Accurate, elastic large-scale distributed training over transient resources
Bachelor’s Thesis Nr. 402b
Systems Group, Department of Computer Science, ETH Zurich

Accurate, elastic large-scale distributed training over transient resources

by
Joel André

Supervised by
Prof. Ana Klimovic, Foteini Strati

March 2022–August 2022
Abstract

Deep Learning models are becoming larger and require training on vast amounts of data in order to reach convergence. Therefore, deep learning jobs often run on distributed clusters. As demand for resources increases and multi-tenant clusters become the norm, it is desirable to schedule jobs dynamically. As a result, training jobs should be elastic, that is, be able to train with variable amounts of resources. Elastic training reduces waiting time to start, increases efficiency, and leads to higher resource utilization. Furthermore, elastic jobs can leverage Transient Virtual Machines whose availability changes dynamically, to decrease cost dramatically. Different approaches have been proposed in the literature to support training with dynamic resource availability. We focus on the approach of scaling the batch size according to the number of available resources and scaling the learning rate accordingly. We explore the effects of elasticity on ML model convergence by empirically evaluating various learning rate scaling rules under frequent resource changes. We base our evaluations on resource traces gathered from real transient Virtual Machines in the Google Cloud.
Acknowledgements

First of all, I would like to thank my supervisors Prof. Ana Klimovic and Foteini Strati for completely integrating me into the research group. It was an amazing experience!

Furthermore I am grateful to Foteini Strati for always being there throughout the Thesis. I can not imagine a more supportive supervisor. Thank you!

Finally I thank my friends Yves and Fabian for reading my thesis and their great feedback.
Contents

List of Figures 3
List of Tables 4
1 Introduction 5
2 Background 7
  2.1 Machine Learning Overview .............................. 7
  2.1.1 Deep Neural Networks ................................. 8
  2.1.2 Gradient Descent .................................. 8
  2.1.3 ImageNet, VGG and ResNet .......................... 9
  2.2 Distributed Training ................................. 10
    2.2.1 Accumulation .................................. 11
  2.3 Transient Virtual Machines ......................... 12
3 Related Work ................................................. 13
  3.1 Training with large batch sizes ................. 13
  3.2 Elasticity in Machine Learning ............. 14
4 Dynamic Synchronous Distributed Gradient Descent 16
  4.1 Reacting to Elasticity ............................... 16
  4.2 Learning Rate Scaling Rules ...................... 18
    4.2.1 Identity Rule ................................ 18
    4.2.2 Linear Rule .................................. 19
    4.2.3 Square Root Rule .............................. 19
    4.2.4 AdaScale Rule ................................. 20
5 Simulating Elasticity ........................................ 21
  5.1 System Overview ..................................... 21
  5.2 Implementation Challenges ...................... 22
    5.2.1 Distributing Sub-Batches ..................... 22
    5.2.2 Connecting multiple TPUs ..................... 23
### 6 Evaluation

6.1 Batch Size Traces ........................................ 24
6.2 Results for ResNet50 ....................................... 26
   6.2.1 Discussion ........................................ 28
6.3 Results for VGG-16 ......................................... 28
   6.3.1 Discussion ......................................... 29

### 7 Conclusion

### 8 Appendix

8.1 Supplementary Convergence Figures ......................... 33
8.2 Insights on the momentum term ........................... 35
8.3 Insights on the learning rate for VGG ....................... 36

Bibliography ............................................. 37
List of Figures

2.1 Deep Neural Network Architecture with four inputs. 8
2.2 Neuron Input Transformation 8
2.3 Synchronous Distributed Gradient Descent with four workers 11
4.1 Node failure in SDGD 17
5.1 Possible sub-batch assignments across 8 TPU cores. Dots represent sub-batches. The total batch size is scaled by 1.3x between steps $t$ and $t + 1$. 23
6.1 Batch size over epochs of ImageNet for the evaluated traces 25
6.2 Training set accuracy convergence of ResNet50 using different scaling rules for each trace 27
6.3 Training set accuracy convergence of VGG-16 using different scaling rules for the GPU and TPU 2 Traces 29
8.1 Learning Rate Schedules for ResNet50 using different scaling rules for each trace 33
8.2 Validation Accuracy Convergence for ResNet50 using different scaling rules for each trace 34
8.3 Learning Rate Schedules for VGG-16 using different scaling rules for each trace 34
8.4 Validation Accuracy Convergence for VGG-16 using different scaling rules for each trace 35
List of Tables

6.1 Final Top-1 (validation) accuracy of ResNet50 for all combinations of traces and scaling rules. Standard deviation displayed based on three runs. .............................................. 27
6.2 Final Top-1 (validation) accuracy of VGG-16 for combinations of traces and scaling rules. Standard deviation displayed based on three runs. .............................................. 29
Chapter 1

Introduction

Deep Learning has revolutionized numerous fields, from Computer Vision to Natural Language Processing. It has become so ubiquitous that it is hard to imagine any industry unaffected by it.

Training the models empowering these applications is generally computationallly expensive. To speed up training, the workload is partitioned into a large number of machines, a technique known as distributed training. Users can easily rent these machines from a Cloud computing provider. The Cloud’s ease of use, scalability, and resource efficiency make it a perfect match for Machine Learning (ML). However, although the Cloud’s pay-as-you-go pricing model dramatically reduces costs for most users, ML training can still be costly. For example, training a BERT-like Natural Language Processing model can cost up to $1.6 Million [36]. As the number of model parameters and the amount of training data grows, these costs are bound to increase. Furthermore, training jobs usually employ an all-or-nothing scheduling policy, which requires all resources to become available before the job gets scheduled. This means that jobs can experience long waiting times to start, and some resources remain idle and underutilized waiting for the whole cluster to become available [33].

All of these issues can be alleviated if jobs can adapt to resources changing dynamically. Waiting time shortens as jobs can launch before all resources are available. Relative cost is reduced as this leads to higher resource utilization. Moreover, jobs can take advantage of Transient Virtual Machines (VM), available in AWS[1], GCP[5], and Azure[2]. These machines are up to 90% cheaper, but their availability depends on demand, i.e., they can get preempted (terminated) at any time. In practice, that means that the number of VMs may vary during a distributed job’s lifetime. We call such an environment elastic.

A particular challenge of training deep learning models in elastic environments is handling the batch size hyperparameter, as it is coupled to the amount of resources. In recent years, a couple of elastic training systems
have been proposed [33][14][16][8], handling the batch size in different ways. Some of these systems maintain the same hyperparameters [16] while others [33] adapt the batch size hyperparameter when resources change. We argue that the second approach is potentially more efficient as maintaining the same batch size can harm performance. Yet it is rarely used in practice due to the lack of training convergence guarantees. Hence, we find that there is a gap in research about the effects of elasticity in ML model convergence in this case.

