ARCADA

Large-scale Deep Learning by Distributed Training

Juha Nyholm

Master’s Thesis
Master's degree in Big Data Analytics
2019

MASTER'S THESIS	
Arcada	
Degree Programme:	Master's degree programme in Big Data Analytics
Identification number:	7248
Author:	Juha Nyholm
Title:	Large-scale Deep Learning by Distributed Training
Supervisor (Arcada):	Magnus Westerlund
Commissioned by:	CSC – IT CENTER FOR SCIENCE LTD.

Abstract:
This thesis is done as part of a service development task of distributed deep learning on the CSC provided infrastructure. The aim is to improve the readiness to provide a service for AI researchers who wish to scale out in deep learning and to benefit from the potential speedup gains of distributed deep learning. The algorithmic challenges in large-scale distributed deep learning involve hardware utilization and model quality related challenges, which must be addressed in order to truly benefit from scaling out. In this thesis, we experiment with the Horovod distributed training framework, which provides a means for more efficient resource utilization, when scaling out in deep learning. We also experiment with the linear scaling rule and learning rate warmup methods, to address the model quality related issues that are present when training is conducted with larger batch sizes. In our experiments, we measure the scaling performance of our distributed programs and the model quality implications of naïvely scaling out in deep learning vs. scaling out utilizing the linear scaling rule and learning rate warmup methods. Our experiments provide concrete examples on how to apply these tools and methods in the provided execution environment, and they also provide answers for the following research questions:
RQ1: What are the implications of using smaller per GPU worker batch sizes vs. using larger per GPU worker batch sizes in terms of scaling performance, training time (wall clock) and efficient resource utilization in the provided execution environment?
RQ2: Does the Horovod distributed training framework provide a speedup in the provided execution environment when scaling out training?
RQ3: What are the model quality implications of larger global batch sizes when utilizing methods such as linear learning rate scaling and gradual learning rate warmup?
The tools and methods utilized in our experiments, enabled us to efficiently scale out a 1000 class image classification problem to 32 GPU workers, without degrading the resulting model quality.

Keywords:	Distributed Deep Learning, Horovod, PyTorch, HPC, CSC
Number of pages:	79
Language:	English
Date of acceptance:	31.05.2019

CONTENTS

Figures

Tables

1 INTRODUCTION

Deep learning, a subset of machine learning, has long been prominent in the areas of image classification, speech recognition, automatic machine translation and autonomous vehicles (Ben-Nun and Hoefler, 2018). Today deep neural networks are able to produce accurate solutions for problems that were previously thought to be unsolvable, merely by observing a vast amount of training data (Ben-Nun and Hoefler, 2018).

The availability of powerful Graphics Processing Units (GPU) has accelerated the adoption of deep learning (Schmidhuber, 2015). Training of neural networks usually involve a plethora of matrix-matrix multiplication operations, which can be assigned to a GPU to speed up computation (Schmidhuber, 2015; Ben-Nun and Hoefler, 2018). Results dating back to year 2006 already demonstrated that GPU-based convolutional neural network (CNN) training can be up to four times faster when compared to CPU-based CNN training (Schmidhuber, 2015). The available hardware and methods have evolved a lot since 2006.

As the solvable problems become more complex and the datasets grow in size and complexity, a distributed parallel computing setting may be required to speed up training and inference. Many of the prominent Python deep learning frameworks available today, such as TensorFlow and PyTorch, natively support the parallel and distributed parallel computing setting. In addition, distribution frameworks, such as the Horovod framework, have been developed with the aim to speed up training and simplify the process of parallelizing and distributing programs that have initially been written for the non-parallel non-distributed single worker setting (Sergeev and Del Balso, 2018).

Large-scale distributed deep learning also involves algorithmic challenges (Ma *et al.*, 2018). When scaling out in deep learning, we will have to address issues such as inefficient resource utilization due to increased communications overhead (Sergeev and Del Balso, 2018) and degraded model quality due to increased global batch size (Keskar *et al.*, 2016).

In this Master's thesis we will address these algorithmic challenges. We examine a coherent method for scaling out in deep learning, that has been successfully applied by Facebook AI Research to scale out a 1000 class image classification problem to 256 GPU workers (Goyal *et al.*, 2017). We will also try out the Horovod framework, which employs efficient communications techniques, that can be utilized to mitigate some of the communications overhead that is present when scaling out in deep learning (Sergeev and Del Balso, 2018).

1.1 Background and need

This work is done as part of a service development task of distributed deep learning on the CSC provided infrastructure. The aim is to improve the readiness to provide a service for AI researchers who wish to scale out in deep learning to benefit from the potential speedup gains of distributed deep learning. Speeding up the training process, enables the researchers to attain their results for their experiments faster, thus potentially speeding up the research cycle.

1.2 The practical problem

Large-scale distributed deep learning is a multidisciplinary challenge. Large-scale distributed deep learning involves expertise in the areas of algorithms, communication libraries, deep learning frameworks and infrastructure engineering (Akiba, Suzuki and Fukuda, 2017; Ben-Nun and Hoefler, 2018; Ma *et al.*, 2018).

According to Ma et al., the algorithmic challenges, when scaling out in deep learning, lie in increased global batch size and in increased communication overhead (Ma *et al.*, 2018). It has previously been observed, that models which have been trained with larger batch sizes, tend to have generalization issues and decreased accuracy (Keskar *et al.*, 2016).

According to Ma et al., methods such as linear learning rate scaling and gradual warmup (Goyal *et al.*, 2017) can be applied to mitigate the model quality issues that are present when training is conducted with larger global batch sizes. Also, frameworks such as Ho-

rovod, which employ efficient communications techniques (Patarasuk and Yuan, 2009), can be utilized to alleviate some of the communication overhead that is present when scaling out (Sergeev and Del Balso, 2018).

1.3 Aim of the project

CSC is improving its readiness to offer a service for AI researchers, by internal competence development in the distributed deep learning domain. This thesis is an integral part of this competence development effort. With the help of the gained knowledge, we aim to better serve researchers who wish to scale out their deep learning research in the CSC provided infrastructure.

This will be achieved by providing examples, in the form of code and scripts, on how scale out in deep leaning in the provided execution environment, and to provide examples, on how to measure the scaling performance of distributed programs. The provided examples can later be used as source material for a user guide on how to scale out deep learning algorithms on CSC provided infrastructure.

In this thesis, we aim to attain hands-on experience and competences in the following areas:

Efficient communication methods
Communication may become a bottleneck, when scaling out training (Ma *et al.*, 2018; Sergeev and Del Balso, 2018); We aim to test the Horovod framework (Sergeev and Del Balso, 2018), which utilizes efficient communication methods (Patarasuk and Yuan, 2009), to measure if the framework provides a speedup, when scaling out training in the provided execution environment.

Maintaining model quality
When scaling out training, the resulting model quality tends to degrade, due to the increase in global batch size (Keskar *et al.*, 2016; Ma *et al.*, 2018); We aim to replicate a method for scaling out in deep learning, which has been successfully applied by Facebook AI Research. Facebook managed to scale out a 1000 class image classification

problem to 256 GPUs, without degrading the resulting models quality (Goyal *et al.*, 2017).

1.4 Research questions and methodology

We will base our research on reproduceable experiments. The research experiments, will be based on the One-factor-at-a-time (OFAT) method, where we adjust a single input parameter, and measure or observe its effects on the output. The strength of the OFAT method is that it provides the researcher an ability to focus on individual parameters influence on the end result.

The scripts used to launch our research experiments in the provided execution environment, and the relevant experiment settings, are appended as appendices to this thesis. The results of our experiments, and the information recoded in the appendices, can later serve as valuable source material for a user guide on how to scale out in deep learning in the provided execution environment.

With our research experiments, we aim to gain better insight and hands-on experience, on how to scale out in deep learning in the provided execution environment, and to answer our research questions, which are posted below. For more detailed information regarding our research experiments and the execution environment, see chapter 4.

The following research questions (RQ) are posted for the thesis:

RQ1: What are the implications of using smaller per GPU worker batch sizes vs. using larger per GPU worker batch sizes in terms of scaling performance, training time (wall clock) and efficient resource utilization in the provided execution environment?

RQ2: Does the Horovod distributed training framework provide a speedup in the provided execution environment when scaling out training?

RQ3: What are the model quality implications of larger global batch sizes when utilizing methods such as linear learning rate scaling and gradual learning rate warmup?

1.5 Definitions

CNN	Convolutional neural network
CuDNN	NVIDIA CUDA Deep Neural Network library
Epoch	One pass over all the samples in the dataset set.
Global batch size	The combined per worker batch size of all workers.
GPU	Graphics Processing Unit
FP16	Floating point precision of 16 / Half- precision
FP32	Floating point precision of 32 / Single-precision
HPC	High-performance computing
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
MPI	Message-Passing Interface
NCCL	NVIDIA Collective Communications Library
OFAT	One-factor-at-a-time
Per worker batch size	The mini-batch size that one worker utilizes.
SGD	(mini-batch) Stochastic Gradient Descent

2 METHOD REVIEW

In this chapter we will go through some of the terminology and background information needed to better understand the concepts that are outlined in the later chapters.

2.1 Deep learning and the backpropagation algorithm

Deep learning is a subset of machine learning, where learning representations from data involves learning from successive layers of increasingly meaningful representations (Chollet, 2017 p. 8). These layered representations are commonly learned via models called (deep) neural networks (Lecun, Bengio and Hinton, 2015). According to Chollet, deep learning often involves tens or even hundreds of successive layers of representation, where the representations are learned automatically during the training phase of the neural network (Chollet, 2017 p. 8). According to Lecun, Bengio and Hinton, deep learning methods have greatly improved the state-of-the-art in object detection, visual

object recognition, speech recognition and other domains such as genomics and drug discovery (Lecun, Bengio and Hinton, 2015).

According to Lecun, Bengio and Hinton, the most common form machine learning is supervised learning (Lecun, Bengio and Hinton, 2015). Supervised learning involves mapping input data to targets, which is achieved by allowing the neural network to observe examples of inputs and their corresponding true targets (ground truth) (Chollet, 2017 p. 9).

The layers in the deep neural network apply data transformations to the input data based on numerical parameters called weights. Learning, in this sense involves finding a set of values for the weights, for all the layers in the network, which will correctly map the inputs to their corresponding true targets (Chollet, 2017 p. 10). According to Lecun, Bengio and Hinton, a typical deep learning system may involve hundreds of millions of adjustable weights, and hundreds of millions of labelled training examples (Lecun, Bengio and Hinton, 2015).

To be able to control the output of our neural network, we need to measure how well our networks predictions (output) matches with the ground truth (Chollet, 2017 p 10). The loss function takes our networks predictions and the ground truth as input parameters. It then calculates a distance score (loss score) based on these parameters, which is used to indicate how well our predictions match with the ground truth (Chollet, 2017 p 10). We typically use different implementations of the loss function depending on the problem that we are solving. For example, we use mean squared error loss for regression problems, binary cross-entropy loss for two-class classification problems and categorical cross-entropy loss for a multi-class classification problems (Chollet, 2017 p. 60).

To update our networks weights, we use the calculated loss score as a feedback signal, to tune each layers weights in a direction, that will lower our loss score (Chollet, 2017 p. 11). The weight adjustment procedure is handled by the optimizer, which implements the backward pass of the backpropagation algorithm (Rumelhart, Hinton and Williams, 1986). According to Chollet, in the backward pass of the backpropagation algorithm, we start with the final loss score, and then proceed to traverse though the networks layers in

reverse, from output to input, applying the chain rule of calculus to compute the contribution (error gradient values) that each weight had in the loss score (Chollet, 2017 p. 52). Then, in the final step of the backpropagation algorithm, we update the weights by applying a gradient descent step on all the weights in the network, by utilizing the error gradient values that we attained during backward pass of the algorithm (cp. Géron, 2007 p. 266).

