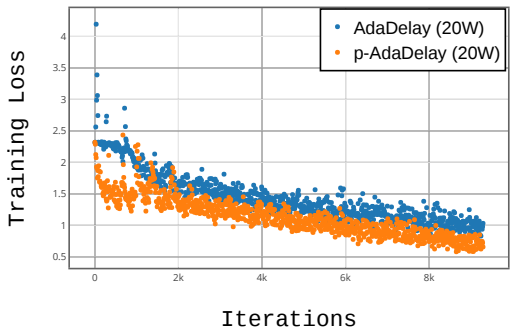
4.3.1 Initial Observations

In Section 3.1.3, I observed that increasing asynchrony led to not only a slower convergence rate but also a different minimum. I was curious whether starting off with a small pool of workers and then adding workers during training would help to achieve a better minimum. I hypothesise that by slowly increasing asynchrony and thus increasing staleness while training, the model would converge to a better minimum at a cost of additional time. I call this setting "Progressive" and name my training under this setting "p-". Although an approach to achieve progressiveness was to schedule the task to all workers and then simply sample and discard updates as seen in previous examples [37], [42], I decided to incrementally schedule to the desired number of workers to avoid unnecessary local computation. However, I recommend the former approach for training in networks with high delays or the presence of stragglers to increase fault tolerance.

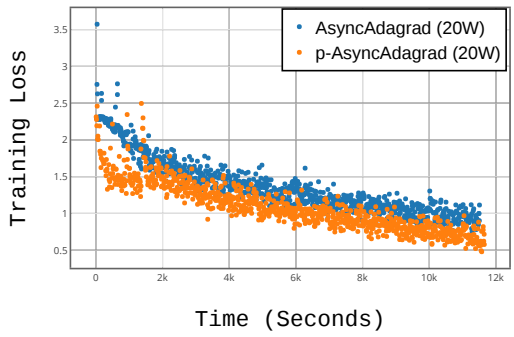
I ran the experiments with $\alpha = 0.001$, mini-batch size of 128, max epoch of 150 and a patience value of 20. Starting from 2 workers, I increased the number of workers involved by a factor of 2 every 5 epochs up to the maximum number of workers specified. Instead of averaging the loss and plotting it against the number of epochs, I plotted the loss against the number of update iterations for every 10 iterations in order to show the effects of increasing asynchrony during training.



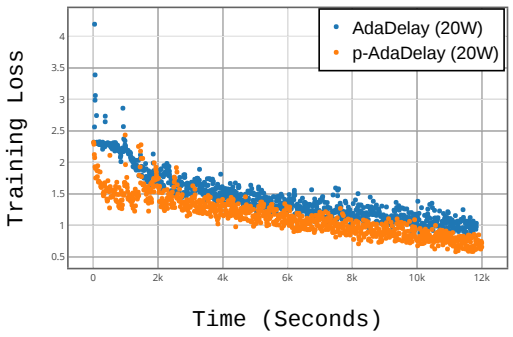
(a) Iterations to Converge (AsyncAdagrad)



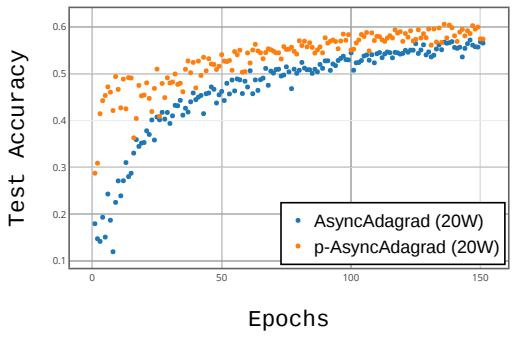
(b) Iterations to Converge (AdaDelay)



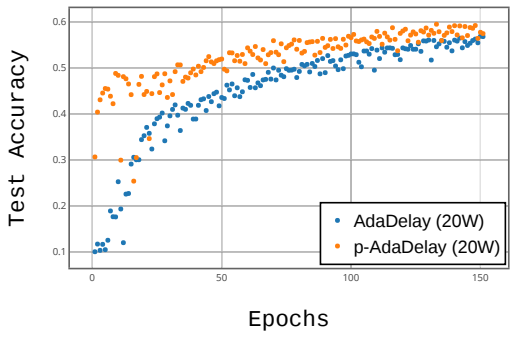
(c) Time to Converge (AsyncAdagrad)



(d) Time to Converge (AdaDelay)



(e) Test Accuracy (AsyncAdagrad)