This work empirically explores how typical Deep Learning workloads perform in an elastic, distributed resource environment when we try to train as efficiently as possible. The standard training algorithm we evaluate is *Distributed Synchronous Gradient Descent* (DSGD). Prior works typically only consider a single change in resources during the lifetime of a job [29][24]. We aim to empirically evaluate convergence of models under frequent resource changes. Furthermore, we evaluate real traces of transient VM availability gathered in the Google Cloud.

When training as described above, a change in resources leads to a change in the batch size hyperparameter. As the batch size is tightly coupled with the learning rate, we need to adapt the learning rate dynamically. Different learning rate scaling rules have been proposed in the literature[27][21][24]. We empirically evaluate the most popular ones and compare them. For our experiments, we train ResNet50[22] and VGG-16[38] on the ImageNet[18] dataset as these are two well studied Deep Learning setups. Overall our contributions are:

- We evaluate how different learning rate scaling rules perform in the DSGD paradigm when resources frequently change during training.
- We evaluate how different learning rate scaling rules perform on real transient VM availability traces.
- We built a PyTorch-based framework that makes it easy to adopt and evaluate different scaling rules, allowing future experimentation.

The thesis is organized as follows. In Chapter II, we explain the required background knowledge. In Chapter III, we discuss how this thesis relates to prior work. In Chapter IV, we review the effects of elasticity on Gradient Descent using the concepts explained in Chapter II. Then we present a few popular learning rate scaling rules from the literature. Chapter V discusses the technical challenges encountered when evaluating different learning rules and the implementation of our evaluation framework. In Chapter VI we present our results and comment on interesting aspects. Finally, we discuss our findings in Chapter VII. Additional result visualizations and miscellaneous insights can be found in the appendix.
Chapter 2

Background

This part explains the existing concepts that the thesis builds on. First, we will give an overview of the field of Machine Learning, emphasizing the techniques we use. Following that, we will address how Machine Learning workloads are run in distributed environments. In the end, we review how Transient Virtual Machines work in practice.

2.1 Machine Learning Overview

Over the last decade, few research fields have had as large an impact as Machine Learning (ML). Machine Learning has enabled applications ranging from recognizing faces, detecting fraudulent transactions[19], automatic language translation[41], beating world Go champions[37], to self-driving cars[26]. ML lets us solve tasks even when they are too difficult to solve by a human-written program. Instead, Machine Learning algorithms statistically find patterns in data to automatically find solutions. The building blocks of ML algorithms typically consist of some function whose parameters need to be determined, some objective we want to maximize, and a feedback mechanism letting the function improve. Most ML algorithms can be classified into one of the following three categories:

- **Supervised Learning**: This is the most common technique. The goal is to learn a function $f : X \rightarrow Y$ given many sample data points $(x, y) \in X \times Y$. Applications can be categorized as classification or regression. Classification models learn to match inputs to a discrete label, while regression tasks require the model to output a continuous estimate.

- **Unsupervised Learning**: Unlike in supervised learning, data points do not have a label. Typical tasks are clustering, i.e. finding good partitions of the data, or density estimation.
- **Reinforcement Learning**: Agents learn to take actions in order to reach objectives in an interactive game-like environment.

### 2.1.1 Deep Neural Networks

Deep Neural Networks (DNN) are at the heart of applications in all three domains. They make the function part of the described ML building blocks. They map input vectors \( x \) to output vectors \( y \) given parameters \( w \), called weights. DNNs are structured into layers of neurons, see Fig. 2.1.

![Deep Neural Network Architecture with four inputs.](image)

Each layer transforms its inputs and acts as input to the next layer. Given inputs \( x_0, \ldots, x_n \), weights \( w_0, \ldots, w_n \), and a bias term \( b \), each neuron computes \( f(\sum_{i=0}^{n-1} w_i x_i + b) \). Where \( n \) is the size of previous layer and \( f \) is a fixed non-linear function called *activation function*. See Fig. 2.2 for graphical intuition.

![Neuron Input Transformation](image)

The success of neural networks is due to their capacity to learn complex input-output relationships. In fact, neural networks can theoretically approximate any function under certain conditions [20]. For the rest of this thesis, it is enough to consider DNNs as differentiable parametrized functions.

### 2.1.2 Gradient Descent

Gradient Descent is an iterative procedure to find parameters minimizing a function. When we train deep neural networks, that function is called the
loss: It measures how close we are to our goal. The goal of training is to find optimal weights minimizing the loss.

Let the $D$ be our dataset of samples, and let $w$ be the weights of a network. The loss can be written:

$$L(w) = \frac{1}{|D|} \sum_{x \in D} l(x, w) \quad (2.1)$$

where $l(x, w)$ is the loss computed on a single sample $x$ with weights $w$. Note that in the case of supervised learning, this notation implies that $D$ consists of (input, label) pairs. Gradient Descent consists of iteratively computing the gradients of the loss in respect to the weights and updating the weights in the negative direction. That is:

$$w_{t+1} = w_t - \eta \nabla L(w_t) = w_t - \eta \frac{1}{|D|} \sum_{x \in D} \nabla l(x, w_t) \quad (2.2)$$

where the learning rate $\eta$ governs how large the step is. The total gradient is the average of the per sample gradients. Note that intuitively the gradients point towards where the function increase is steepest, so moving in the negative direction decreases the loss.

The above algorithm passes over the entire dataset at each iteration. Other variants of gradient descent trade off step quality for more speed:

- **Stochastic Gradient Descent** performs the weight update using only the gradients of a single sample at each iteration.

- **Minibatch Gradient Descent** performs the weight update using the average gradients of a batch of samples at each iteration. The number of samples in a batch is called the *minibatch size*. The computation of sample gradients in a batch can be parallelized.

Minibatch Gradient Descent achieves stable convergence and can be accelerated through parallelization. This makes it widely used in practice. The larger the batch size, the fewer iterations are required to process the entire dataset. Hence the batch size is typically set to the number of samples hardware can process in parallel. Yet, note that there is a limit to speedups with always larger batch sizes, as the number of steps cannot infinitely decrease [35].

### 2.1.3 ImageNet, VGG and ResNet

ImageNet[18] is an extensive database of images designed for benchmarking image recognition models. The ImageNet1k dataset consists of $\sim 1.28$ million images partitioned into 1000 categories, ranging from dog breeds to everyday objects. Starting 2010, the ImageNet project would run a yearly
contest, letting models compete over classifying the images as accurately as possible. VGG\[38\] and ResNet\[22\] are both deep neural network architectures for image recognition. By today’s standards VGG is a very generic deep model design. ResNet is a more sophisticated design with so called residual connections that span over multiple layers.

ResNet won the 2015 ImageNet contest by a significant margin. ResNets design principles have influenced many subsequent model architectures. So much, that the paper introducing ResNet has become the most cited Neural Networks paper of the 21st century [9]. ResNet50 is one of the variants proposed in the paper. The convergence behavior of ResNet50 on ImageNet is very well studied, making it perfect for evaluating new ideas.