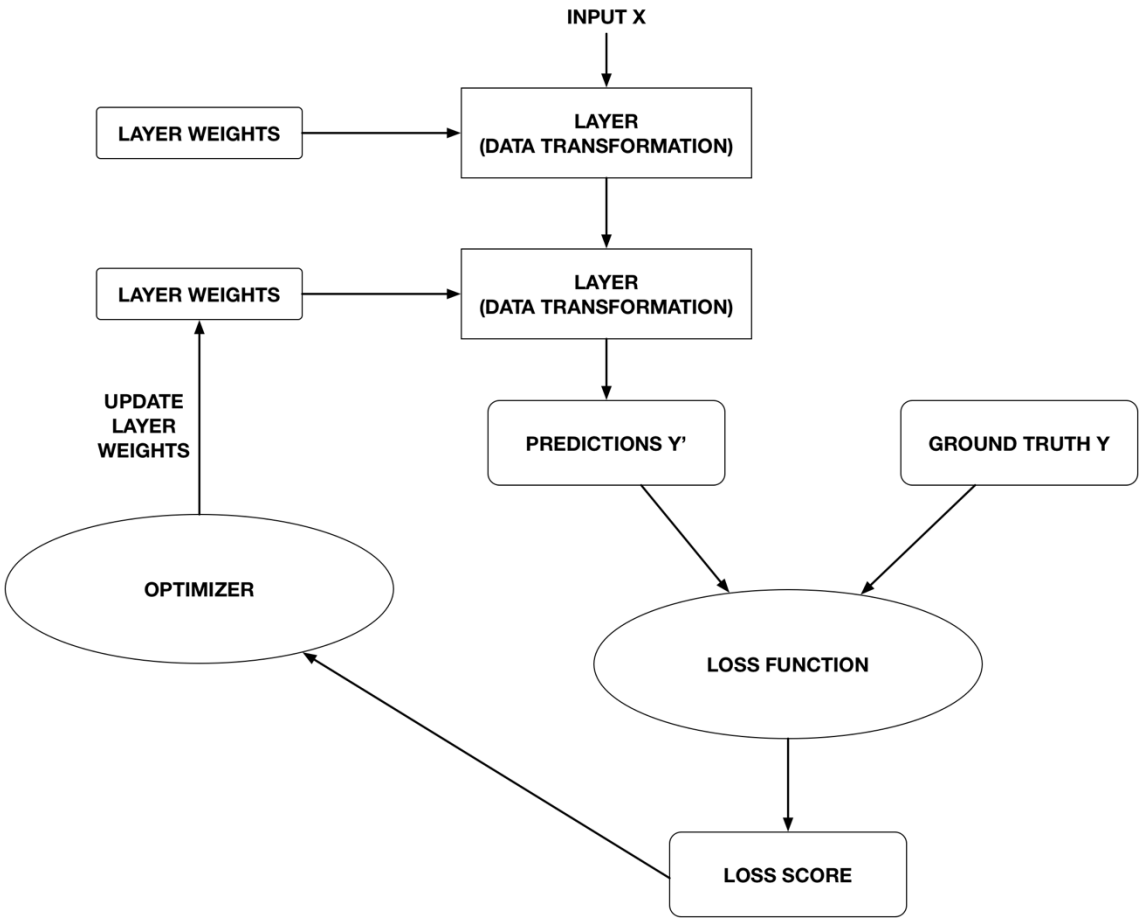

Figure 1. Conceptual illustration of model weight update via backpropagation

2.2 Gradient descent and learning rate

Gradient descent is a generic optimization algorithm capable of finding an optimal solution for a plethora of different types of problems (see Géron, 2007 p. 113). The core idea in gradient descent is to tweak parameters indefinitely in order to minimize a loss function. In gradient descent, we measure the local gradient of the loss function, with

regards to the parameter (weight) vector θ and move in the direction of the descending gradient. We have reached a minimum when the gradient is zero (see Géron, 2007 p. 114).

We start of by initializing parameter vector θ with random values, and then we improve θ gradually one learning step at a time, by attempting to decrease the loss function until we converge to a minimum (see Géron, 2007 p. 114).

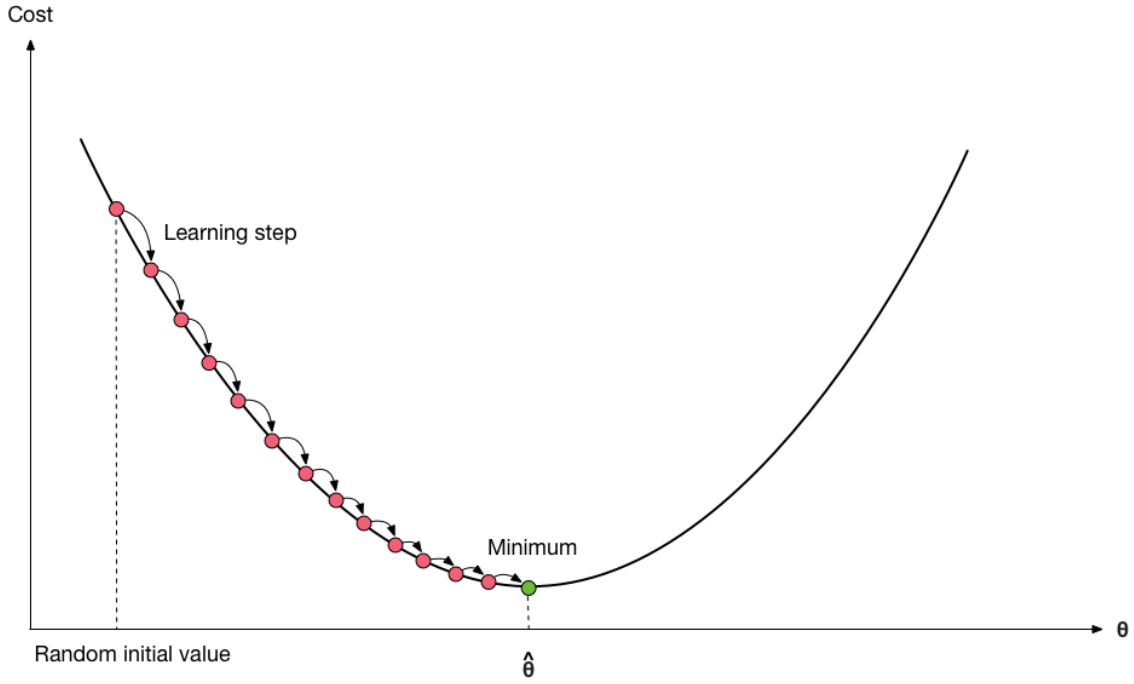

Figure 2. Conceptual illustration of gradient descent

In gradient descent the learning step size is determined by the *learning rate* hyperparameter (γ). If we set the learning rate to a too small value, then we have to repeat training over many more iterations for the algorithm to converge (see Géron, 2007 p. 114).

Gradient descent

$$\theta_{n+1} = \theta_n - \gamma * \nabla f(\theta_n)$$

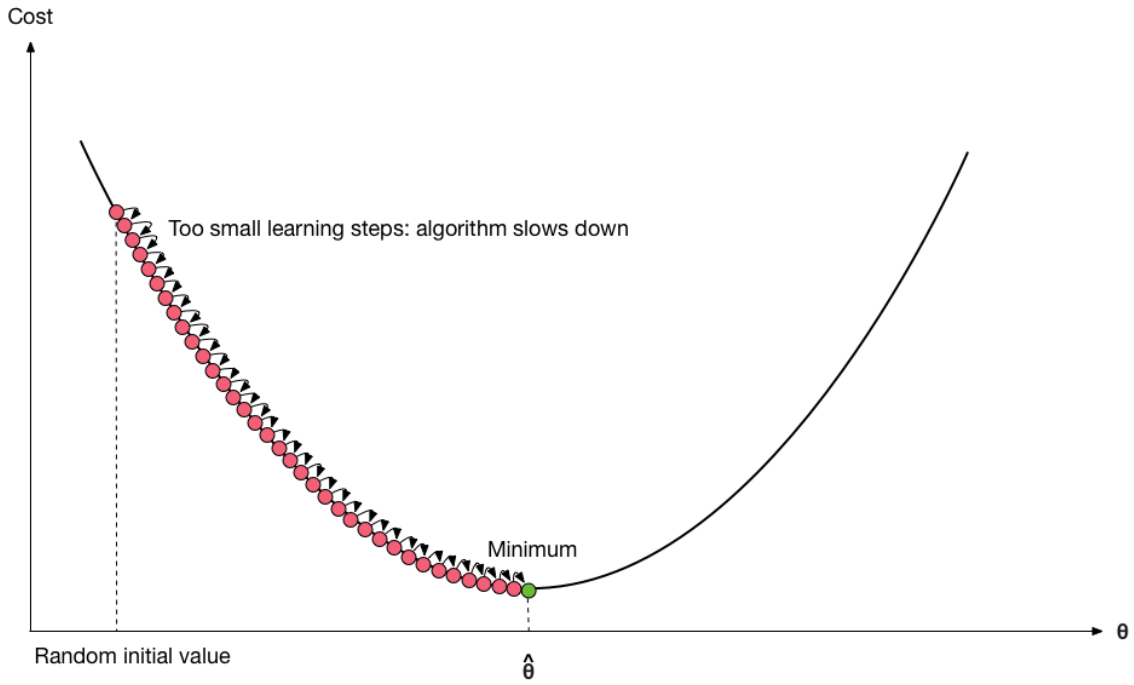

Figure 3. Conceptual illustration of gradient descent with a small learning rate

If the learning rate is set is set to a too high value, then the algorithm might diverge and fail to find a good solution (see Géron, 2007 p. 114).

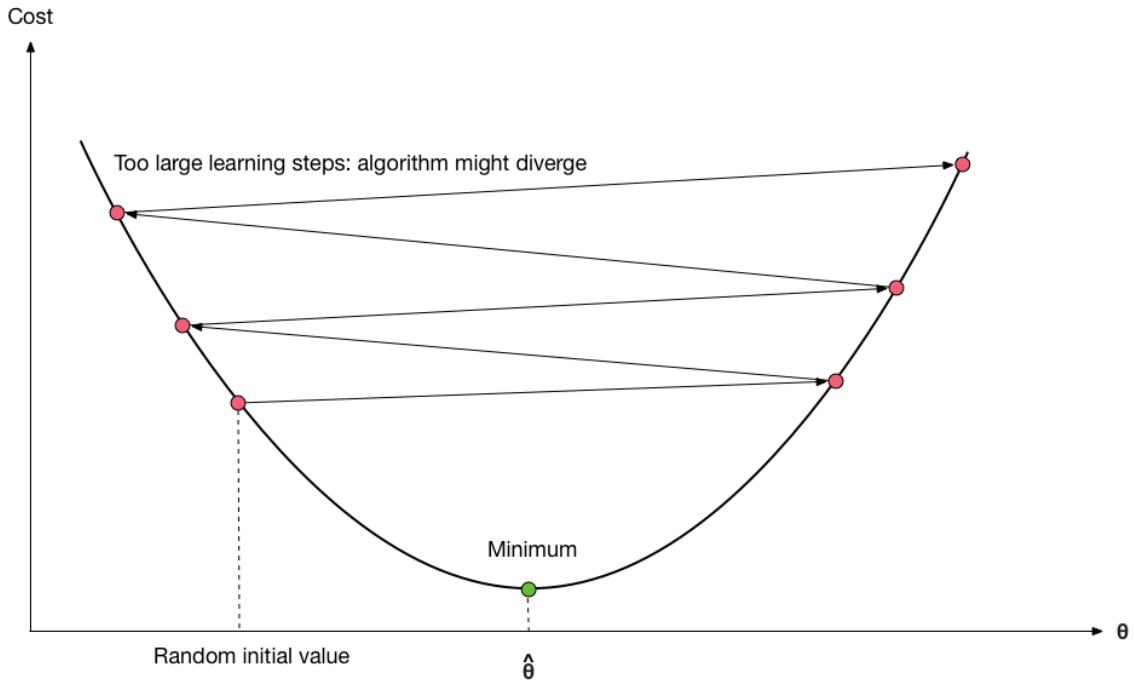

Figure 4. Conceptual illustration of gradient descent with a large learning rate

All loss functions are not bowl shaped (see Géron, 2007 p. 115). The terrain might include plateaus, ridges and valleys, making it hard for the algorithm to converge at a global minima. According to Géron, the two main pitfalls of gradient descent are valleys or plateaus (Géron, 2007 p. 115). Valleys can cause the algorithm to converge at a local minima and plateaus can slow down the algorithm, since they usually take a long time for the algorithm to traverse.

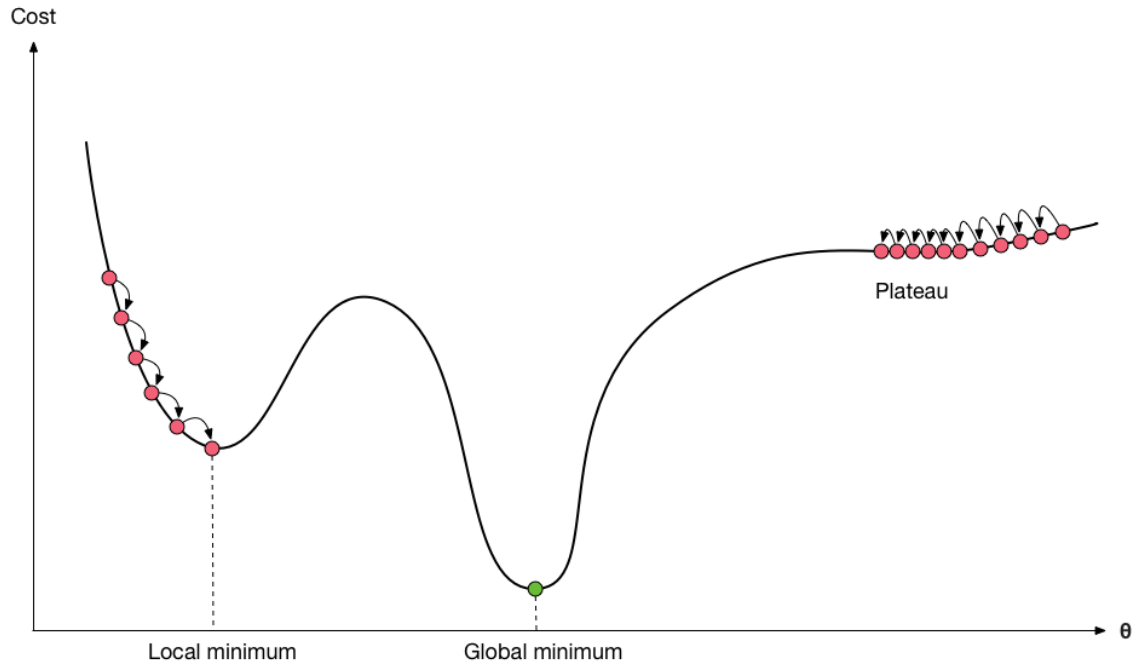

Figure 5. Conceptual illustration of common gradient descent pitfalls

2.2.1 Stochastic gradient descent

With the (batch) gradient descent algorithm we use the whole training set to compute the gradients at each learning step. Computing gradients over the whole training set will slow down the algorithm if the training set is large (Géron, 2007 p. 120). If the training set is very large, we also might not be able to fit the whole training set into the computer's memory. With the stochastic gradient descent algorithm we randomly select a single sample from the training set and compute the gradients based on the selected sample. Calculating the gradients based on a single sample, speeds up the algorithm and enables us to train with larger datasets (Géron, 2007 p. 120).