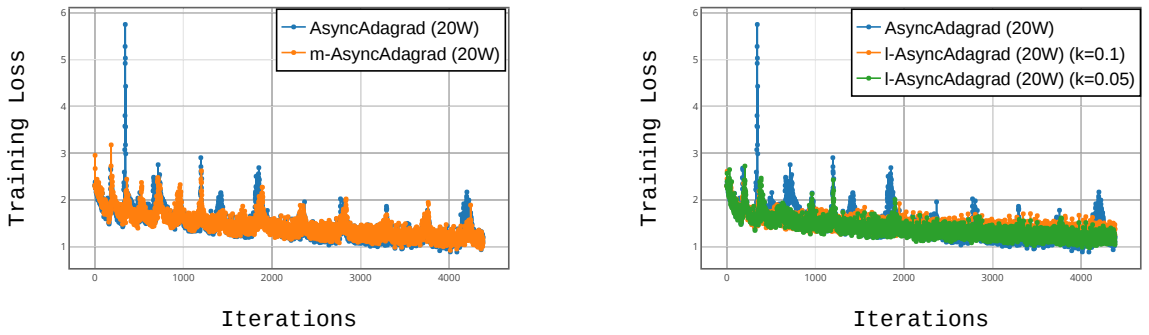
(f) Test Accuracy (AdaDelay)

Figure 4.3: Progressive Mode

Although the results in Figure 4.3 confirmed my earlier hypothesis, I expected a more significant increase in the total training time. I showed that by starting the training with a small pool of workers and slowly incrementing it exponentially during training, I was able to allow the model to convergence faster to a better minimum with a slight extra increase in total time by about 1.2%. I hypothesise that the small staleness count helped

the model to converge to a better minimum from the beginning and, thus, speed-up to a better result later. However, I realized that every time I increased the asynchrony, there would be a spike in both AsyncAdagrad and AdaDelay and more training iterations were needed to get the model to converge smoothly again. Chen et al. [37] also reported that slowly increasing asynchrony during training did not help to stabilize results. Although they tried reducing the learning rate to manage these spikes when staleness was at least 20, they could not prevent some runs from diverging. I reckoned that I did not deploy enough workers or simulate sufficient staleness to observe divergence but would expect so as a greater increase in staleness would result in a greater spike.

In order to empirically observe the effect of increasing and decreasing staleness during training, I trained the model for 45 epochs, increasing the number of workers by threefold every 2 epochs up to the max worker count and decreasing the number of workers by twofold every 3 epochs. As expected, I observed a series of explosions every time workers were added in Figure 4.4. I first tried to tune the momentum term and termed it "m-", as Mitliagkas et al. [74] did to offset the momentum caused by asynchrony in ASGD. However, as mentioned in Section 2.6, doing so with Adagrad did not seem intuitive and my results supported my intuition.



(a) Tuning Momentum

(b) Tuning Learning Rate

Figure 4.4: Effects of Increasing Asynchrony During Training

I tuned the learning rate and termed it as "l-", as described in Section 2.5.2. Therefore, the learning rate increased if the number of workers dropped and decreased if the number of workers rose. I observed a trade-off between using a lower and a higher decay, k. As shown in Figure 4.4, the model might not converge to the same minimum with a higher k and a lower k might not sufficiently mitigate the explosions and smooth the rate of convergence. Regardless, tuning the learning rate alone did not seem to solve instability without sacrificing accuracy. Therefore, I tuned both the batch size and the learning rate based on the number of workers changed.

4.3.2 Effectiveness

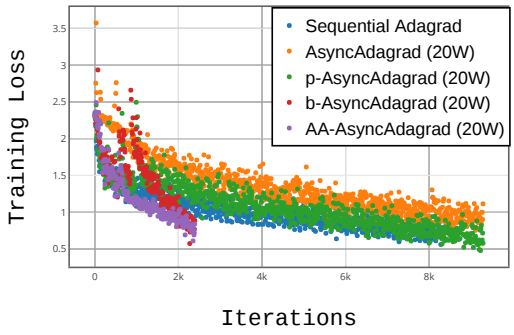
As shown in the previous section (Section 4.3.1), tuning the learning rate alone is not sufficient in maintaining stability and improving results. Therefore, I tried tuning the batch size only, and termed it "-b", as described in Section 2.5.2 only on AsyncAdagrad. For instance, I increased the batch size instead of decaying the learning rate. Lastly, I deployed AA to show its effectiveness in terms of stabilizing the model without sacrificing accuracy. I tried $k = [0.025, 0.075, 0.1, 0.15, 0.2]$ and found that $k = 0.075$ achieved the best test accuracy with a relatively better stability as compared to the rest. I also used a higher initial learning rate that would be tuned to the best learning rate found in my previous experiments which started with the max number of workers. For instance, since $\alpha = 0.001$ was the best learning rate if I trained with 20 workers throughout and $k = 0.075$, I would start at $\alpha = 0.004$ on two workers such that the learning rate would be tuned to be closed to 0.001 when the number of workers hit 20. Note that as the mini-batch size increased, the number of batches per epoch decreased, leading to fewer iterations and a faster training time.