2.2 Distributed Training

As datasets and models grow larger, training on multiple nodes has become necessary in many cases [17]. Larger datasets require more iterations to be processed, while larger models may not fit into the memory of a single machine. Depending on the issue, two types of parallelism exist:

- **Model/Pipeline Parallelism**: The model is split across multiple machines. This is mainly used when the model does not fit into a single machine.

- **Data Parallelism**: The data is split across multiple machines, while the model is replicated across machines. This is used in order to speed up training when datasets are large.

The two techniques are orthogonal to each other and can be used independently or together. We interchangeably refer to the machines working together as workers or nodes. Depending on how workers communicate to update the model, the data-parallel approach can be further divided into two categories:

- **Synchronous Distributed Gradient Descent**: Each worker initially starts off with the same model to compute the gradients of a minibatch. Note that as data is distributed across nodes, each node uses different data to compute different gradients. Before updating the model, workers average their gradients. As gradients are identical after averaging, workers end up with the same model after updating it, see Fig. 4.1.

- **Asynchronous Distributed Gradient Descent**: The setup is as with the synchronous variant. However, workers do not wait for each other to average their gradients in order to update the model. In the most extreme scenario, one central node keeps track of the model state,
and each worker updates it independently at each iteration, fetching
the new model afterward. More constrained versions exist. The moti-
vation for this approach is to get rid of the previous synchronization
bottleneck. Unfortunately, model convergence often degrades with this
approach, and few training guarantees exist.

![Diagram of Synchronous Distributed Gradient Descent](image)

Figure 2.3: Synchronous Distributed Gradient Descent with four workers

Note that Synchronous Distributed Gradient Descent (SDGD) does not
change the training semantics. That is, training using SDGD with \(k\) workers
of batch size \(n\) is equivalent to training on a single worker with batch size
\(k \cdot n\) (ignoring batch-normalization layers). From now on we refer to
the number of samples processed in a single forward pass of a single worker as
the \textit{sub-batchsize}, and to the total mini-batch-size as \textit{batch size}.

### 2.2.1 Accumulation

Research labs typically have access to powerful resources, allowing them
to train with large batch sizes. In order to reproduce their results using
a smaller batch size, a technique called \textit{Accumulation} can be used. The funda-
mental idea is to perform the work usually done in parallel sequentially.

Remember that the model update is determined by the average of the
per-sample loss gradients. This total average can also be obtained by av-
eraging the averages of batch subsets, where each subset gradient average
is computed sequentially. Formally, an iteration of batch size \(k \cdot n\) may be
performed by summing (accumulating) the gradients of \(k\) batches of size \(n\):

\[
w_{t+1} = w_t - \eta \frac{1}{k \cdot n} \sum_{i=1}^{k \cdot n} \nabla l(x_i, w_t) = w_t - \eta \frac{1}{k} \sum_{j=0}^{k-1} \left[ \frac{1}{n} \sum_{i=1}^{n} \nabla l(x_{j \cdot n + i}, w_t) \right]
\]

(2.3)

Another reason accumulation is used in practice is that ResNet and many
other models use a layer called \textit{Batch Normalization}. On a high level, Batch
Normalization improves convergence by normalizing inputs between layers
across the batch. However, an issue is that through batch normalization
the loss function changes slightly when we change the batch size. Hence in practice, many training regimes choose to be on the safe side by accumulating small batches to form a larger total batch, even though hardware would allow increasing the batch size.

2.3 Transient Virtual Machines

ML training takes a significant amount of time, even with high-end accelerators. This can make training modern ML models to convergence very expensive. One way to decrease the cost is to use transient instances.

Transient instances are virtual machines with a lower price tag than regular VMs allocated from a Cloud Provider’s spare capacity. Over time different cloud providers have used different names such as Low Priority VMs, Preemptible VMs or Spot instances to refer to this type of machines. The general idea has always been that Cloud Providers sell off their spare capacity at a reduced cost to maximize utilization. As spare capacity is variable, transient instances are not always available and may be reclaimed by providers. When an instance is reclaimed (preempted), it is either shut down or deleted. The exact details differ between providers:

- **Google Cloud** [5] : GCP Spot VMs are sold at 60-91% discounts. Users are notified 30s before preemption.

- **Amazon Web Services** [1] : EC2 Spot Instances are up to 90% cheaper. The price depends on capacity and demand. Users can specify a maximal price, and machines are preempted if the price exceeds the maximal price, or no more capacity is available. Users are notified 30s before preemption.

- **Microsoft Azure** [2] : Azure Spot Virtual Machines are up to 90% cheaper. As with AWS, a maximal price can be set. Users are notified 2 minutes before preemption.
Chapter 3

Related Work

This chapter reviews prior work related to our goals stated in chapter I. We first discuss papers that present techniques to train with increasingly larger batch sizes. Then we explore works focusing on making machine learning workloads elastic.

3.1 Training with large batch sizes

Driven by the potential training speedup, much research has gone into training with large batch sizes. The challenge is usually to keep the same final model accuracy as with smaller batch sizes [23].

Krizhevsky[27] explores how to parallelize convolutional neural networks efficiently. On a simple convolutional network, he reaches speedups of up to $\sim 6x$ without significant model quality degradation by scaling the learning rate by up to 8x. As the batch size is increased, the learning rate is scaled linearly. However, Krizhevsky also suggests scaling the batch size in proportion to the square root of the batch size in order to keep update variance constant.

Goyal et al.[21] use synchronous distributed gradient descent to train ResNet50 on ImageNet in 1 hour by using 256 GPUs. Even with a total batch size of 8192 samples, they report no significant model accuracy degradation. Like Krizhevsky, they also scale the learning rate linearly, and argue that this is optimal when gradients change little. We will delve into the reasoning behind this algorithm and others in Chapter 4. On top of this, they introduce learning rate *warmup* during the first five epochs, since gradients are changing quickly in the beginning. That is, they propose linearly scaling up the learning rate from a low initial value. After warmup, the learning rate is decreased by 10x at fixed epochs to fine-tune the model.

You et al.[40] propose Layer-wise Adaptive Rate Scaling (LARS). Their training procedure uses different learning rates for different layers depending on the weights’ and the gradients’ norms. The goal being to keep the scale
of the weight updates constant relatively to the scale of the weights. Using
LARS, they train ResNet50 on ImageNet with a total batch size of 32k at
baseline model quality. Johnson, Agrawal et al.[24] achieve the same result
with ResNet50 using a training method they call AdaScale. AdaScale dy-
namically adapts the learning rate to gradient variance. However, compared
to LARS, the learning rate is identical across layers. A nice property of
their algorithm is that a sort of warmup emerges naturally, without explic-
itly programming it. Furthermore, AdaScale also performs well in various
tasks, from speech recognition and object detection to machine translation.
On top of that, they also find that suddenly halving or doubling the batch
size for ResNet50 does not impact final accuracy.