Due to its random nature, the stochastic gradient descent algorithm is less precise than the batch gradient descent algorithm (Géron, 2007 p. 120). Instead of steadily decreasing to a minima, the loss function will oscillate, and decrease only on average, and we will end up finding an good solution, but not the optimal one (Géron, 2007 p. 120). On the other hand, the algorithms random nature can help it escape local minimum, which means that the stochastic gradient descent algorithm has a better chance of finding global minimum, than the batch version of the algorithm (Géron, 2007 p. 121).

2.2.2 Mini-batch gradient descent

The mini-batch (stochastic) gradient descent algorithm represents a trade-off between the stochastic gradient descent algorithm and batch gradient descent algorithm (Ben-Nun and Hoefler, 2018). Instead of computing gradients based on a single sample or based on the whole training set, we compute the gradients based on a small random set of samples called mini-batches (Géron, 2007 p. 123). According Géron, the main advantage of mini-batch gradient descent over stochastic gradient descent is, that with mini-batch gradient descent, we are able to utilize our hardware more efficiently, especially when utilizing GPUs for matrix operations (Géron, 2007 p. 123).

With fairly large mini-batch sizes, the mini-batch gradient descent algorithm is usually able to find a better solution than the stochastic gradient descent algorithm (Géron, 2007 p 123). Mini-batch gradient descent also provides us with a straightforward approach for parallelization of computation (Ben-Nun and Hoefler, 2018). When parallelizing computation, the mini-batch samples are partitioned among multiple computational resources e.g. CPU or GPUs cores (Ben-Nun and Hoefler, 2018). According to Ben-Nun and Hoefler, this sort of data parallelism is supported by most of the currently available deep learning frameworks, using a single GPU, multiple GPUs (local data parallel), or a cluster of multi-GPU nodes (distributed data parallel) (Ben-Nun and Hoefler, 2018).

2.3 Learning rate scheduling

Finding the optimal learning rate can be a tedious and time consuming task. When the learning rate is set way too high, then training might diverge. When the learning rate is

set too low, then training will be very slow, but will eventually converge near the optimum. When the learning rate is set slightly too high, then training progresses quickly in the beginning, but ends up oscillating around the optimum. (Géron, 2007 p. 307)

Learning rate scheduling is the task of reducing the learning during the training process based on a learning rate reduction strategy (learning rate schedule). Usually we want to use a higher learning rate during the initial phases of training, and then reduce the learning rate once we stop making fast progress. Optimally, with the help of learning rate scheduling, we are able to find a good solution faster (in fewer epochs) than using a good constant learning rate. (Géron, 2007 p. 307 - 308)

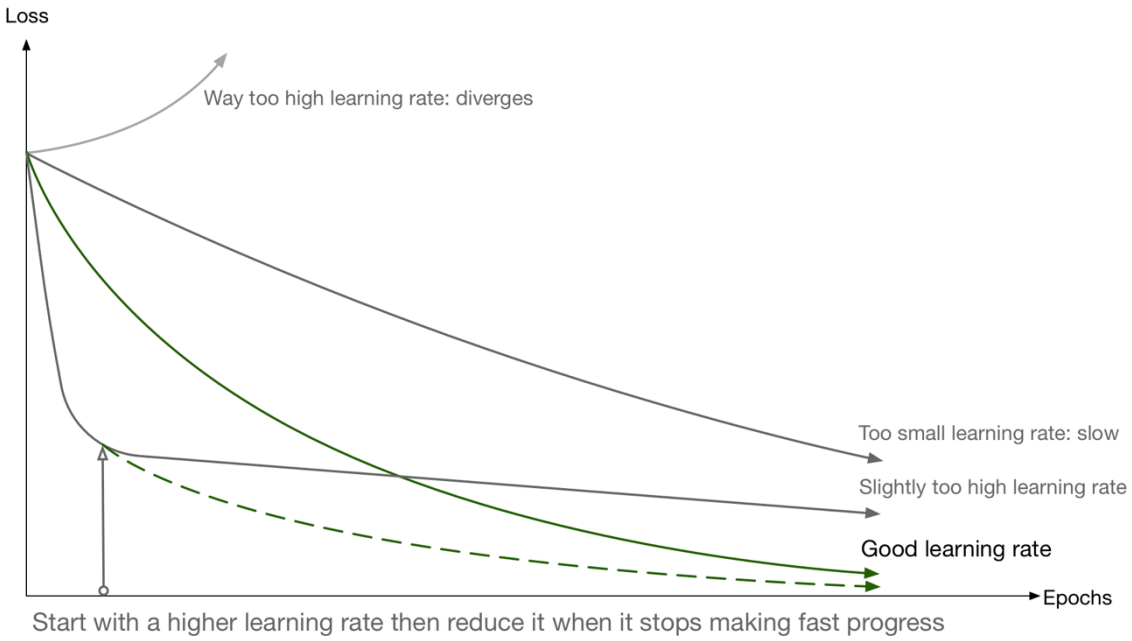

Figure 6. Conceptual illustration of learning rate and learning rate scheduling

There are a number of different learning rate schedules available. A paper by Senior et al. compares the performance of popular learning rate schedulers when utilized in the task of training deep neural networks for speech recognition (Senior *et al.*, 2013). The authors concluded that for their task both *exponential scheduling* and *performance scheduling* yielded good results. The authors favored exponential scheduling over performance scheduling, because for their task exponential scheduling converged slightly faster and the algorithm was easier to implement and tune.

2.3.1 Exponential scheduling

In exponential scheduling the learning rate is set to a function of the iteration number t (see Géron, 2007 p. 307 - 308).

Exponential scheduling

$$\gamma(t) = \gamma_0 \, 10^{-t/r}$$

2.4 Momentum optimization

Momentum optimization can be applied to speed up the gradient descent algorithms (Polyak, 1964). Momentum optimization introduces a new hyperparameter called *momentum* (μ) which is typically set to the value 0.9 (see Géron, 2007 p. 300).

Momentum assisted gradient descent

$$\theta_{n+1} = \theta_n + \mu * (\theta_n - \theta_{n-1}) - \gamma * \nabla f(\theta_n)$$

Momentum optimization enables the gradient descent algorithm to escape loss landscape consisting of plateaus and valleys in a faster pace, at the cost of the algorithm possibly overshooting a bit, when it reaches minimum (Géron, 2007 p. 300). Overshooting will typically cause the algorithm to oscillate around the minimum, before stabilizing at the minimum (Géron, 2007 p. 300). A common variant of the momentum optimization algorithm is the Nesterov Accelerated Gradient (Nesterov, 1983).

2.5 Weight decay

Weight decay is a regularization technique which helps to reduce overfitting. We reduce overfitting by making the distribution of weight values more regular, by forcing the networks weights to only take on small values (Chollet, 2017 p. 107). This is typically achieved by adding a cost associated with having large weights, which is applied during the weight update step (see Chollet, 2017 p. 107).

2.6 Data augmentation

Data augmentation is a regularization technique which helps to reduce overfitting. The idea is to generate new training samples from existing ones, to boost the size of the training set, which helps to reduce overfitting (see Géron, 2007 p. 307 - 315) (Krizhevsky, Sutskever and Hinton., 2012). The initial training sample can for example be rotated, cropped, zoomed or flipped horizontally to generate a new training sample. The generated training sample maintains the class label of the initial training sample, and can thus be used to increase the size of the training set.

Figure 7. Data augmentation. Example of generated training samples.

2.7 ImageNet and the ILSVRC 2012 dataset

The ImageNet project is an ongoing research effort to provide researchers with an easily accessible image database for visual object recognition software research (ImageNet, 2016). The ImageNet image database is organized conforming with the WordNet hierarchy (ImageNet, 2016; WordNet, 2019). Synonym sets, which may consist of multiple words or word phrases, are used to describe meaningful concepts in WordNet (ImageNet, 2016). WordNet consists of more than 100 000 synonym sets, of which 80 000 are nouns (ImageNet, 2016). The ImageNet project aims to provide 1000 images on

average to illustrate each synonym set (ImageNet, 2016). The images of each synonym set are cross-checked and annotated by human operators.

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was a competition in algorithm development for object detection and image classification, that was first organized in 2010 (ILSVRC, 2015). According to Russakovsky et al. ILSVRC has established itself as the standard benchmark for large-scale object recognition (Russakovsky *et al.*, 2015). As a standard benchmark, the ILSVRC 2012 dataset is often found utilized in research articles and journals that deal with object recognition and image classification (Krizhevsky, Sutskever and Hinton., 2012; Gupta, Zhang and Milthorpe, 2015; Akiba, Suzuki and Fukuda, 2017; Goyal *et al.*, 2017).

The ILSVRC 2012 training set is a subset of ImageNet containing 1000 categories and 1.2 million images. The ILSVRC 2012 validation set consists of 50 000 images with their corresponding labels.

3 LITERATURE REVIEW

In the literature review chapter, we aim to shed some light on the peculiarities of distributed deep learning. First we look at commonly used parallelization and distribution strategies. Then we consider parameter distribution strategies in parallel and distributed training. After considering parameter distribution strategies, we examine the guidelines for selecting per worker batch size. And finally, we look at a distribution framework and two large-batch methods, that can be applied for overcoming the algorithmic challenges of distributed deep learning, particularly the communication and model quality related issues, that we must address when scaling out in deep learning.

3.1 Parallel and distributed computing

There are several ways to parallelize and distribute computation across multiple machines and computational resources (cores or GPUs). In the following section we look at some of the commonly used parallelization and distribution strategies.

3.1.1 Local strategies

In local training we assume that the whole model and the dataset can be fit into the memory of a single machine (Hegde and Usmani, 2016).

Local training, data parallel: In the data parallel computing setting, the dataset is partitioned across multiple local computational resources (cores or GPUs) on a single machine (Hegde and Usmani, 2016). Here we want to achieve faster training by parallelizing the training process across multiple local computational resources.

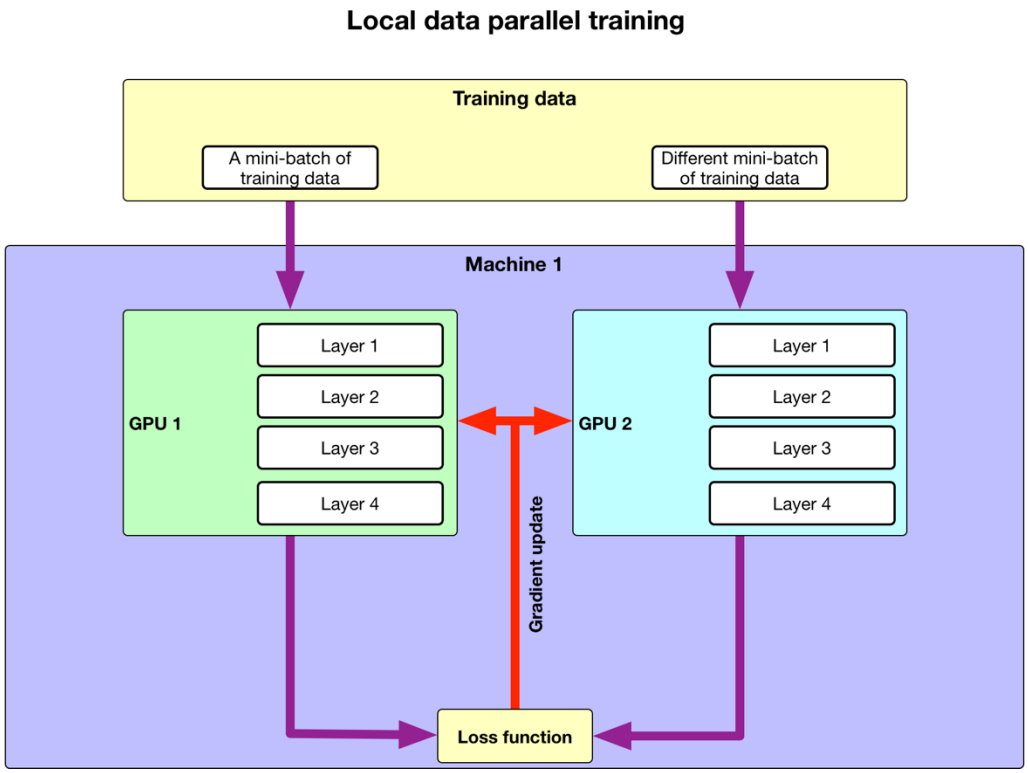

Figure 8. Conceptual illustration of local data parallel training

3.1.2 Distributed strategies

When datasets or models grow in size, and when it is no longer possible to fit the whole dataset and/or model into the memory of a single machine, it may become necessary to split the dataset and/or model across multiple machines (Xiang *et al.*, 2014; Hegde and Usmani, 2016).