I could observe from the AsyncAdagrad case in Figure 4.5 that although increasing the batch size sped up training, it affected the rate of convergence and increased the frequency of spikes. Additionally, test accuracy varied heavily and training was not stable. Decaying the learning rate only (as shown in Section 4.3.1) would result in a smoother rate of convergence but might lead the model to converge to a similar or better minimum. Decaying the learning rate while increasing the batch size, on the other hand, reaped the benefits of both worlds. Furthermore, I realized that deploying AA with AdaDelay caused the model to converge to a better minimum, further mitigating the effects of staleness.

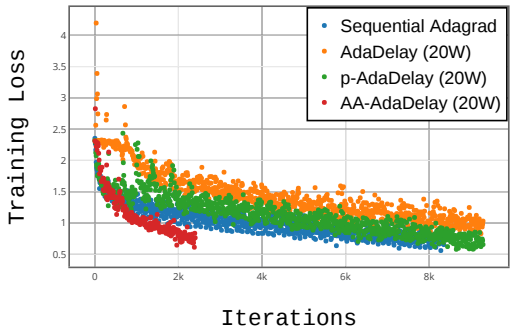
4.3.3 Consistency

I pondered if AA could work in a dynamic environment in which workers were joining and leaving the network uniformly at random, a situation which I labelled "d-". Being able to stabilise the rate of convergence and maintain test accuracy regardless of the number of workers during training would be useful for various situations. For instance, it would allow the training of a model in an arbitrary network with a varying number of available nodes and in a distributed P2P network where the number of workers could fluctuate.

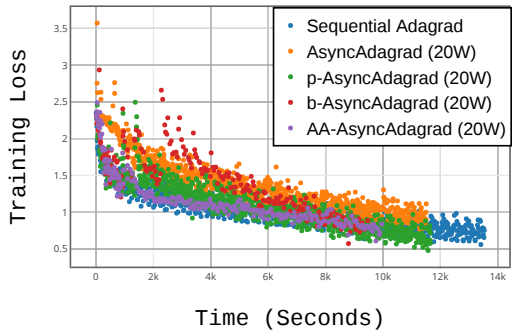
In order to simulate a dynamic setting, I added or removed an arbitrary number of workers at random every 100 iterations, starting with 2, as shown in Listing 13. Due to limited resources, I conducted my experiments with a maximum worker pool of 10. I rejected additional workers during training if their introduction would result in a higher total number of iterations than the current iteration. In practice, a parameter server



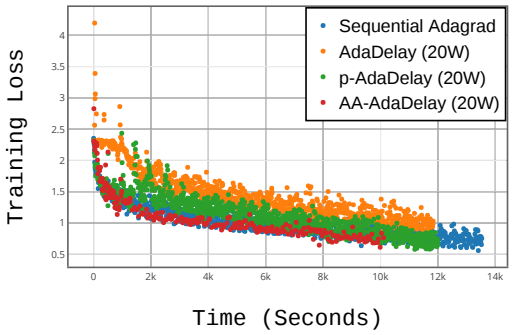
(a) Iterations to Converge (AsyncAdagrad)



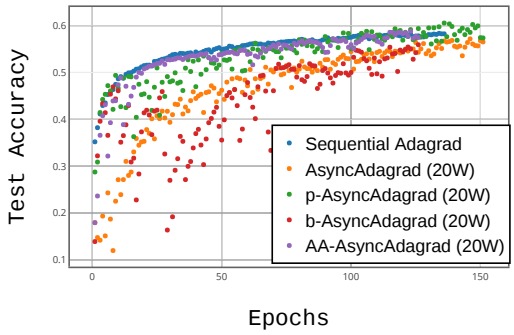
(b) Iterations to Converge (AdaDelay)



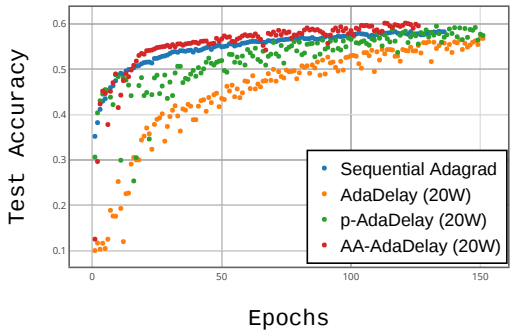
(c) Time to Converge (AsyncAdagrad)