Most of the mentioned works evaluate their derived learning rate scaling
rule on different but static amounts of resources. The only exception is
AdaScale, which considers one change of resources throughout the whole
training. Our goal is to evaluate how these rules perform in environments
with dynamic resource availability, since we believe this will be an important
setup for future training workloads.

3.2 Elasticity in Machine Learning

For a long time, machine learning libraries like Pytorch[32] and Tensorflow[15]
have considered resources to be static during training. However, elastic ex-
tensions have become increasingly popular. Horovod[34] is a framework on
top of Tensorflow for distributed data-parallel training. It makes use of ef-
cient inter-GPU communication to scale to large clusters. It further provides
a very accessible API for distributed training, only requiring a few lines of
Horovod to execute with a dynamic amount of resources at runtime. Work-
ers regularly commit a model state and roll back to the last commit upon a
worker failure. Callbacks can be added to adapt the learning rate depending
on the size of the cluster. Horovod is also available for Pytorch, but Pytorch
v1.9 comes packaged with Torch Distributed Elastic[14], a module for elastic
distributed data-parallel training. Users can specify a maximal and minimal
number of nodes, and training is not interrupted as long as the cluster size
is in that range. Upon a worker failure, all workers are restarted from the
last checkpoint, assigned a new rank, and informed of the new world size.
The checkpoint frequency needs to be manually set. Administrative work is
performed by one or more agents, each being responsible for a set of workers.
Note that in all of the presented works so far, users have to manually decide
how to adapt the learning rate upon a resource change. This is not always
easy to decide, and not adequately studied. We believe our work helps to-
wards evaluating how the most common learning rate scaling rules perform
as resources change.
More recently, Wagenländer et al.[39] focus on designing machine learning frameworks for transient instances. They build Spotnik, an adaptive implementation of the all-reduce operation to synchronize gradients. Spotnik efficiently detects and broadcasts failures using the allreduce operation and copes with node failures without restarting the workers. The problem of partial updates is solved by synchronizing the weights in a temporary structure and only applying the update if synchronization was successful. Their system dramatically decreases the time it takes to continue training following a node failure. Qiao et al.[33] explore job elasticity from a completely different angle. They focus on how cluster schedulers can dynamically allocate the right amount of resources to training jobs to optimize performance. By monitoring jobs at runtime, they try to determine how adding or removing resources would affect throughput and efficiency. They report that their scheduler, Pollux, reduces average job completion time by 37-50%.

Finally, Lin et al.[29] focus on the impact of momentum in an elastic training environment. As in this work, they explore how elasticity affects synchronous distributed gradient descent. As we discuss in chapter 4, they reason that the batch size may change dynamically in an elastic environment. They then argue that when the batch size increases, momentum noise negatively affects convergence. Their solution is to continuously increase the actual learning rate over a fixed number of steps when the batch size increases, until the target learning rate is reached. The target learning rate scales linearly. Lin et al. evaluate their scaling rule by increasing the scale by 12-16x at a fixed step during training. They evaluate different image recognition networks, object detection, and semantic segmentation. With ResNet50 on ImageNet, they report a 0.08-0.55% final model accuracy improvement compared to directly linearly scaling the learning rate.
Chapter 4

Dynamic Synchronous Distributed Gradient Descent

In this chapter, we discuss the impact of an elastic environment on synchronous distributed gradient descent and explore possible options to react to workers failing or joining. We argue that keeping the batch size proportional to the number of workers can result in significant speedups and explain why the learning rate needs to be dynamically adjusted in that scenario. We then present different possible learning rate scaling rules and their motivations.

4.1 Reacting to Elasticity

Assume we train a model using synchronous distributed gradient descent with \( k \) workers, each computing the gradients of a sub-batch of size \( B \) at every step. At each step, they synchronize to average their gradients in order to perform the same weight update. The global batch size is \( k \cdot B \).

Elasticity requires us to handle two types of events. Each event has a different impact on the training:

- **Workers failing**: When a worker gets preempted/fails during a step, it cannot contribute the sub-batch gradients it was computing. If an adaptive implementation of allreduce as in [39] is used, the other workers are unaffected; see Fig 4.1. Even though the preempted worker’s computation is lost, the training invariant that each sample is seen precisely once per epoch can still hold if we dynamically reassign that sub-batch to another worker.

- **Workers joining**: When a worker joins the cluster, it can participate in the gradient computation as soon as the next step begins. The only overhead is that the model weights must be shared with the new worker. This can efficiently be parallelized [31].
In either case, the critical question is: What to do with the batch size when the number of workers changes?

![Diagram of gradient flow and data distribution]

Figure 4.1: Node failure in SDGD

If we keep the sub-batch size identical, the total batch size changes, e.g., if one node fails, the new global batch size is \((k-1) \cdot B\). On the other hand, if we change the amount of work per worker, our system may not be as efficient anymore. What follows is a more detailed comparison of our options.

1. **Keep the global batch size constant.**

   **When a worker fails,** we must assign more samples to other workers to keep the global batch size. This is the approach *Varuna*\cite{16} follows. We can either reassign the worker’s \(B\) samples to a small subset of other workers or uniformly to all workers. In the former scenario, the workers performing extra work will likely become stragglers. For example, if we assume the throughput of all workers to be identical and all the work is reassigned to a single worker, the step computation time may double as all the other workers wait for the extra work to be done. In practice, a system might be able to reduce the overhead by picking the fastest worker for the extra work.

   If we evenly distribute the samples to all workers instead, computational efficiency might decrease as certain hardware accelerators like TPUs prefer well-formed sub-batch sizes \cite{3}. Furthermore, if batch normalization is used, this may impact convergence.

   **When a worker joins,** we need to dilute the per-node sub-batch size to keep the global batch size. This increases the communication to computation ratio. Moreover, as the sub-batch size shrinks, computational efficiency decreases as soon as the hardware is no longer fully utilized.

2. **Scale the global batch size accordingly.**

   The global batch size automatically scales with the number of workers when we do not change anything about the sub-batch size. When a new
worker joins, for example, it will contribute an additional $B$ sample gradi-
ents to the next update so that the new global batch size will be $(k + 1) \cdot B$. This is
the default behaviour of popular elastic frameworks like PyTorch Elastic[14] and Horovod Elastic[8].
The issue with this approach is that batch size changes may impact training convergence. Popular con-
vergence bounds on mini-batch SGD no longer hold [25][24][29].

As explained before, this thesis focuses on the second approach. According to our internal experiments, scaling glob-
al batch size dynamically is 20-25% faster than keeping it constant. We believe that the computa-
tional efficiency and lower complexity of this approach, together with the other
issues of approach one explained above, make it worthwhile to explore.