Distributed training, data parallel: In the distributed data parallel computing setting, the dataset is partitioned across multiple machines. We use this approach when the dataset is too large to be processed on a single machine or when we want to achieve faster training by distributing the training process across multiple machines (Hegde and Usmani, 2016). The machines that participate in the distributed data parallel training process can also have multiple local computational resources (cores or GPUs) which can be utilized during the distributed training process.

Distributed training, model parallel: When the model is too large to fit into the memory of a single machine, it may become necessary to split the model across multiple machines. A single network layer could for example fit into the memory of a single machine, thus, forward and backward propagation would involve communication of output from one machine to another, in a serial fashion (Hegde and Usmani, 2016). According to Ben-Nun and Hoefler, deep neural network architecture creates network layer interdependencies (Ben-Nun and Hoefler, 2018). These interdependencies generate communication, which in turn will affect the overall performance of a deep neural network that utilizes model parallelism. As a concrete example of increased communication caused by network layer interdependency Ben-Nun and Hoefler exemplifies the fully connected layers that involve all-to-all communication, as neurons from the current layer connect to all the neurons of the next layer (Ben-Nun and Hoefler, 2018). According to Hegde and Usmani, we resort to model parallelism only when the model cannot be fit into the memory of a single machine, and not so much to speed up the training process (Hegde and Usmani, 2016).

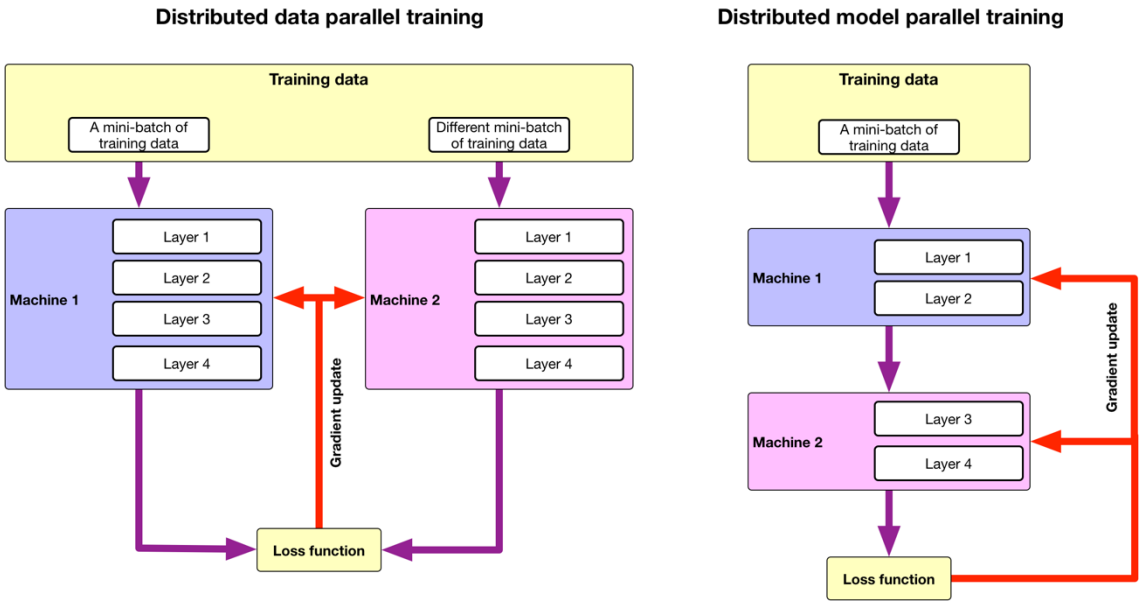

Figure 9. Conceptual illustration of distributed data parallel and model parallel training

Distributed training, hybrid parallel: According to Ben-Nun and Hoefler, in the hybrid parallel scheme we combine multiple parallelization schemes, with aim to overcome the drawbacks of each combined scheme (Ben-Nun and Hoefler, 2018).

3.2 Parameter update and distribution methods in data parallel and distributed data parallel training

In data parallel training each worker computes gradients for a chunk (mini-batch) of training data. These computed gradients are then collected from the workers and averaged. The averaged gradients are then used to update the model, after which the workers use this updated model to compute gradients for a new mini-batch of training data (Sergeev and Del Balso, 2018).

Parallel and distributed parallel training require a means of combining the workers results and synchronizing model parameters between the workers. The synchronization protocol can be synchronous or asynchronous and the network architecture can be centralized or decentralized (Ben-Nun and Hoefler, 2018).

3.2.1 Centralized architecture

According to Ben-Nun and Hoefler, centralized network architectures typically involves one or more special purpose nodes called parameter servers (Ben-Nun and Hoefler, 2018). The parameter servers maintain the current state of the model parameters (Akiba, Fukuda and Suzuki, 2017). Parameter servers can increase the clusters fault tolerance by periodically persisting the models state in a checkpoint file. The checkpoint file can then be utilized to restart training using the stored models parameters (Ben-Nun and Hoefler, 2018). According to Ben-Nun and Hoefler, parameter servers can also increase the clusters fault tolerance, by enabling dynamic spin-up and removal of workers during the training process (Ben-Nun and Hoefler, 2018).

In the centralized network architecture scheme, the workers perform their forward pass and backward pass calculations (Rumelhart, Hinton and Williams, 1986) on their current mini-batch of training data, and then send the resulting gradients to the parameter server (Akiba, Fukuda and Suzuki, 2017). The parameter server then uses these collected gradients to update the model. The workers receive the updated model from the parameter server and proceed to calculate gradients for the next mini-batch of training data (Akiba, Fukuda and Suzuki, 2017). In larger clusters, the parameter server may become a network bottleneck, since all workers need to communicate with the parameter server during gradient updates, and the parameter server needs to send the updated model to all workers after it has finished averaging all the gradients. According to Iandola et al., a parameter servers communication time scales linearly as we increase the number of GPU workers (Iandola *et al.*, 2015). Thus, doubling the number of GPU workers leads to at least 2x more communication time per gradient update (Iandola *et al.*, 2015). To mitigate some of the communication overhead, multiple parameter servers can be added to the computing cluster. For example, Project Adam utilizes a sharded parameter server infrastructure (Chilimbi *et al.*, 2014).

3.2.2 Decentralized architecture

The decentralized network architecture scheme does not involve a parameter server. In the decentralized network architecture scheme, the parameter updates typically rely on an allreduce operation, to communicate parameter updates among all the workers

(Akiba, Fukuda and Suzuki, 2017; Ben-Nun and Hoefler, 2018; Sergeev and Del Balso, 2018). The decentralized update is then computed separately on each worker, where every worker creates its own optimizer (Akiba, Fukuda and Suzuki, 2017; Ben-Nun and Hoefler, 2018).

3.2.3 Synchronous communication

In the synchronous communication setting, the cluster has to wait for all participating workers to finish their gradient calculations, before the gradient update can commence (Dutta *et al.*, 2018). A straggling worker can thus slow down the computing cluster in the synchronous communication setting. There are different methods for handling straggling workers in a synchronized communication setting. For example, Teng and Wood suggest a method, in which a model of worker runtimes can be used to predict order statistics, that allow for a near optimal choice of straggler cut-off that maximizes gradient computation throughput (Teng and Wood, 2018).

3.2.4 Asynchronous communication

In the asynchronous communication setting, the cluster does not wait for straggling workers, and stale gradients are used during parameter updates. According to Chen et al. stale gradients are stochastic gradients computed based on outdated parameters, instead of using the latest parameters (Chen *et al.*, 2016). Asynchronous communication usually results in faster processing at the cost of model consistency (Chen *et al.*, 2016; Ben-Nun and Hoefler, 2018).

According to Ben-Nun and Hoefler, the prominently used parameter update approaches use synchronous communication for up to 32 - 50 nodes, where the allreduce operation still scales almost linearly (Ben-Nun and Hoefler, 2018). Asynchronous updates are usually used in heterogeneous environments and larger clusters (Dai *et al.*, 2014; Ben-Nun and Hoefler, 2018).

3.3 Guidelines on selecting per worker batch size when scaling out in deep learning

As noted in the introduction chapter, the algorithmic challenges when scaling out in deep learning, lie in an increased global batch size and in an increased communication overhead (Ma et al., 2018). Models which have been trained with larger batch sizes, tend to have generalization issues and decreased accuracy (Keskar et al., 2016).

According to Ben-Nun and Hoefler, selecting the optimal per worker batch size is a complex optimization space on its own merit (Ben-Nun and Hoefler, 2018). When the per worker batch size is set too small, then we are not able to utilize our hardware to its full extent, typically due to communication overhead (Ben-Nun and Hoefler, 2018). If the per worker batch size is set too high, then the quality of the resulting model starts to degrade, once the global batch size increases beyond a certain point (Ben-Nun and Hoefler, 2018).

As illustrated in Figure 10, when the per worker batch size is set to a too small value (region A), then we are not able to efficiently utilize our hardware. If the per worker batch size is set too large (region C), then the global batch size surpasses the point where the resulting models quality starts to degrade. (Ben-Nun and Hoefler, 2018)

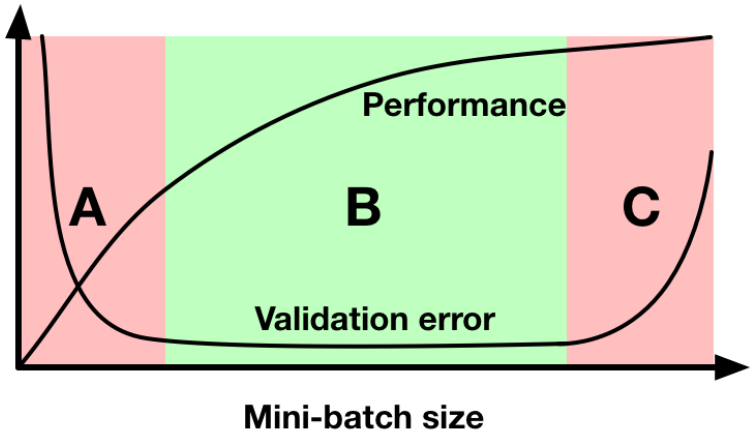

Figure 10. Selecting optimal per worker batch size. Adapted from (Ben-Nun and Hoefler, 2018)

3.4 Horovod distributed training framework

Horovod is a distributed training framework for deep learning, which currently supports distributed training with the TensorFlow, Keras, PyTorch and MXNet deep learning frameworks (Horovod, 2019). The framework has been designed from an HPC background, and is able to utilize an available MPI implementation (Message Passing Interface Forum, 2012) for transparently setting up communication infrastructure that enables workers to find and communicate with each other (Sergeev and Del Balso, 2018).

The Horovod framework website provides comprehensive examples on how to refactor single worker programs, to support local data parallel or distributed data parallel training with the Horovod framework (Horovod, 2019). Typically, only between 5 to 10 lines of code need to be changed, when refactoring a TensorFlow, Keras, PyTorch, or MXNet single worker program, to enable the program to utilize the Horovod framework for distribution (Prace, 2019).

The Horovod framework utilizes the decentralized network architecture scheme combined with the synchronous communication scheme, which enables it to scale out more efficiently (Sergeev and Del Balso, 2018). In a paper by Sergeev and Del Balso, the authors compare the scaling performance of standard distributed TensorFlow with Horovod distributed TensorFlow (Sergeev and Del Balso, 2018). In their paper, the authors note, that they were only able to leverage 50 percent of their computing clusters resources, due to communication overhead, when scaling out training to 128 GPU workers, while using standard distributed TensorFlow and its centralized network architecture scheme (Sergeev and Del Balso, 2018). Using the Horovod framework to distribute the TensorFlow program, greatly improved their ability to scale out training; Sergeev and Del Balso report achieving 88 percent scaling efficiency, when scaling out training to 128 GPU workers with the Horovod framework (Sergeev and Del Balso, 2018). According to Sergeev and Del Balso, when performing parameter updates, the Horovod framework utilizes the ring-allreudce and Tensor Fusion algorithms, which enable the framework to more optimally utilize the available network bandwidth (Patarasuk and Yuan, 2009; Sergeev and Del Balso, 2018).

3.5 Linear learning rate scaling and gradual learning rate warmup methods

When scaling out training in deep learning, the global batch size increases as we increase the worker count. Large global batch sizes have traditionally been a limiting factor in scaling out in deep learning, since models trained with large batch sizes tend to have generalization issues and decreased accuracy (Keskar *et al.*, 2016).

A paper by Goyal et al. introduces two efficient methods, that can be applied when scaling out in deep learning, that enable us to train with larger global batch sizes, without degrading the resulting models quality as we scale out (Goyal *et al.*, 2017).

The first method, the linear scaling rule, involves increasing the learning rate linearly, in proportion to the increase in global batch size, when scaling out (Goyal *et al.*, 2017). The linear scaling rule enables us to scale out, without adjusting any other hyperparameters except the learning rate, which makes it a straight forward method to implement and debug (Goyal *et al.*, 2017).

Linear Scaling Rule

When the (global) batch size is multiplied by k, multiply the learning rate by k.

The second method, learning rate warmup, involves utilizing a smaller learning rate during the initial epochs of training, and then gradually increasing the learning rate in small steps, to the final linearly scaled learning rate (Goyal *et al.*, 2017). Increasing the learning rate in small steps, during the early stages of training, when the network is changing rapidly, allows for a more healthy convergence at the start of training (Goyal *et al.*, 2017). A pseudocode example for implementing a 5 epoch gradual learning rate warmup period can be seen in Appendix 13.