(d) Time to Converge (AdaDelay)



(e) Test Accuracy (AsyncAdagrad)

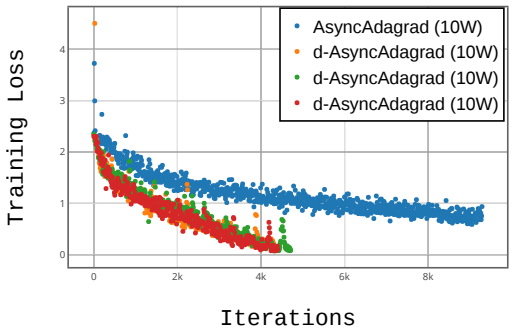


(f) Test Accuracy (AdaDelay)

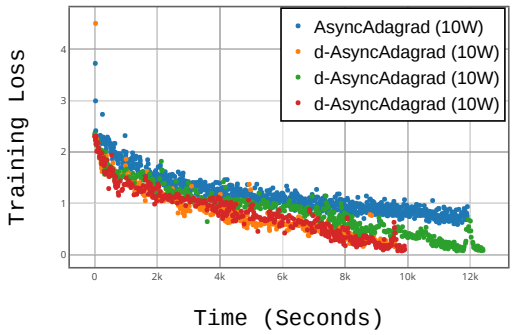
Figure 4.5: Adaptable Asynchrony

or a P2P network could stop additional workers from joining the training process if the training was about to end.

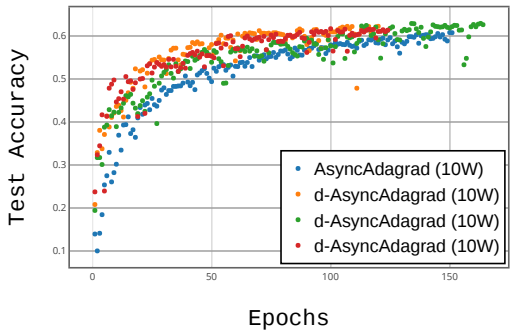
Lian et al. [67] suggested that a linear speedup was achievable if the maximum number of workers did not exceed $\sqrt{K}$ where K is the total number of iterations for ASGD. However, I could not find any theoretical backup for approaches apart from ASGD and did not have the resources to observe an upper limit. As usual, I started with a mini-batch size of 128, a patience value of 30, and a max epoch of 150. I used higher k values $[0.1, 0.15, 0.2]$ as I predicted higher instability in a dynamic environment. I found that although training with $k = 0.2$ achieved the best stability, it required more iterations for the model to converge to a similar accuracy. Therefore, I chose $k = 0.15$ as a trade-off between convergence rate and stability.



(a) Iterations to Converge



(b) Time to Converge



(c) Test Accuracy

Figure 4.6: AA-AsyncAdagrad in a Dynamic Environment

I expected vastly different paths of convergence before running the experiments. Surprisingly, I observed in Figure 4.6 that uniformly and randomly sampled workers led to similar convergence paths. I speculate that such occurrences were due to the averaging of paths between 2 to 10 workers and would occur if the rate of workers joining and leaving

were similar. If the rate of workers joining were more than the rate of workers leaving, I would expect a steeper training curve. Likewise, if there were more workers leaving than workers joining, I would expect a curve to be similar to that of training the model with only two workers throughout. Therefore, I hypothesise that different rates of workers joining and leaving will result in a different minimum. Regardless, I presume that the model will still converge in a stable manner and certain rates may require more epochs for the model to reach the same generalisation. Additionally, my initial experiments made us question if training a model in a dynamic environment in which asynchrony is always uniformly and randomly fluctuating will help improve performance in general in a larger scale experiment.

Chapter 5

Summary and Conclusions

Techniques to handle both the straggler and staleness problem will become more prominent as the number of data increases, putting more emphasis on distributed training. I redesigned the parameter server engine in Owl+Actor to support adaptive SGD optimisation, showed the effects of varying asynchrony, and demonstrated that test accuracy could be improved by slowly increasing asynchrony during training. By increasing the rate of scheduling exponentially while training, large spikes and instability would occur, costing more update iterations for the model to converge and occasionally causing divergence. I observed that deploying popular techniques, such as tuning momentum, to mitigate staleness were not as useful on AsyncAdagrad as they were on SGD. Therefore, I proposed AA as a technique with AsyncAdagrad to ensure stability, improve the rate of convergence, and maintain or even improve test accuracy. I also showed empirically that AA could be used together AdaDelay to further improve results. Additionally, AA could also be conducted in a dynamic environment in which the number of workers changed uniformly at random.