4.2 Learning Rate Scaling Rules

The batch size hyperparameter of SGD is strongly coupled to the learning
rate [35]. As models are trained with increasingly large batch sizes, the learn-
ing rate is also scaled [35]. Hence if an elastic training environment dynami-
cally affects the batch size, we should also consider dynamically adapting the
learning rate. Lin et al. [29] only focus on dynamically linearly changing the
learning rate, but various scaling rules are theoretically possible. The goal
of this section is to present an overview of various scaling rules proposed in
the literature before we proceed to empirically evaluate a few in chapter 6.
Note that the following scaling rules have originally been proposed for static
use. They focus on how we should scale the learning rate if we train with
a different but constant new batch size. However, this makes them no less
relevant in our dynamic context.

4.2.1 Identity Rule

The primary motivation behind increasing the batch size is to reduce training
time since larger batch sizes let us perform more work in parallel.

The identity rule suggests keeping the learning rate constant when the
batch size increases. This rule is rarely used in practice as it fails to reduce the
number of steps required to converge [35]. In other words, training does
not get faster. The reason is that the algorithm can ‘cover less ground’ as
the number of steps per epoch decreases without increasing the steps’ scale.
Hence even though we approach the target minima, we do not reach it in time
if we keep the same number of epochs. Hoffer et al.[23] show that increasing
the number of steps can help to overcome this. However, this defeats the
motivation for larger batch sizes since it increases training time again.
4.2.2 Linear Rule

The linear learning rate prescribes scaling the learning rate by \( k \) when the batch size changes by factor \( k \). On a superficial level, the increase can be explained by the fact that the average gradient from a larger batch size better approximates the global gradient \( \nabla L(w) \). Hence we might be able to take a larger step. Goyal et al.[21] explain a deeper intuition as follows. Compare the updated weights after \( k \) steps of batch size \( n \) (eq. 4.1) to the weights after one step with batch size \( kn \) (eq. 4.2):

\[
\dot{w}_{t+1} = w_t - \frac{\eta}{kn} \sum_{i=0}^{kn-1} \nabla l(x_i, w_t)
\]

where \( \eta \) and \( \hat{\eta} \) are the respective learning rates and \( w_t \) represents the weights at step \( t \). The insight is that under the assumption that \( \nabla l(x, w_{t+j}) \approx \nabla l(x, w_t) \) for all \( j < k \), the final weights are identical. If the gradients at the \( k-1 \) intermediate steps are identical, taking one large step is equivalent to \( k \) small steps. This is a strong assumption, but the empirical success of linear scaling might suggest that it often holds in practice for large batch sizes. Lin et al.[29] review further theoretical properties of linear scaling.

The linear learning rate is undoubtedly the most popular scaling rule. Goyal et al.[21] train ResNet50 on ImageNet in under 1 hour with batch-size 8192 and a linearly scaled learning rate compared to a baseline with batch size 256. Johnson et al.[24], and Lin et al.[29] both evaluate how linear scaling affects convergence when batch size is scaled at a fixed point in training and observe a slight accuracy degradation.

4.2.3 Square Root Rule

Krizhevsky[27] proposes the Square Root Scaling Rule: When the batch size scales by \( k \), increase the learning rate by \( \sqrt{k} \). He observes that this conserves the scale of weight update variance. Write a weight update as follows

\[
w_{t+1} = w_t - \eta \frac{1}{n} \sum_{i=1}^{n} \nabla l(x_i, w_t) := w_t + \Delta w
\]

where \( n \) is the batch size. Assuming \( \nabla l(x_i, w_t) \) is i.i.d with variance \( \theta^2 \), the variance of the update is

\[
\text{Var}[\Delta w] = \frac{n \theta^2}{n^2} \text{Var} \left[ \sum_{i=1}^{n} \nabla l(x_i, w_t) \right] = \frac{n \theta^2}{n} = \theta^2
\]
. So to keep Var[Δw] constant, the learning rate η must scale with the root of n. However, in practice Krizhevsky notices that when scaling AlexNet[28] to a batch size of 1024, the linear learning rate rule unexpectedly performs better.

4.2.4 AdaScale Rule

Johnson, Agrawal et al.[24] propose an algorithm they call AdaScale to train with very large batch sizes. Let lr(t) be the learning rate used at step t in a baseline training procedure with batch size n. When the batch size is scaled to kn, AdaScale suggests using the learning rate ηt defined as follows:

\[ η_t = r_t \cdot lr(\lfloor \tau_t \rfloor) \quad \text{with} \quad \tau_t = \sum_{t'=0}^{t-1} r_t' \]

The original learning rate is multiplied by the gain r_t ∈ [1, k]. Their idea is that step t performs the equivalent progress of r_t steps with the original learning rate. τ_t accumulates this progress and represents the scale-invariant-iteration. Training finishes when τ_t ≥ T, where T is the total number of iterations of the baseline procedure. Note that AdaScale may hence train for up to k times more epochs than originally set. Let \( \sigma^2_g(\mathbf{w}_t) \) and \( \mu^2_g(\mathbf{w}_t) \) be the variance and the squared norm of the gradients at iteration t respectively. The gain is defined

\[ r_t = \frac{\mathbb{E}_{\mathbf{w}_t} \left[ \sigma^2_g(\mathbf{w}_t) + \mu^2_g(\mathbf{w}_t) \right]}{\mathbb{E}_{\mathbf{w}_t} \left[ \frac{1}{k} \sigma^2_g(\mathbf{w}_t) + \mu^2_g(\mathbf{w}_t) \right]} \]

The intuition is that when the variance of the gradients approaches 0, we cannot expect to profit from increasing the batch size; hence the above formula evaluates to 1. However, as the variance increases, a large batch size helps to better approximate the true gradient \( \nabla L(w) \). The gain converges to k in this case. The authors evaluate AdaScale on Image Recognition, Speech Recognition, Machine Translation, and Object Detection and observe no model degradation even for very large batch sizes.
Chapter 5

Simulating Elasticity

In Chapter 4, we motivated why the batch size could change over time in an elastic environment and presented different rules to adjust the learning rate dynamically. In Chapter 6, we evaluate the rules using different elasticity scenarios. In order to reliably compare the rules, we must run them using the same dynamic batch size. To this end, we build a framework that lets us specify how the batch size should change over time and trains accordingly. In this chapter, we focus on its features and implementation.

We start by giving a high-level overview of our system. Then we present key implementation challenges.

5.1 System Overview

The system takes a learning rate scaling rule and a batch size trace as input. The trace specifies what batch size should be used for each training step. The system then trains the model as if the batch size dynamically changed due to nodes failing or joining as specified in the trace. Note that the system itself is run on normal virtual machines, i.e., not transient VMs. Furthermore, our system does not dynamically add or remove nodes from the cluster but instead controls the batch size through accumulation. For example, when the batch size should increase, the system accumulates more gradients before performing the weight update.