When training with large global batch sizes, Goyal et al. suggests, that the per worker batch size should be kept the same across all workers. The per worker batch size is a key component of the loss. If we change the per worker batch size, it affects the underlying loss function that is being optimized. Using mixed per worker batch sizes has an

negative impact on Batch Normalization (Ioffe and Christian Szegedy, 2017) statistics, which are calculated at worker level. (Goyal *et al.*, 2017).

In their paper, Goyal et al. also evaluated the linear scaling rule and gradual warmup methods applicability on other computer vision related tasks. According to the authors, they observed no generalization issues, when applying the same methods for an image detection / segmentation task, this time utilizing the COCO dataset (COCO, 2018), while training with large batch sizes. (Goyal *et al.*, 2017)

The linear scaling rule and learning rate warmup methods cannot be applied to raise the global batch size indefinitely. According to Goyal et al., global batch size has an upper bound, after which the resulting models accuracy starts to degrade (Goyal *et al.*, 2017). For the ILSVRC 2012 dataset, the upper bound for global batch size is 8k, when utilizing the linear scaling rule and learning rate warmup methods (Goyal *et al.*, 2017). According to Ben-Nun and Hoefler, large-batch methods, such as the linear scaling rule and gradual learning rate warmup, are able to increase the upper bound on feasible global batch size, but not completely remove it (Ben-Nun and Hoefler, 2018). Additionally, other large-batch methods such as LARS (You, Gitman and Ginsburg, 2017), have been able to push the upper bounds of feasible global batch size for the ILSVRC 2012 dataset even furhter.

4 RESEARCH FRAMEWORK

In chapter 1, we established that our research is based on reproduceable experiments. We also established, that our research experiments are based on the One-factor-at-a-time (OFAT) method, where we adjust a single input parameter, and measure or observe its effects on the output. In this chapter, we first establish the execution environment, in which our research experiments will be conducted. Then, after establishing the execution environment, we take a closer look at our research experiments.

4.1 Taito cluster environment

All of our research experiments will be conducted on the CSC's Taito computing clusters GPU partition (Using Taito-GPU, 2018). The Taito computing cluster includes a separate partition of compute nodes with dedicated GPU accelerator cards. The Taito clusters GPU partition contains 26 nodes equipped with 4 Pascal P100 GPU cards each and 12 nodes with 2 dual GPU K80 GPU cards (Taito-GPU hardware, 2018). The Pascal P100 nodes consists of 26 Dell PowerEdge C4130 servers with:

- 2 x Xeon E5-2680 v4 CPUs with 14 cores each running at 2.4GHz
- 512 GB of DDR4 memory
- 4 x Pascal P100 GPUs connected in pairs to each CPU
- 2 x 800 GB of SATA SSD scratch space
- FDR InfiniBand connection to other nodes in the cluster

Our experiments will utilize the Taito clusters compute nodes that are equipped with Pascal P100 GPU cards.

The Taito cluster utilizes cluster management and job scheduling system called slurm workload manager (Slurm Workload Manager, 2013). End users can submit their computing jobs using the slurm *sbatch* job submission utility (sbatch, 2019) and the scheduling system automatically takes care of fair and optimal scheduling of user submitted jobs . If there are not enough free computing resources available on the time of submission, the submitted job will be placed on the job queue. The jobs on the job queue are executed as resources become available based on the jobs queuing priority. Our experi-

ments will be run utilizing the slum workload manager and the sbatch job submission utility.

The experiments will utilize the following software versions:

- Python version: Python 3.6.3 :: Intel Corporation
- TensorFlow version: 1.12.0
- TensorFlow cnn benchmarks git branch: cnn_tf_v1.12_compatible
- PyTorch version: 1.0.0
- Horovod version: 0.15.2
- mpirun version: (Open MPI) 2.1.2
- CUDA version: 9.0
- CuDNN version: 7.4.1
- NCCL version: 2.3.7

4.2 Research experiments

In this chapter, we describe our research experiments. We start off by establishing the deep neural network architecture and the dataset that we utilize in our experiments, and then move on to describing the experiments in more detail.

According to Akiba, Suzuki and Fukuda, the task of training on ResNet-50 architecture (He *et al.*, 2015) for 90 epochs with the ILSVRC 2012 dataset (Russakovsky *et al.*, 2015) is an extensively used benchmark, when evaluating performance of distributed deep learning (Akiba, Fukuda and Suzuki, 2017). Thus, for all of our experiments outlined in this chapter, we will utilize the ResNet-50 deep convolutional neural network architecture and the ILSVRC 2012 dataset. When evaluating model quality, we note that the baseline top-1 validation accuracy for ResNet-50 is 75.3% (Prace, 2019). We also note, that the experiments outlined in this chapter, will be conducted in single-precision (FP32), and that they will utilize the synchronous SGD optimizer.

In experiments 1 to 3 we measure throughput. The throughput metric is then used to establish the distributed programs scaling performance in the provided execution environment.

In experiments 4 to 9 we inspect model quality. When scaling out in deep learning, common model quality issues include degraded training and validation accuracy and model generalization issues (Keskar *et al.*, 2016; Ben-Nun and Hoefler, 2018).

Experiment 1

In the initial experiment we establish a baseline in the form of processed images per second (throughput). The baseline will be established by training with ILSVRC 2012 data on ResNet-50 architecture, for single epoch, on single cluster node, while utilizing a single Pascal P100 GPU. The experiment will be conducted utilizing the following deep learning frameworks: Horovod distributed TensorFlow, PyTorch and Horovod distributed PyTorch. When applicable, the initial experiment will be conducted using the different per GPU worker batch sizes of 32, 64, 128.

Experiment 2

In the second experiment we continue to measure throughput; This time in the local data parallel computing setting. Throughput will be measured while training with ILSVRC 2012 data on ResNet-50 architecture, for single epoch, on single cluster node, while utilizing two and four Pascal P100 GPUs. The second experiment will be conducted utilizing the same deep learning frameworks and per GPU worker batch sizes as experiment 1. If the given deep learning framework supports multiple communications backends (e.g. MPI, NCCL) the second experiment will be conducted using the communications backend the frameworks authors recommends for local data parallel training. Ideally, when compared to the 1 GPU baseline that was established in experiment 1 the throughput should be doubled when utilizing two Pascal P100 GPUs and quadrupled when utilizing four Pascal P100 GPUs.

Experiment 3

In the third experiment we measure throughput in the distributed data parallel computing setting. Throughput will be measured while training with ILSVRC 2012 data on ResNet-50 architecture, for single epoch, on multiple cluster nodes, while utilizing 8, 16, 24 and 32 Pascal P100 GPUs. Again, the third experiment will be conducted utilizing the same deep learning frameworks and per GPU worker batch sizes as experiment 1. If the deep learning framework supports multiple communication backends, the third

experiment will be conducted using the communications backend (e.g. MPI, NCCL) the frameworks authors recommend for distributed data parallel training. Ideally, when compared to the 1 GPU baseline that was established in experiment 1 the throughput should be sixteen times greater when training with sixteen Pascal P100 GPUs and thirty-two times greater when training with thirty-two Pascal P100 GPUs.

Experiment 4

While the throughput metric recorded in experiments 1 to 3 can be utilized to measure the distributed programs scaling performance in the provided execution environment, and to deduce a rough estimate of the time (wall clock) required for the algorithm to process the whole training dataset, it does not tell us anything about the resulting models quality.

In the fourth experiment we measure the models quality, when scaling out training naïvely without adjusting any of the networks hyperparameters. The per worker batch size will be set to the maximum value that can fit in a single Pascal P100 GPUs memory to ramp-up the global batch size. The resulting models accuracy will be measured, while training with ILSVRC 2012 data on ResNet-50 architecture for 90 epochs, while utilizing 1, 2, 16, 24 and 32 Pascal P100 GPUs.

Experiment 5

In the fifth experiment we again measure the models quality. This time we apply linear learning rate scaling as described by Goyal et al., while omitting the learning rate warmup period (Goyal et al., 2017). The per worker batch size will be set to the maximum value that can fit in a single Pascal P100 GPUs memory to ramp-up the global batch size and speed up the training process. The resulting models accuracy and the training time (wall-clock) will be measured, while training with ILSVRC 2012 data on ResNet-50 architecture for 90 epochs, while utilizing 1 and 32 Pascal P100 GPUs.

Experiment 6

In the sixth experiment we again measure the models quality. Here we apply linear learning rate scaling (Goyal et al., 2017) combined with a 5 epoch stepping learning rate warmup period. Again, the per worker batch size will be set to the maximum value that

can fit in a single Pascal P100 GPUs memory to ramp-up the global batch size and speed up the training process. The resulting models accuracy and the training time (wall-clock) will be measured, while training with ILSVRC 2012 data on ResNet-50 architecture for 90 epochs, while utilizing 8, 16, 24 and 32 Pascal P100 GPUs.

Experiment 7

In the seventh experiment we continue measure the models quality. Here we apply linear learning rate scaling (Goyal et al., 2017) combined with a 5 epoch stepping learning rate warmup period. This time, the per GPU worker batch size will be set to 32 to reduce the global batch size. The resulting models accuracy and the training time (wall-clock) will be measured, while training with ILSVRC 2012 data on ResNet-50 architecture for 90 epochs, while utilizing 32 Pascal P100 GPUs.

Experiments 8 and 9

Experiments eight and nine will involve the comparison of different warmup schemes and their effect on the resulting models quality. In experiment eight and nine we apply linear learning rate scaling combined with 5 and 10 epoch gradual learning rate warmup periods (Goyal et al., 2017). Again, the per worker batch size will be set to the maximum value that can fit in a single Pascal P100 GPUs memory to ramp-up the global batch size and speed up the training process. The resulting models accuracies and the training times (wall-clock) will be measured, while training with ILSVRC 2012 data on ResNet-50 architecture for 90 epochs, while utilizing 24 and 32 Pascal P100 GPUs.

5 RESULTS

In this chapter, we review the results of our experiments. In experiments 1 to 3, we measured throughput and scaling performance of different deep learning frameworks, when trained with different per GPU worker batch sizes, in the provided execution environment. In experiment 4, we inspected the quality of the resulting models, when training was scaled out utilizing a naïve method. In 5 experiment, we scaled out training while utilizing linear learning rate scaling, and inspected how it affected the resulting models quality (Goyal *et al.*, 2017). In experiments 6 to 9, we inspected the quality of

the resulting models, when training was scaled out utilizing the linear scaling and learning rate warmup methods (Goyal *et al.*, 2017).

5.1 Experiment 1 – Throughput – Single GPU baseline

The first experiment consisted of establishing a baseline in form of processed images per second (throughput) while training a ResNet-50 architecture deep convolutional neural network with the ILSVRC 2012 dataset for the task of 1000 class image classification. Table 1 lists the results of Horovod distributed TensorFlow for per worker batch sizes 32, 64 and 128. Tables 2 and 3 list the results for PyTorch and Horovod distributed PyTorch for a per worker batch size of 128.

Samples of the slurm batch scripts that were used for applying resources and launching the experiments can be seen in Appendix 1.

The Horovod distributed TensorFlow baseline was established utilizing the tf_cnn_benchmarks script (tf_cnn_benchmarks, 2018). To improve data loading time, the ILSVRC 2012 dataset, which had been preprocessed by converting the image data to TFRecord format (TFRecord, 2017), was copied to the compute nodes local SSD drive before starting training. The tf_cnn_benchmarks script was configured to train the network on ResNet-50 architecture and to process the first 200 training batches before stopping and reporting throughput.

The PyTorch baseline was established utilizing the PyTorch examples projects script for training with ImageNet data (PyTorch Examples ImageNet, 2019). The ILSVRC 2012 dataset, which had been preprocessed by resizing the training samples to a smaller dimension of 250x250 pixels (PyTorch ImageNet resize, 2015) and copied into appropriate training and validation subfolders by utilizing the valprep script (PyTorch ImageNet valprep, 2015), was copied to the compute nodes local SSD drive before starting training. The PyTorch Examples ImageNet training script was configured to train the network for 1 epoch on ResNet-50 architecture. Once the script had finished training, the training scripts logs were inspected. An example of the PyTorch Examples ImageNet training scripts log output can be seen in Figure 11.

0: Epoch: [0][197/10010]	Time 0.581 (0.633)	Data 0.273 (0.283) …
0: Epoch: [0][198/10010]	Time 0.581 (0.633)	Data 0.273 (0.283) …
0: Epoch: [0][199/10010]	Time 0.582 (0.633)	Data 0.273 (0.283) …
0: Epoch: [0][200/10010]	Time 0.581 (0.632)	Data 0.273 (0.283) …
0: Epoch: [0][201/10010]	Time 0.581 (0.632)	Data 0.273 (0.283) …

Figure 11. Training log output of PyTorch Examples ImageNet script.

The PyTorch Examples ImageNet training script reports the average batch processing time enclosed in parenthesis after the processing time for the current batch of data. For example, in the data given in Figure 11 above, the average processing time during training batch [199/10010] is reported as 0.633. Thus, the average throughput for the first 200 training batches was then calculated using the formula:

$$throughput = \frac{(per\ GPU\ batch\ size * number\ of\ GPUs\ used\ during\ training)}{average\ processing\ time\ reported\ during\ training\ batch\ 199}$$

The Horovod distributed PyTorch baseline utilized a modified version of the training script that was used to establish the PyTorch baseline (Horovod PyTorch Examples ImageNet, 2019). The Horovod frameworks FP16 compression features (Horovod FP16, 2018), were not utilized to allow for fairer comparison of PyTorch results with Horovod distributed PyTorch results.