The aim of this report is not to achieve art-of-the-state optimal results but to show the effects of staleness on popular optimisation techniques that would otherwise be effective in other approaches. I foresee the need to gradually increase the learning rate at the start to ensure smooth convergence if I were to further increase asynchrony. I hypothesise that using AA with delay-tolerant adaptive SGD optimization algorithms will perform much better in an environment in which stragglers are present or network delays are high. I also observed that having a small extent of staleness could help to prevent over-fitting and improve accuracy. Thus, I propose viewing asynchrony not only as a method to accommodate large models and data or an approach to speed-up training but also as a regularisation technique to improve the model generalisation.

AA was proposed based on two intuitions: a small amount of staleness at the start would

enable the model to find a better minimum and tuning the learning rate and batch size to cope with the additional noise from asynchrony. Therefore, a future direction will be to tune the hyper-parameters and level of asynchrony based on theoretical guarantees on adaptive SGD optimization to completely negate the effects of asynchrony similar to the effect of momentum on asynchronous SGD. In the meantime, results show that AA is a viable approach to improve performance. Lastly, a potential immediate future direction will be to scale up the number of workers in the dynamic environment and observe if training in asynchrony uniformly at random will improve generalisation given the stability that AA offers.

Bibliography

[1] F. Hill, K. Cho, A. Korhonen, and Y. Bengio, "Learning to Understand Phrases by Embedding the Dictionary", *CoRR*, vol. abs/1504.00548, 2015. arXiv: 1504.00548. [Online]. Available: http://arxiv.org/abs/1504.00548.

[2] A. L. Buczak and E. Guven, "A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection", *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1153–1176, 2016.

[3] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio, "End-to-End Attention-based Large Vocabulary Speech Recognition", in *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, IEEE, 2016, pp. 4945–4949.

[4] S. Bhattacharya and N. D. Lane, "From Smart to Deep: Robust Activity Recognition on Smartwatches using Deep Learning", in *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, IEEE, 2016, pp. 1–6.

[5] E. P. Xing, Q. Ho, P. Xie, and D. Wei, "Strategies and Principles of Distributed Machine Learning on Big Data", *Engineering*, vol. 2, no. 2, pp. 179–195, 2016.

[6] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling Distributed Machine Learning with the Parameter Server.", in *OSDI*, vol. 14, 2014, pp. 583–598.

[7] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, *et al.*, "Large Scale Distributed Deep Networks", in *Advances in Neural Information Processing Systems*, 2012, pp. 1223–1231.

[8] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an Efficient and Scalable Deep Learning Training System.", in *OSDI*, vol. 14, 2014, pp. 571–582.

[9] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[10] J. C. Duchi, M. I. Jordan, and H. B. McMahan, "Estimation, Optimization, and Parallelism when Data is Sparse or Highly Varying", 2013.

[11] S. Sra, A. W. Yu, M. Li, and A. J. Smola, "AdaDelay: Delay Adaptive Distributed Stochastic Convex Optimization", *ArXiv e-prints*, 2015. arXiv: 1508.05003.

[12] L. Wang, "Owl: A General-Purpose Numerical Library in OCaml", *CoRR*, vol. abs/1707.09616, 2017. [Online]. Available: http://arxiv.org/abs/1707.09616.

[13] L. Wang. (2018). Owl's Parallel & Distributed Computing Engine, [Online]. Available: https://github.com/owlbarn/actor.

[14] R. Ge, F. Huang, C. Jin, and Y. Yuan, "Escaping from saddle points—online stochastic gradient for tensor decomposition", in *Conference on Learning Theory*, 2015, pp. 797–842.

[15] R. Johnson and T. Zhang, "Accelerating Stochastic Gradient Descent using Predictive Variance Reduction", in *Advances in Neural Information Processing Systems*, 2013, pp. 315–323.

[16] A. Defazio, F. Bach, and S. Lacoste-Julien, "SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives", in *Advances in Neural Information Processing Systems*, 2014, pp. 1646–1654.

[17] S. J. Reddi, A. Hefny, S. Sra, B. Poczos, and A. J. Smola, "On Variance Reduction in Stochastic Gradient Descent and its Asynchronous Variants ", in *Advances in Neural Information Processing Systems*, 2015, pp. 2647–2655.

[18] R. Leblond, F. Pedregosa, and S. Lacoste-Julien, "ASAGA: Asynchronous Parallel SAGA", *ArXiv e-prints*, arXiv: 1606.04809.

[19] H. Mania, X. Pan, D. Papailiopoulos, B. Recht, K. Ramchandran, and M. I. Jordan, "Perturbed iterate analysis for asynchronous stochastic optimization", *ArXiv preprint arXiv:1507.06970*, 2015.