The trace can either be manually set or come from some measurement of available resources in the cloud. The input format is a CSV file with a 'step' and 'batch size' column. It is the user’s responsibility to ensure that the batch size is specified for enough steps. All our training scripts as well as the described system, are implemented in Pytorch[32]. The system is designed to run on Google TPUs[6] using Pytorch XLA[13]. Any Pytorch XLA training script can be modified to use our framework by adding our custom sampler class and wrapping the optimizer with one of our custom optimizers. Listings 5.1 and 5.2 show the required changes.
We refer to the code base for further details.

## 5.2 Implementation Challenges

### 5.2.1 Distributing Sub-Batches

The most interesting technical challenge is actively steering the batch size. What made this tricky is that TPUs are inherently distributed devices. A single TPU-VM consists of 8 separate cores, on which models must be trained with synchronous distributed gradient descent. Furthermore, TPUs record all operations in a computational graph to optimize performance. Changing the batch size in a TPU core causes graph recompilation, which takes a non-negligible amount of time.

Our solution is to split the batch size into sub-batches of a constant size $B$. In our experiments, $B = 32$ is used. The sub-batches are distributed to all cores. As the batch size changes, the number of sub-batches changes. So at each step, each core is dynamically assigned none, one or more sub-batches. The cores accumulate the gradients across sub-batches. When the gradients are averaged across cores, the result is scaled inversely to the total number of sub-batches. Figure 5.1 shows how sub-batches could be assigned at a specific step.

Note that this method has the nice property that no matter how the batch size changes, batch-normalization is computed using the same constant sub-batch size.

In our framework, the custom sampler and optimizer implement the policy as follows:

- **The Custom Sampler** is responsible for assigning the correct sub-batches to each core data loader. It further guarantees that each dataset sample is still seen exactly once during the epoch.

- **The Optimizer Wrapper** manages accumulation. The number of accumulated sub-batches at this step depends on the batch size according to the trace. When `optimizer.step()` is called, it determines whether gradients need to be synced across cores at this step or not.
Figure 5.1: Possible sub-batch assignments across 8 TPU cores. Dots represent sub-batches. The total batch size is scaled by 1.3x between steps $t$ and $t+1$.

For all intermediate accumulation steps, our optimizer wrapper only accumulates the gradients without updating the weights. Only after the last accumulated sub-batch of each step do cores synchronize.

### 5.2.2 Connecting multiple TPUs

The second challenge is scaling out the system to more than one TPU-VM. We only had access to 8-core TPU-VMs and not larger TPU-PODs. Our solution was to connect the VMs using the `torch.distributed`[10] communication package with the Gloo[4] backend. In order to synchronize gradients globally, we perform the following steps:

1. Average the gradients across cores in a single TPU-VM.
2. Load the gradients to CPU memory.
3. Average the gradients across machines with `torch.distributed`.
4. Broadcast the result to all cores.

Unfortunately, the overhead of our system was relatively high. Hence for most of the experiments performed in Chapter 6, we opted to perform different experiments concurrently with multiple VMs instead of running one experiment on multiple machines.
Chapter 6

Evaluation

This chapter empirically evaluates how the presented learning rate scaling rules perform if the batch size varies during training. We evaluate the scaling rules with batch size traces gathered from real spot instances in Google Cloud as well as with a self-designed trace. We choose to evaluate the scaling rules by pre-training the ResNet50 and VGG-16 models on the ImageNet dataset. We opt for these two model architectures since they are very generic and have both been widely studied in other works. First, we present the traces we use for evaluation. Then we show our results for the ResNet50 and VGG-16 architectures.

6.1 Batch Size Traces

Goyal et al.[21] train ResNet on ImageNet with a total batch size of 8192. Our traces are of a similar magnitude, as we deem this an appropriate scale for modern synchronous distributed gradient descent. We evaluate two types of traces: Traces gathered by measuring the availability of spot instances in the cloud and self-designed, 'artificial' traces. Since experiments are computationally expensive and time-intensive, we restrict ourselves to three 'real' traces and one artificial trace. As the availability of spot instances varies at all times, we can only evaluate a subset of possible traces. We chose the selected three real traces since we deem them interesting in terms of batch size dynamics and representative of our overall measurements. Figure 6.1 presents the traces.
Figure 6.1: Batch size over epochs of ImageNet for the evaluated traces
Trace (a) is obtained by gathering the availability of spot instances with a GPU accelerator. We created a Google GKE[7] node pool of preemptible A100 GPU’s. The cluster automatically attempts to request new VMs when others are preempted. Trace (b) and (c) are based on the availability of TPU-VM spot instances. We request 16 TPUv2-8 TPU-VMs in the us-central1-f zone, monitor the cluster, and attempt to increase the number of VMs at regular time intervals.

In both cases, we assume that each spot instance can efficiently handle a batch size of 128. That is, a batch size of 1024 in the traces indicates that we were able to obtain eight machines at this point in time. Note that the raw data we obtained specified the number of available machines at a point in time. In order to convert this to the batch size at a particular step we have to make assumptions on the performance of the instances. For simplicity, we assume that each step takes 0.6s independently of the batch size. In other words, as the batch size increases, we can perfectly scale performance. The 0.6s are based on the speeds we experience on both GPUs and TPUs. Depending on the number of steps, each trace covers around 200 to 275 minutes of cloud availability measurements.

Trace (d) is our custom self-designed trace that we evaluate to observe how the rules perform under high stress. We designed the trace so that the batch size changes at epoch boundaries. First, it trains the model with batch size 8192 for one epoch, then with batch size 4608 = (8192+1024)/2, followed by batch size 1024 and batch size 4608 again. The pattern is repeated until the end of training.

6.2 Results for ResNet50

We train ResNet50 according the training procedure described in [21]. In the following, we describe some key hyperparameters: The model is trained for 90 epochs and evaluated on the validation set after each epoch. The learning rate is linearly warmed up during the first five epochs. At epochs 30, 60, and 80, the learning rate is scaled by 1/10. The reference learning rate used for the learning rate scaling rules is 0.1 for batch size 256. The sub-batch size is always 32. As a baseline, we reproduce the result of [21] by training with batch size 8192 and a linearly scaled learning rate (8192/256 * 0.1 = 3.2). Goyal et al.[21] report a Top-1 of accuracy 73.27% for static batch size 8192.