The Horovod distributed PyTorch baseline utilizes the same preprocessed training and validation data that was used when establishing the PyTorch baseline. The training data was again copied to the compute nodes local SSD drive before starting training. The Horovod distributed PyTorch training script was configured to train the network for 1 epoch on ResNet-50 architecture. Once the script had finished training, the training scripts logs were inspected.

The Horovod distributed PyTorch script and PyTorch Examples ImageNet training script both utilize the same training log format, so the throughput for Horovod distributed PyTorch was calculated using the same formula that was used when calculating the throughput for PyTorch.

When inspecting the results in Table 1, we notice that the throughput increases when we increase the per worker batch size. We achieve more efficient resource utilization when we increase the per worker batch size, which in turn leads to increased throughput.

Table 1. Single-node single-GPU throughput baseline, Horovod+TensorFlow per worker batch sizes 32, 64 and 128.

Deep learning framework	Batch size per GPU	Number of GPUs	Throughput (images per second)
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	32	1	210.15
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	64	1	234.01
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	128	1	249.28

When inspecting the results in Tables 2 and 3, we notice that the Horovod distributed version of the PyTorch Examples ImageNet training script performs equally well as the PyTorch version of the same training script. This is expected, since no parallelization or distribution of computation is performed when we are training with one single GPU worker.

Table 2. Single-node single-GPU throughput baseline, PyTorch per worker batch size 128.

Deep learning framework	Batch size per GPU	Number of GPUs	Throughput (images per second)
PyTorch 1.0.0	128	1	207.11

Table 3. Single-node single-GPU throughput baseline, Horovod+PyTorch per worker batch size 128.

Deep learning framework	Batch size per GPU	Number of GPUs	Throughput (images per second)
Horovod 0.15.2 + PyTorch 1.0.0	128	1	207.79

We note that even though Horovod distributed TensorFlow displayed better throughput than PyTorch and Horovod distributed PyTorch in our experiment, that these result should not be interpreted as the latter frameworks being inferior when compared to the former. We suggest that the Horovod distributed TensorFlow results should be utilized for measuring the frameworks scaling efficiency in the provided execution environment. We find that the PyTorch and Horovod distributed PyTorch results make for a fairer framework comparison, since both frameworks utilized a similar training script during the experiment, where the Horovod distributed TensorFlow utilized the benchmarking framework (tf_cnn_benchmarks) that is optimized for speed.

5.2 Experiment 2 – Throughput – Local data parallel

The second experiment consisted of parallelizing training on multiple GPU workers within a single cluster node (local data parallel). We again measured the throughput in the form of processed images per second while utilizing two and four Pascal P100 GPUs, while training a ResNet-50 architecture deep convolutional neural network with the ILSVRC 2012 dataset for the task of 1000 class image classification. Table 4 list the results of Horovod distributed TensorFlow for per worker batch sizes 32, 64 and 128. Tables 5 and 6 list the results for PyTorch and Horovod distributed PyTorch for a per worker batch size of 128.

Samples of the slurm batch scripts that were used for applying resources and launching the experiments can be seen in Appendix 2. We note that in experiment 2 we utilized the same training scripts and methods for calculating throughput that were used in experiment 1.

When inspecting the results in Table 4, we again notice that the throughput increases when we increase the per worker batch size. We also notice that throughput increases when we add more GPU workers. We note that the gain in throughput is not linear when adding GPU workers.

Table 4. Single-node multi-GPU throughput, Horovod+TensorFlow per worker batch sizes 32, 64 and 128.

Deep learning framework	Batch size per GPU	Number of GPUs	Throughput (images per second)
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	32	2	375.41
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	32	4	769.49
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	64	2	424.27
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	64	4	856.30
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	128	2	466.31
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	128	4	953.63

When inspecting the results in Tables 5 and 6, we notice that the Horovod distributed version of the PyTorch Examples ImageNet training script is able to process more images per second when compared to the PyTorch version of the same training script.

Table 5. Single-node multi-GPU throughput, PyTorch per worker batch size 128.

Deep learning framework	Batch size per GPU	Number of GPUs	Throughput (images per second)
PyTorch 1.0.0	128	2	372.09
PyTorch 1.0.0	128	4	752.94

Table 6. Single-node multi-GPU throughput, Horovod+PyTorch per worker batch size 128.

Deep learning framework	Batch size per GPU	Number of GPUs	Throughput (images per second)
Horovod 0.15.2 + PyTorch 1.0.0	128	2	390.24
Horovod 0.15.2 + PyTorch 1.0.0	128	4	807.57

As we inspect the training logs for the PyTorch Examples ImageNet training script and the Horovod distributed version of the PyTorch Examples ImageNet training script in Figure 12, we notice that the first batch of training data in the beginning of each epoch takes a longer time to process. The PyTorch Examples ImageNet training script spends more time processing this first batch of training data than the Horovod distributed version of the PyTorch Examples ImageNet training script. This increased data loading time during the initial batch of training data attributes to the PyTorch Examples ImageNet training scripts reduced throughput metric.

Training log for PyTorch 1 node 2 GPUs
Epoch: [0][0/5005] Time 18.532 (18.532) Data 12.142 (12.142) …

Training log for Horovod + PyTorch 1 node 2 GPUs
Epoch: [0][0/5005] Time 9.696 (9.696) Data 2.759 (2.759) …

Figure 12. Training log output. Increased data loading time in local data parallel setting.

We again note that even though Horovod distributed TensorFlow displayed better throughput than PyTorch and Horovod distributed PyTorch in our experiment, that these result should not be interpreted as the latter frameworks being inferior when compared to the former. We again suggest that the Horovod distributed TensorFlow results should be utilized for measuring the frameworks scaling efficiency in the provided execution environment. We find that the PyTorch and Horovod distributed PyTorch results make for a fairer framework comparison, since both frameworks utilized a similar training script during the experiment, where the Horovod distributed TensorFlow utilized a benchmarking framework (tf_cnn_benchmarks) that is optimized for speed.

5.3 Experiment 3 – Throughput – Distributed data parallel

The third experiment consisted of distributing and parallelizing training on multiple cluster nodes and multiple GPU workers within the cluster nodes (distributed data parallel). We again measured the networks throughput in the form of processed images per second while utilizing 8, 16, 24 and 32 GPU workers while training a ResNet-50 architecture deep convolutional neural network with the ILSVRC 2012 dataset. Table 7 lists the results of Horovod distributed TensorFlow for per worker batch sizes 32, 64 and 128. Tables 8 and 9 list the results for PyTorch and Horovod distributed PyTorch for a per worker batch size of 128.

Samples of the slurm batch scripts that were used for applying resources and launching the experiments can be seen in Appendix 3. In experiment 3 we utilized the same training scripts and methods for calculating throughput that were used in experiment 1.

Table 7. Multi-node multi-GPU throughput, Horovod+TensorFlow per worker batch sizes 32, 64 and 128.

Deep learning framework	Batch size per GPU	Number of nodes	Number of GPUs	Throughput (images per second)
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	32	2	8 (2 * 4)	1438.07
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	32	4	16 (4 * 4)	2812.84
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	32	6	24 (6 * 4)	3809.91
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	32	8	32 (8 * 4)	4767.74
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	64	2	8 (2 * 4)	1683.31
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	64	4	16 (4 * 4)	3301.11
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	64	6	24 (6 * 4)	4832.52
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	64	8	32 (8 * 4)	6210.30
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	128	2	8 (2 * 4)	1855.84
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	128	4	16 (4 * 4)	3646.30
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	128	6	24 (6 * 4)	5439.04
Horovod 0.15.2 + TensorFlow 1.12.0 + cnn_tf_v1.12_compatible	128	8	32 (8 * 4)	6643.48

When inspecting the results in Table 7, we notice that Horovod+TensorFlow per worker batch size 32 does not scale as well as per worker batch sizes 64 and 128.

When inspecting the results in Tables 8 and 9, we notice that the Horovod distributed version of the PyTorch Examples ImageNet training script is again able to process more images per second when compared to the PyTorch version of the same training script.

Table 8. Multi-node multi-GPU throughput, PyTorch per worker batch size 128.

Deep learning framework	Batch size per GPU	Number of nodes	Number of GPUs	Throughput (images per second)
PyTorch 1.0.0	128	2	8 (2 * 4)	1418.28
PyTorch 1.0.0	128	4	16 (4 * 4)	2828.72
PyTorch 1.0.0	128	6	24 (6 * 4)	4243.09
PyTorch 1.0.0	128	8	32 (8 * 4)	5696.80

Table 9. Multi-node multi-GPU throughput, Horovod+PyTorch per worker batch size 128.

Deep learning framework	Batch size per GPU	Number of nodes	Number of GPUs	Throughput (images per second)
Horovod 0.15.2 + PyTorch 1.0.0	128	2	8 (2 * 4)	1551.51
Horovod 0.15.2 + PyTorch 1.0.0	128	4	16 (4 * 4)	3150.76
Horovod 0.15.2 + PyTorch 1.0.0	128	6	24 (6 * 4)	4711.65
Horovod 0.15.2 + PyTorch 1.0.0	128	8	32 (8 * 4)	6104.32

As we inspect the training logs for the PyTorch Examples ImageNet training script and the Horovod distributed version of the PyTorch Examples ImageNet training script in

Figure 13, we notice that the first batch of training data in the beginning of each epoch again takes a longer time to process. We again note that this increased data loading time during the initial batch of training data attributes to the PyTorch Examples ImageNet training scripts reduced throughput metric.

PyTorch 8 nodes 32 GPUs
0: Epoch: [0][0/313] Time 21.925 (21.925) Data 11.993 (11.993) …

Horovod + PyTorch 8 nodes 32 GPUs
Epoch: [0][0/313] Time 9.950 (9.950) Data 1.085 (1.085) …

Figure 13. Training log output. Increased data loading time in distributed data parallel setting.

When inspecting the scaling performance of Horovod distributed TensorFlow for per worker batch sizes 32, 64 and 128 in Figures 14, 15 and 16, we notice that we are only able to achieve a 22.7x speedup when utilizing 32 GPU workers when running with per worker batch size 32. Per worker batch sizes 64 and 128 both display better scaling performance and achieve a 26.5x and 26.7 speedup when utilizing 32 GPU workers.

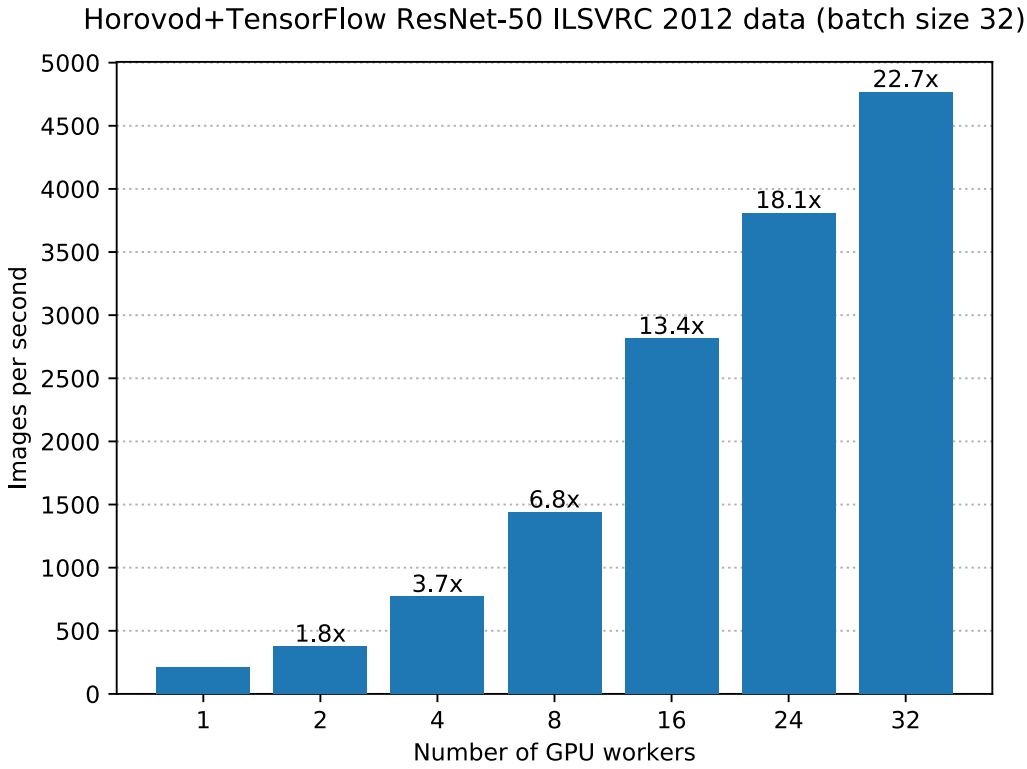

Figure 14. Scaling performance of Horovod+TensorFlow with per GPU worker batch size 32.