[20] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization Methods for Large-Scale Machine Learning", *ArXiv e-prints*, arXiv: 1606.04838.

[21] N. Qian, "On the Momentum Term in Gradient Descent Learning Algorithms", *Neural Networks*, vol. 12, no. 1, pp. 145–151, 1999.

[22] Y. Nesterov, "A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence O (1/kˆ 2)", in *Doklady AN USSR*, vol. 269, 1983, pp. 543–547.

[23] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the Importance of Initialization and Momentum in Deep Learning", in *International Conference on Machine Learning*, 2013, pp. 1139–1147.

[24] M. D. Zeiler, "Adadelta: An Adaptive Learning Rate Method", 2012.

[25] H. Geoffrey, S. Nitish, and S. Kevin, "Overview of mini-batch gradient descent", 2014, [Online]. Available: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[26] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization", *ArXiv preprint arXiv:1412.6980*, 2014.

[27] T. Dozat, "Incorporating Nesterov Momentum into Adam", 2016.

[28] S. Hadjis, F. Abuzaid, C. Zhang, and C. Ré, "Caffe Con Troll: Shallow Ideas to Speed Up Deep Learning", in *Proceedings of the Fourth Workshop on Data analytics in the Cloud*, ACM, 2015, p. 2.

[29] "Deep image: Scaling up image recognition", *CoRR*, vol. abs/1501.02876, 2015, Withdrawn. arXiv: 1501.02876. [Online]. Available: http://arxiv.org/abs/1501.02876.

[30] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics.", in *NSDI*, 2016, pp. 363–378.

[31] X. Pan, S. Venkataraman, Z. Tai, and J. Gonzalez, "Hemingway: Modeling Distributed Optimization Algorithms", *CoRR*, vol. abs/1702.05865, 2017. arXiv: 1702.05865. [Online]. Available: http://arxiv.org/abs/1702.05865.

[32] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "On Parallelizability of Stochastic Gradient Descent for Speech DNNs", in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, IEEE, 2014, pp. 235–239.

[33] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-Bit Stochastic Gradient Descent and its Application to Data-parallel Distributed Training of Speech DNNs", in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[34] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep Learning with COTS HPC Systems", in *International Conference on Machine Learning*, 2013, pp. 1337–1345.

[35] Q. V. Le, "Building High-level Features using Large Scale Unsupervised Learning", in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, IEEE, 2013, pp. 8595–8598.

[36] X. Chen, A. Eversole, G. Li, D. Yu, and F. Seide, "Pipelined Back-propagation for Context-dependent Deep Neural Networks", in *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.

[37] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd", *ArXiv preprint arXiv:1604.00981*, 2016.

[38] H. Li, A. Kadav, E. Kruus, and C. Ungureanu, "Malt: Distributed Data-Parallelism for Existing ML Applications", in *Proceedings of the Tenth European Conference on Computer Systems*, ACM, 2015, p. 3.

[39] N. Strom, "Scalable Distributed DNN Training using Commodity GPU Cloud Computing", in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[40] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. Pietzuch, "Ako: Decentralised Deep Learning with Partial Gradient Exchange", in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ACM, 2016, pp. 84–97.

[41] N. D. Vanli, M. O. Sayin, I. Delibalta, and S. S. Kozat, "Sequential Nonlinear Learning for Distributed Multiagent Systems via Extreme Learning Machines", *IEEE transactions on neural networks and learning systems*, vol. 28, no. 3, pp. 546–558, 2017.

[42] L. Wang, B. Catterall, and R. Mortier, "Probabilistic synchronous parallel", *ArXiv preprint arXiv:1709.07772*, 2017.

[43] S. Hadjis, C. Zhang, I. Mitliagkas, and C. Ré, "Omnivore: An optimizer for multi-device deep learning on cpus and gpus", *CoRR*, vol. abs/1606.04487, 2016. arXiv: 1606.04487. [Online]. Available: http://arxiv.org/abs/1606.04487.

[44] B. C. Ooi, K.-L. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, A. K. Tung, Y. Wang, *et al.*, "SINGA: A Distributed Deep Learning Platform", in *Proceedings of the 23rd ACM International Conference on Multimedia*, ACM, 2015, pp. 685–688.

[45] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems", *CoRR*, vol. abs/1512.01274, 2015. arXiv: 1512.01274. [Online]. Available: http://arxiv.org/abs/1512.01274.

[46] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-specialized Parameter Server", in *Proceedings of the Eleventh European Conference on Computer Systems*, ACM, 2016, p. 4.

[47] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Managed communication and consistency for fast data-parallel iterative analytics", in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ACM, 2015, pp. 381–394.