All experiments are run with full precision on TPUv3-8 virtual machines using all eight cores. For a single run, the reported accuracy is the mean of the last three epochs’ validation accuracy. Each experiment is run three times with different seeds, and the reported numbers are the mean and standard deviation of the runs. For a fair comparison we also evaluate the root scaling rule and the adascale rule using the static 8192 batch size. The results are presented in table 6.1.
Table 6.1: Final Top-1 (validation) accuracy of ResNet50 for all combinations of traces and scaling rules. Standard deviation displayed based on three runs.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Linear Rule</th>
<th>Root Rule</th>
<th>AdaScale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Baseline</td>
<td>76.02 ± 0.11</td>
<td>73.70 ± 0.14</td>
<td>76.31 ± 0.18</td>
</tr>
<tr>
<td>GPU (a)</td>
<td>76.15 ± 0.12</td>
<td>74.38 ± 0.07</td>
<td>76.39 ± 0.23</td>
</tr>
<tr>
<td>TPU 1 (b)</td>
<td>76.04 ± 0.10</td>
<td>74.08 ± 0.03</td>
<td>76.29 ± 0.14</td>
</tr>
<tr>
<td>TPU 2 (c)</td>
<td>76.20 ± 0.10</td>
<td>74.84 ± 0.19</td>
<td>76.31 ± 0.03</td>
</tr>
<tr>
<td>Saw (d)</td>
<td>73.86 ± 0.43</td>
<td>74.80 ± 0.13</td>
<td>76.27 ± 0.07</td>
</tr>
</tbody>
</table>

In Figure 6.2, we further display the training accuracy convergence curves for each trace. The sudden increases are due to learning rate decay. Note that for AdaScale, learning rate decay often happens later since it depends on AdaScale’s scale-invariant-iteration progress as explained in chapter 4. The validation accuracy training curves can be found in Figure 8.2 of the appendix. Additionally, we visualize how the learning rate changes in Figure 8.1 of the appendix.

Figure 6.2: Training set accuracy convergence of ResNet50 using different scaling rules for each trace
6.2.1 Discussion

We investigated how frequent batch size changes impact training convergence using real traces. Our results show that for the linear scaling rule, model quality never degrades except for the extreme 'Saw' trace.

The root scaling rules performs worse on most traces, but so does its baseline. The learning rate computed by the root scaling rule is most likely too small for this setting. Interestingly the root scaling rule outperforms the linear scaling rule on the 'Saw' trace. This could be due to the fact that the root scaling rule scales the learning rate less aggressively, so the learning rate variance is considerably smaller.

The AdaScale optimizer outperforms both other rules on all traces. Accuracy does not degrade significantly even with the extreme 'Saw' trace. Yet when analyzing how AdaScale scales the learning rate (Fig. 8.1), we observe that the AdaScale learning rate only significantly differs from the linear scaled learning rate during the first 30 epochs. This implies that model degradation observed with the linear scaling rule on the 'Saw' trace stems predominantly from the variance in a batch size of the first 30 epochs.

Finally, we would like to note that the batch size changes are hardly visible on the Training Accuracy Converge plots of Fig 6.2, except for the 'Saw' trace.

6.3 Results for VGG-16

We opt to evaluate a modified version of VGG-16 that includes batch-normalization as defined in [12], since we faced difficulties when scaling the original VGG-16 network (see section 8.3). We train VGG-16 similarly to ResNet50, and the official Pytorch recipe [11]. We again establish a baseline using batch size 8192. We use the same training script as for ResNet50 but change the reference learning rate for both the linear scaling rule and the AdaScale rule. In our experiments, a reference learning rate of 0.1 for batch size 256 leads to exploding gradients when linearly scaled to batch size 8192. Hence, we change the reference learning so that the linear scaling rule baseline is equivalent to the root scaling rule baseline. That is, we use reference learning rate $0.1 \cdot \sqrt{32} = 0.566$ for batch size 8192 for the linear scaling rule. For example, if the batch size is halved to 4096, the linear rule will scale the learning rate to 0.283. The reference learning rate for AdaScale is changed identically.

Unlike with ResNet50, we choose to evaluate only the GPU and TPU 2 traces. All experiments are again run with full precision on TPUv3-8 virtual machines using all eight cores. For a single run, the reported accuracy is the mean of the last three epochs’ validation accuracy. Each experiment is run three times with different seeds, and the reported numbers are the mean and standard deviation of the runs. Results are presented in table 6.2.
Table 6.2: Final Top-1 (validation) accuracy of VGG-16 for combinations of traces and scaling rules. Standard deviation displayed based on three runs.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Linear Rule</th>
<th>Root Rule</th>
<th>AdaScale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Baseline</td>
<td>-</td>
<td>72.62 ± 0.14</td>
<td>72.37 ± 0.10</td>
</tr>
<tr>
<td>GPU</td>
<td>72.71 ± 0.11</td>
<td>73.05 ± 0.12</td>
<td>72.37 ± 0.03</td>
</tr>
<tr>
<td>TPU 2</td>
<td>72.63 ± 0.09</td>
<td>73.21 ± 0.14</td>
<td>72.58 ± 0.10</td>
</tr>
</tbody>
</table>

In Figure 6.3 we further display the training accuracy convergence curves for each trace. The validation accuracy training curves can be found in Figure 8.4 of the appendix. We visualize how the learning rate changes in Figure 8.3 of the appendix.

![Graphs showing training accuracy convergence for GPU and TPU 2 traces](image)

Figure 6.3: Training set accuracy convergence of VGG-16 using different scaling rules for the GPU and TPU 2 Traces

6.3.1 Discussion

We evaluated the VGG model to investigate how well the ResNet50 results generalize to other models. The linear learning rate scaling rule does not perform well with the VGG architecture. If we use the same reference learning rate as for the root scaling rule to scale the batch size, training accuracy diverges. Yet, even if we reduce the learning rate so that the linear and root rule baselines are equivalent, the linear learning rate performs worse on all traces. Now the learning rate is most likely too low.

The root learning rate, on the other hand, has a stable baseline, and model quality does not degrade on any of the traces. Like with ResNet50, we observe that the benefit of smaller batch sizes outweighs the potentially negative effect of batch size changes.

Surprisingly, AdaScale performs worse than the root scaling rule. Like the linear scaling rule, AdaScale training did not properly converge with the
standard learning rate. As for the linear scaling rule, the new learning rate is probably too small.

Overall, these results show that it is not the linear scaling rule per se that is better than the root scaling rule (as suggested by the ResNet50 results). Instead, the best learning rate scaling rules seem workload-dependent. The fact that linear scaling works better with ResNet50 and root scaling is better with VGG-16 matches the results in [35]. Shallue et al.[35] empirically investigate the optimal learning rate for different batch sizes and various models. Part of their results is that for ResNet50 the optimal learning rate closely follows the linear scaling rule and for VGG-11 the optimal learning rate is closer to the root scaling rule.