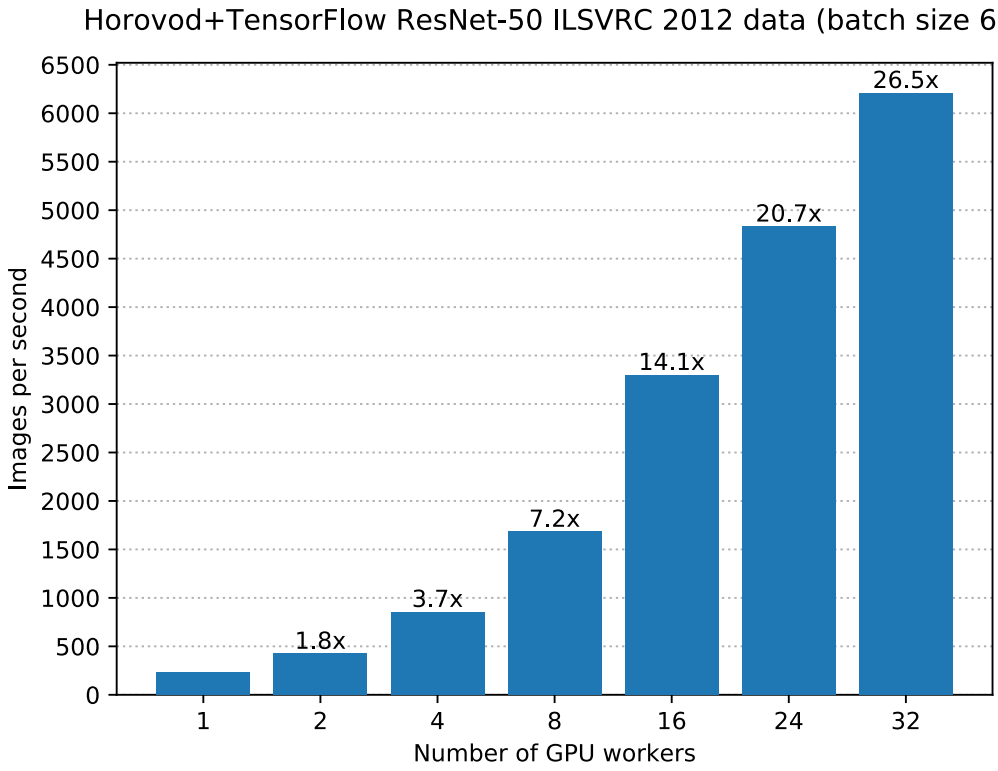

Figure 15. Scaling performance of Horovod+TensorFlow with per GPU worker batch size 64.

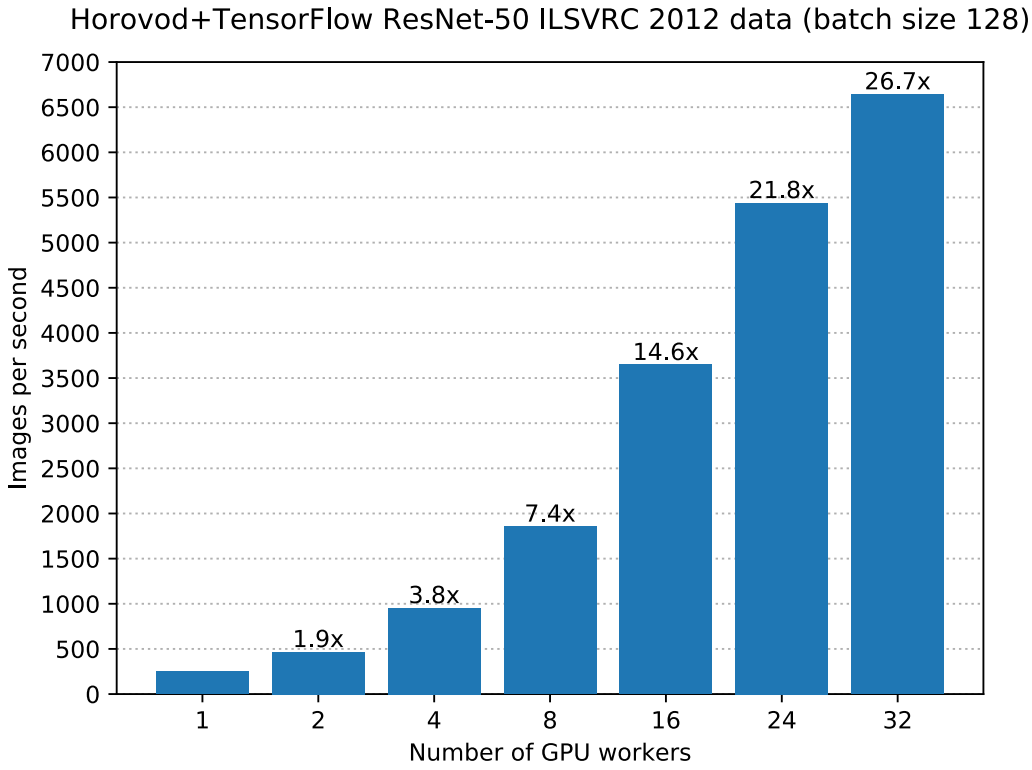

Figure 16. Scaling performance of Horovod+TensorFlow with per GPU worker batch size 128.

When we inspect the scaling performance comparison of PyTorch and Horovod distributed PyTorch in Figure 17, we notice that the Horovod distributed version of the training script is able to scale out more efficiently and achieves a 29.4x speedup when running on 32 GPU workers. With the PyTorch version of the training script we are only able to achieve a speedup of 27.5x when running on 32 GPU workers.

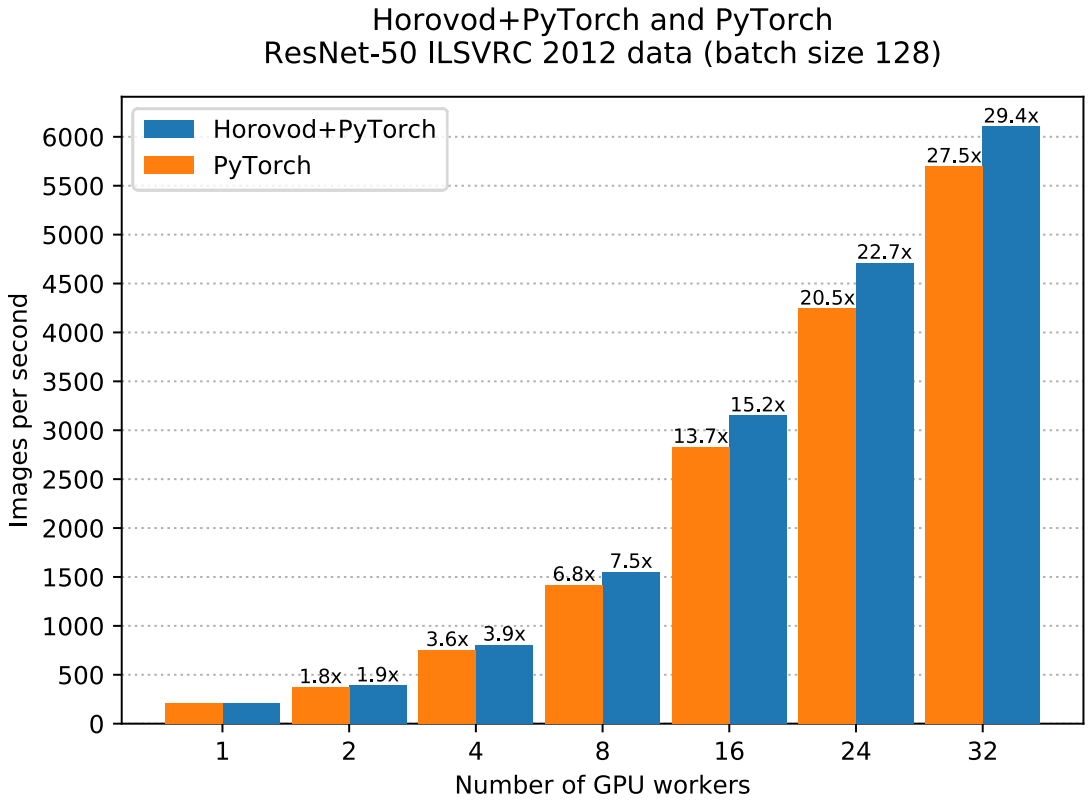

Figure 17. Scaling performance comparison of PyTorch and Horovod+PyTorch.

As with the two previous experiments, we again note that even though Horovod distributed TensorFlow displayed better throughput than PyTorch and Horovod distributed PyTorch in our experiment. Still these results should not be interpreted as the latter frameworks being inferior when compared to the former. We again suggest that the Horovod distributed TensorFlow results should be utilized for measuring the frameworks scaling efficiency in the provided execution environment. We find that the PyTorch and Horovod distributed PyTorch results make for a fairer framework comparison, since both frameworks utilized a similar training script during the experiment, where the Horovod distributed TensorFlow utilized a benchmarking framework (tf_cnn_benchmarks) that is optimized for speed.

5.4 Experiment 4 – Model quality – Naïve scaling method

In experiment 4 we trained multiple ResNet-50 architecture deep convolutional neural networks with the ILSVRC 2012 dataset for the task of 1000 class image classification. Here we applied the naïve method of gradually increasing the GPU worker count, without adjusting any of the networks hyperparameters when adding more GPU workers. The per worker batch size was set to the maximum value that could fit in a single Pascal P100 GPUs memory to ramp-up the global batch size and speed up the training process.

The PyTorch Examples ImageNet training script was used for training the network on ResNet-50 architecture for 90 epochs (PyTorch Examples ImageNet, 2019). Apart from the per worker batch size, all other hyperparameter were left unconfigured. The unconfigured hyperparameters thus utilized the PyTorch Examples ImageNet training scripts default values. The best Top-1/Top-5 validation accuracies for the various GPU worker counts were recorded and the results of experiment 4 can be seen in Table 11.

Samples of the slurm batch scripts that were used for applying resources and launching the experiments can be seen in Appendix 4. The hyperparameters, data transforms, learning rate schedule and other relevant settings of the experiment can be seen in Appendix 5.

When we inspect the results in Table 11, we notice that the Top-1/Top-5 validation accuracy for 2 GPUs at global batch size 256 is slightly better than the result for a single GPU at batch size 128. This is expected, since we used the PyTorch Examples ImageNet training scripts default hyperparameters, which are fine-tuned for the scripts default batch size of 256. We also notice that the Top-1/Top-5 validation accuracy degrades as the global batch size increases. This is also expected; In a paper by Keskar et al. which studies the phenomena of degraded generalization ability of models trained with larger batch sizes, the authors present numerical evidence that large batch methods tend to converge to sharp minimizers of the training function (Keskar *et al.*, 2016). We note that increasing the learning rate for larger batch sizes might help the algorithm to escape local minimum.

Table 10. Validation accuracy, naïve scaling method.

Deep learning framework	Learning rate	Batch size per GPU	Global batch size	Number of GPUs	Best Top-1/Top-5 validation accuracy
PyTorch 1.0.0	0.1	128	128	1	73.286% / 91.558%
PyTorch 1.0.0	0.1	128	256	2	73.946% / 91.642%
PyTorch 1.0.0	0.1	128	2048	16 (4 * 4)	71.726% / 90.512%
PyTorch 1.0.0	0.1	128	3072	24 (6 * 4)	70.332% / 89.488%
PyTorch 1.0.0	0.1	128	4096	32 (8 * 4)	69.006% / 88.564%

When we inspect the plot of training and validation error in Figure 18, we notice that the training error curves and validation error curves for the different global batch sizes do not align.

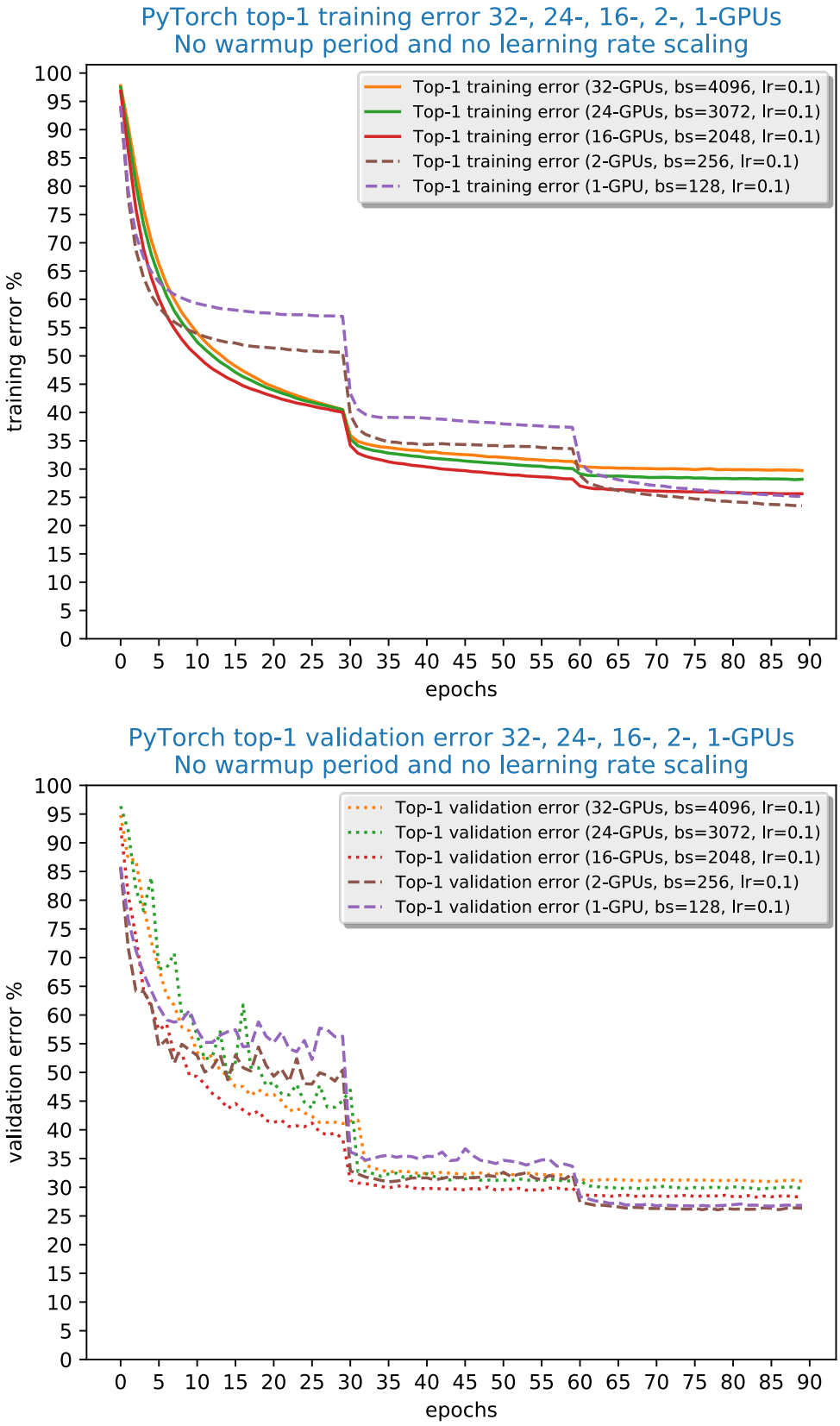

Figure 18. Plot of training and validation error. Naïve scaling method.