[48] H. Zhang, Z. Hu, J. Wei, P. Xie, G. Kim, Q. Ho, and E. P. Xing, "Poseidon: A System Architecture for Efficient GPU-based Deep Learning on Multiple Machines", *CoRR*, vol. abs/1512.06216, 2015. arXiv: `1512.06216`. [Online]. Available: `http://arxiv.org/abs/1512.06216`.

[49] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server", in *Advances in Neural Information Processing Systems*, 2013, pp. 1223–1231.

[50] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mccauley, M. Franklin, S. Shenker, and I. Stoica, "Fast and interactive analytics over Hadoop data with Spark", *Usenix Login*, vol. 37, no. 4, pp. 45–51, 2012.

[51] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan, "SparkNet: Training Deep Networks in Spark", *ArXiv e-prints*, 2015. arXiv: `1511.06051`.

[52] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, *et al.*, "Communication-Efficient Learning of Deep Networks from Decentralized Data", *ArXiv preprint arXiv:1602.05629*, 2016.

[53] W. F. McColl, "Bulk Synchronous Parallel Computing", in *Abstract Machine Models for Highly Parallel Computers*, Oxford University Press, 1995, pp. 41–63.

[54] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud", *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[55] W. Dai, A. Kumar, J. Wei, Q. Ho, G. A. Gibson, and E. P. Xing, "High-Performance Distributed ML at Scale through Parameter Server Consistency Models", *CoRR*, vol. abs/1410.8043, 2014. [Online]. Available: `http://arxiv.org/abs/1410.8043`.

[56] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized Stochastic Gradient Descent", in *Advances in Neural Information Processing Systems*, 2010, pp. 2595–2603.

[57] Y. Zhang, M. J. Wainwright, and J. C. Duchi, "Communication-Efficient Algorithms for Statistical Optimization", in *Advances in Neural Information Processing Systems*, 2012, pp. 1502–1510.

[58] J. Chen, R. Monga, S. Bengio, and R. Józefowicz, "Revisiting distributed synchronous SGD", *CoRR*, vol. abs/1604.00981, 2016. arXiv: `1604.00981`. [Online]. Available: `http://arxiv.org/abs/1604.00981`.

[59] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "100-epoch ImageNet Training with Alexnet in 24 Minutes", *ArXiv e-prints*, 2017.

[60] D. Das, S. Avancha, D. Mudigere, K. Vaidyanathan, S. Sridharan, D. D. Kalamkar, B. Kaul, and P. Dubey, "Distributed Deep Learning Using Synchronous Stochastic Gradient Descent", *CoRR*, vol. abs/1602.06709, 2016. arXiv: `1602.06709`. [Online]. Available: `http://arxiv.org/abs/1602.06709`.

[61] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A Lock-free Approach to Parallelizing Stochastic Gradient Descent", in *Advances in Neural Information Processing Systems*, 2011, pp. 693–701.

[62] H. R. Feyzmahdavian, A. Aytekin, and M. Johansson, "An Asynchronous Mini-batch Algorithm for Regularized Stochastic Optimization", *IEEE Transactions on Automatic Control*, vol. 61, no. 12, pp. 3740–3754, 2016.

[63] T. Paine, H. Jin, J. Yang, Z. Lin, and T. S. Huang, "GPU Asynchronous Stochastic Gradient Descent to Speed Up Neural Network Training", *CoRR*, vol. abs/1312.6186, 2013. arXiv: 1312.6186. [Online]. Available: http://arxiv.org/abs/1312.6186.

[64] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-Aware Async-SGD for Distributed Deep Learning", *CoRR*, vol. abs/1511.05950, 2015. [Online]. Available: http://arxiv.org/abs/1511.05950.

[65] T. Kurth, J. Zhang, N. Satish, E. Racah, I. Mitliagkas, M. M. A. Patwary, T. Malas, N. Sundaram, W. Bhimji, M. Smorkalov, *et al.*, "Deep learning at 15pf: Supervised and semi-supervised classification for scientific data", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2017, p. 7.

[66] A. Agarwal and J. C. Duchi, "Distributed Delayed Stochastic Optimization", in *Advances in Neural Information Processing Systems*, 2011, pp. 873–881.

[67] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization", in *Advances in Neural Information Processing Systems*, 2015, pp. 2737–2745.

[68] C. Noel and S. Osindero, "Dogwild!-Distributed Hogwild for CPU & GPU", in *NIPS Workshop on Distributed Machine Learning and Matrix Computations*, 2014.

[69] I. Mitliagkas, M. Borokhovich, A. G. Dimakis, and C. Caramanis, "FrogWild!: Fast PageRank Approximations on Graph Engines", *Proceedings of the VLDB Endowment*, vol. 8, no. 8, pp. 874–885, 2015.