Nevertheless, our results confirm that with the right scaling rule, model convergence is barely influenced by batch size variation.
Chapter 7

Conclusion

This thesis explored how elasticity impacts model convergence if we dynamically adapt the batch size to the number of resources. First, we explored different ways to handle node failures in distributed synchronous gradient descent. Then we reasoned that changing the batch size could be more efficient when resources change. To scale the learning rate when the batch size increases, we reviewed different learning rate scaling rules proposed in other works. Furthermore, in order to compare the different learning rate scaling rules, we built a system enabling us to train with a specified batch size trace. We used this system to evaluate the impact of real traces on the popular computer vision models ResNet50 and VGG-16.

Our results show that realistic batch size changes have little impact on model convergence if the right learning rate scaling rule is used. However, we observe that the best scaling rule depends on the model. Our findings in this regard match with [35], who show that the optimal learning rate to batch size relationship differs between models. Furthermore, our empirical results confirm AdaScale’s ability to cope with a varying batch size. Johnson, Agrawal et al.,[24] only evaluate AdaScale with a single batch size change. We show that with ResNet50, it can train smoothly even under very frequent aggressive batch size changes.

Unlike [29], we do not observe model degradation due to inappropriate momentum scale upon a batch size change, so we chose not to evaluate their proposed fix. The reason why we do not observe model degradation is most likely due to our baseline for ResNet50 being a bit weaker. Nevertheless, we explore this aspect a bit further in the appendix section 8.2.

As it is, our results suggest that variable batch size training requires users to know what kind of scaling rule to use for their workload. This would make it difficult to be used in practice. Ideally, there would be an optimizer that automatically scales the learning rate well for as many settings as possible. AdaScale is a candidate, but it does not scale the learning rate well for VGG-16 in our experiments. We note that more experiments are needed to give
a more comprehensive picture of elasticity impact across various domains. We only evaluate two Image Recognition models on the same dataset, but it would be interesting to observe the effect on other Deep Learning workloads like Transformers and Object Detection. Our system can be used for these tasks as well, and new task-specific learning rate rules can be plugged-in to enable further experimentation.

Furthermore we would like to stress that we always assumed that no data points are lost throughout training. But this does not always hold, especially with TB-sized datasets [30]. Future work could explore how such scenarios affect training convergence when matched with a dynamic batch size.

Wrapping up, we think that elastic training is an essential part of the future of Deep Learning. Its improved resource utilization, reduced waiting time, and potentially significant cost savings, will be attractive for many applications. While we investigated elastic training with variable batch sizes, we are also very excited to see how systems that keep the batch size constant will improve in the future.
Chapter 8

Appendix

8.1 Supplementary Convergence Figures

Figure 8.1: Learning Rate Schedules for ResNet50 using different scaling rules for each trace
Figure 8.2: Validation Accuracy Convergence for ResNet50 using different scaling rules for each trace

Figure 8.3: Learning Rate Schedules for VGG-16 using different scaling rules for each trace
Figure 8.4: Validation Accuracy Convergence for VGG-16 using different scaling rules for each trace

8.2 Insights on the momentum term

Intuitively, the fact that the learning rate can suddenly increase by a significant factor when the batch size increases is already questionable. Especially if we think of our small steps as closing in on a good optimum. Lin et al. [29] show that momentum makes this worse and rigorously define the issue. They focus on how momentum affects SGD when the batch size changes. They observe that with momentum, the gradient variance does not immediately decrease when the batch size rises. Momentum can be written

$$ v_{t+1} = \mu v_t + \frac{1}{B} \sum_{i=1}^{B} \nabla l(w_t, x_i) \quad \text{and} \quad w_{t+1} = w_t - \eta v_{t+1} \quad (8.1) $$

where $\mu$ is the momentum, $B$ the batch size, and $\eta$ the learning rate. They find that the variance of $v_{t+1}$ does not immediately decrease a lot upon a batch size change, even though linear scaling immediately scales $\eta$. Hence the variance of $\eta v_{t+1}$ is temporarily very high.

Lin et al. propose linearly warming up the learning rate for a fixed number of steps $T$ when the batch size increases. They claim that $T = 8000$ is a good choice for their use cases. The drawback of this method is that it introduces a hyperparameter of unknown scale to tune.

Since our evaluations of the linear scaling rule (with ResNet50) on real traces already beat our baseline, we decided not to test this method. We did, however somewhat explore the effect of momentum compensation on the ‘Saw’ trace. Instead of linearly warming up the learning rate, we choose to introduce learning rate warmup of $\mu = 0.9$ since this method makes it easier to set the hyperparameter. With this method, linear scaling reaches 75.6% accuracy on the ‘Saw’ trace with ResNet50.

Note that all of these momentum compensation methods do not keep the weight update variance constant when used with linear scaling. As explained
in section 4.2.3 this would require using root scaling the learning rate. When
the batch size is scaled by \( k \), the variance of the weight update increases by
\( k \) with the linear learning rate scaling rule. However, what these methods
try to avoid is for the variance to exceed even that temporarily.

Finally, we would like to mention an interesting fix proposed in the
appendix of [29]. They suggest not scaling the learning rate and instead sum-
ing the gradients in place of averaging them; see equation 8.2. Unfortunately, they do not evaluate the proposed method.

\[
v_{t+1} = \mu v_t + \sum_{i=1}^{B} \nabla l(w_t, x_i) \quad \text{and} \quad w_{t+1} = w_t - \eta v_{t+1} \quad (8.2)
\]

At most points of training, the above method is equivalent to linear
scaling. Yet it decouples the accumulated momentum from the batch size.
In practice, this method leads to the scale of the update increasing more
smoothly upon a batch size change.

We implemented this approach in PyTorch and observed that a naive
implementation does not handle weight decay correctly. Weight decay must
be linearly increased relatively to the batch size in order for the above method
to be equivalent to linear learning rate scaling. Unfortunately, even though
we find this method very elegant, we could not empirically observe significant
gains on the ‘Saw’ trace.

8.3 Insights on the learning rate for VGG

The official PyTorch recipe[11] suggests that training VGG-16-BN with batch
size 256 and learning rate 0.1 reaches 73.36% accuracy. For the standard
VGG-16 a learning rate of 0.01 is reported to reach 71.592%. We performed
the following experiments to to get a static large batch size baseline:

- Train VGG-16 with batch size 8192 and learning rate 0.32 (linear
  scaled): \( \rightarrow \) Exploding Gradients.
- Train VGG-16-BN with batch size 8192 and learning rate 3.2:
  \( \rightarrow \) Exploding Gradients
- Train VGG-16-BN with batch size 8192 and learning rate 0.32:
  \( \rightarrow < 70\% \) Accuracy
- Train VGG-16-BN with batch size 8192 and 0.566 (root scaled):
  \( \rightarrow \approx 72.65\% \) Accuracy

We opted to use the root scaled learning rate for our baseline experiments.
But based on the convergence curves we believe that a learning rate between
0.566 and 3.2 most likely achieves even better results for VGG-16-BN.
Bibliography


[34] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.