[70] S. Zhang, A. E. Choromanska, and Y. LeCun, "Deep learning with Elastic Averaging SGD", in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., Curran Associates, Inc., 2015, pp. 685–693. [Online]. Available: http://papers.nips.cc/paper/5761-deep-learning-with-elastic-averaging-sgd.pdf.

[71] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z. Ma, and T. Liu, "Asynchronous Stochastic Gradient Descent with Delay Compensation for Distributed Deep Learning", *CoRR*, 2016. [Online]. Available: http://arxiv.org/abs/1609.08326.

[72] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images", 2009.

[73] B. McMahan and M. Streeter, "Delay-Tolerant Algorithms for Asynchronous Distributed Online Learning", in *Advances in Neural Information Processing Systems*, 2014, pp. 2915–2923.

[74] I. Mitliagkas, C. Zhang, S. Hadjis, and C. Ré, "Asynchrony begets Momentum, with an Application to Deep Learning", *ArXiv e-prints*, arXiv: 1605.09774.

[75] S. L. Smith, P. Kindermans, and Q. V. Le, "Don't Decay the Learning Rate, Increase the Batch Size", *CoRR*, vol. abs/1711.00489, 2017. arXiv: 1711.00489. [Online]. Available: http://arxiv.org/abs/1711.00489.

[76] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks", *CoRR*, vol. abs/1404.5997, 2014. arXiv: 1404.5997. [Online]. Available: http://arxiv.org/abs/1404.5997.

[77] S. Jastrzebski, Z. Kenton, D. Arpit, N. Ballas, A. Fischer, Y. Bengio, and A. J. Storkey, "Three Factors Influencing Minima in SGD", *CoRR*, vol. abs/1711.04623, 2017. arXiv: 1711.04623. [Online]. Available: http://arxiv.org/abs/1711.04623.

[78] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", *CoRR*, vol. abs/1706.02677, 2017. arXiv: 1706.02677. [Online]. Available: http://arxiv.org/abs/1706.02677.

[79] S. L. Smith and V. Quoc, "A Bayesian Perspective on Generalization and Stochastic Gradient Descent", in *Proceedings of Second workshop on Bayesian Deep Learning (NIPS 2017)*, 2017.

[80] E. Hoffer, I. Hubara, and D. Soudry, "Train longer, generalize better: closing the generalization gap in large batch training of neural networks", *ArXiv e-prints*, arXiv: 1705.08741.

[81] D. Masters and C. Luschi, "Revisiting Small Batch Training for Deep Neural Networks", *CoRR*, vol. abs/1804.07612, 2018. arXiv: 1804.07612. [Online]. Available: http://arxiv.org/abs/1804.07612.

[82] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima", *CoRR*, vol. abs/1609.04836, 2016. arXiv: 1609.04836. [Online]. Available: http://arxiv.org/abs/1609.04836.

[83] D. Rolnick, A. Veit, S. J. Belongie, and N. Shavit, "Deep Learning is Robust to Massive Label Noise", *CoRR*, vol. abs/1705.10694, 2017. arXiv: 1705.10694. [Online]. Available: http://arxiv.org/abs/1705.10694.

[84] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient Coding", *ArXiv e-prints*, arXiv: 1612.03301.

[85] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of caffe, neon, theano, and torch for deep learning", *CoRR*, vol. abs/1511.06435, 2015. arXiv: 1511.06435. [Online]. Available: http://arxiv.org/abs/1511.06435.

[86] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch", 2017.

[87] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding", in *Proceedings of the 22nd ACM international conference on Multimedia*, ACM, 2014, pp. 675–678.

[88] D. Team, "Deeplearning4j: Open-source Distributed Deep Learning for the JVM", *Apache Software Foundation License*, vol. 2, 2016.

[89] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "TensorFlow: A System for Large-Scale Machine Learning.", in *OSDI*, vol. 16, 2016, pp. 265–283.

[90] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems", *ArXiv e-prints*, arXiv: 1603.04467.

[91] C. I.-D. Joeri R. Hermans, *Distributed Keras: Distributed Deep Learning with Apache Spark and Keras*, https://github.com/JoeriHermans/dist-keras/, 2016.

[92] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation", *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[93] E. Jones, T. Oliphant, and P. Peterson, "{SciPy}: Open Source Scientific Tools for {Python}", 2014.

[94] B. Recht and C. Re, "Beneath the valley of the noncommutative arithmetic-geometric mean inequality: conjectures, case-studies, and consequences", *ArXiv e-prints*, Feb. 2012. arXiv: 1202.4184.