When we inspect the plots of training and validation error in Figure 19, we notice that there are potential generalization issues during the early stages of training for batch size 3072, which dissipate during the first scheduled drop in learning rate at epoch 30. Overall, the training and validation error seem to follow a similar pattern, which we interpret as the resulting models having good generalization capabilities.

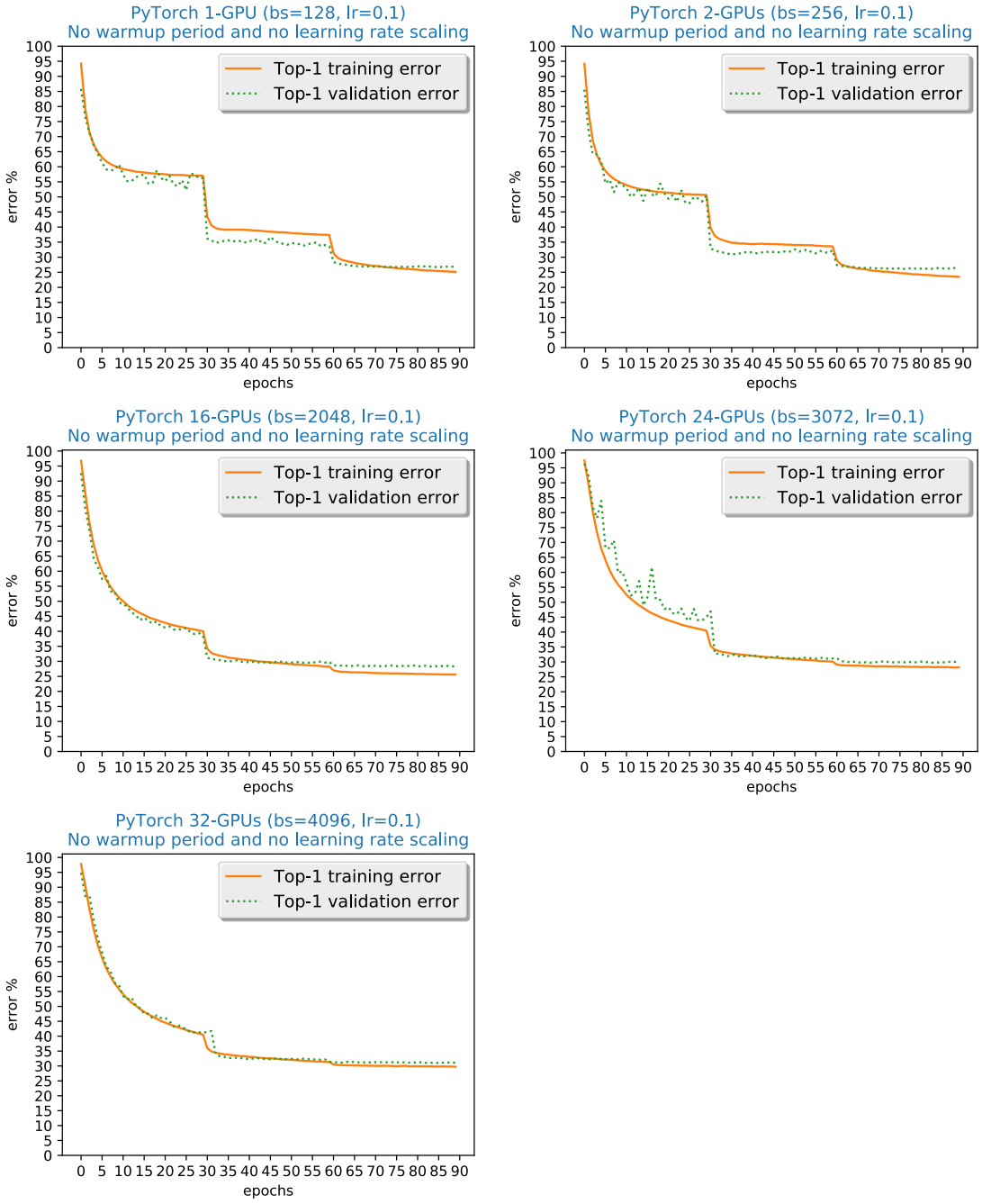

Figure 19. Grid plot of training and validation error. Naïve scaling method.

5.5 Experiment 5 – Model quality – Linear learning rate scaling

In experiment 5 we applied linear learning rate scaling as described by Goyal et al., while omitting the learning rate warmup period (Goyal *et al.*, 2017). We trained two ResNet-50 architecture deep convolutional neural networks with the ILSVRC 2012 dataset for the task of 1000 class image classification. The per GPU worker batch size was set to the maximum value that could fit in a single Pascal P100 GPUs memory to ramp-up the global batch size and to speed up the training process. The learning rate schedule was adjusted to follow the schedule outlined by Goyal et al. (Goyal *et al.*, 2017). As with the previous experiment, all other hyperparameters, apart from the per GPU worker batch size and linearly scaled learning rate, were set to the PyTorch Examples ImageNet training scripts defaults.

The first network was trained using the PyTorch Examples ImageNet training script. The network was trained for 90 epochs while utilizing a single Pascal P100 GPU with a linearly scaled learning rate of 0.05 for the per GPU batch size of 128. We note that these single GPU results serve as a baseline for later experiments.

We also created a modified version of the PyTorch Examples ImageNet training script which utilizes the Horovod framework for local data parallel and distributed data parallel training (Horovod PyTorch Examples ImageNet, 2019). The second network was trained using the modified version of the training script which utilizes the Horovod framework. We trained the network for 90 epochs while utilizing 32 Pascal P100 GPUs and a linearly scaled learning rate of 1.6 for the global batch size of 4096. An example on how the learning rate was scaled can be seen in the following section.

The PyTorch Examples ImageNet training script utilizes a default learning rate of 0.1 for the scripts default batch size of 256. The training scripts other hyperparameter default values have been fine-tuned for the scripts default learning rate of 0.1 and default batch size of 256. Let us establish the learning rate of 0.1 for batch size 256 as the *base learning rate*. Then, by applying the following formula, we can scale up the base learning rate linearly to account for 32 GPU workers with the per worker batch size of 128.

$$linearly\ scaled\ learning\ rate = \frac{(32 * 128)}{256} * 0.1 = 1.6$$

The same formula can also be applied to scale down the base learning rate linearly to account for 1 GPU worker with the per worker batch size of 128.

$$linearly\ scaled\ learning\ rate = \frac{(1 * 128)}{256} * 0.1 = 0.05$$

When we scale the learning rate linearly according to the global batch size, we can keep all other hyperparameters fixed (Goyal *et al.*, 2017). We note that this greatly simplifies the scaling out process, since we only need to tune one hyperparameter (learning rate) when scaling out.

The slurm batch scripts that were used for applying resources and launching the experiments can be seen in Appendix 6. The hyperparameters, data transforms, learning rate schedule and other relevant settings of the experiment can be seen in Appendix 7.

As we inspect the results for experiment 5 in Table 12, we notice that linearly scaling down the learning rate to 0.05 and adjusting the learning rate scheduler to drop the learning rate by 1/10 at epochs 30, 60 and 80 helped us improve our best Top-1/Top-5 validation accuracy for our 1 GPU baseline. We also notice that the linearly scaled learning rate of 1.6 and the adjusted learning rate schedule helped us improve our best Top-1/Top-5 validation accuracy for 32 GPUs, when compared to the corresponding results in experiment 4, where we utilized a learning rate of 0.1. We notice that there is still a slight deviation in best Top-1/Top-5 validation accuracy when comparing the 1 GPU baseline results with the results for 32 GPU workers. We note that applying a learning rate warmup period as described by Goyal et al. might help reduce this deviation in best Top-1/Top-5 validation accuracy (Goyal *et al.*, 2017). A lower learning rate is required during the initial epochs of training, when network is changing rapidly, to address the optimization difficulty present during the initial epochs (Akiba, Suzuki and Fukuda, 2017; Goyal *et al.*, 2017).

Table 11. Linear learning rate scaling without warmup period. Validation accuracy and training time.

Deep learning framework	Linearly scaled learning rate	Batch size per GPU	Global batch size	Number of GPUs	Best Top-1/Top-5 validation accuracy	Wall clock time taken for training 90 epochs
PyTorch 1.0.0 (baseline)	0.05	128	128	1	75.028% / 92.296%	6 days 4 hours 16 minutes
Horovod 0.15.2 + PyTorch 1.0.0	1.6	128	4096	32 (8 * 4)	73.078% / 91.288%	0 days 6 hours 34 min

When we inspect the plot of training error in Figure 20, we notice that the curves for training error do not align. According to Goyal et al. the misalignment in training error curves can be interpreted as a an early proxy for decreased final validation accuracy (Goyal *et al.*, 2017).

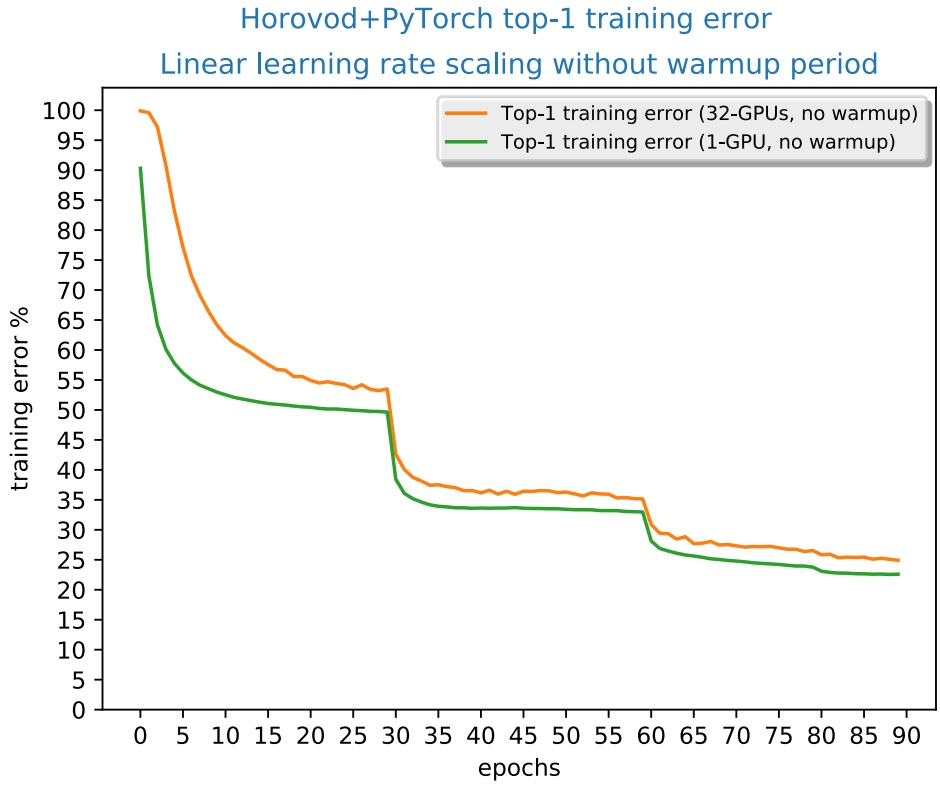

Figure 20. Plot of training error. 32 GPUs vs. 1 GPU baseline. Linear learning rate scaling without warmup period.

When inspecting the plot of training and validation error for 32 GPUs in Figure 21, we notice that there are potential generalization issues during the early stages of training, which dissipate during the first scheduled drop in learning rate at epoch 30. We note that these generalization issues were not present during experiment 4, where we trained the network with 32 GPU workers and a smaller learning rate of 0.1. Overall, after the first scheduled drop in learning rate, the training and validation error curves seem to follow a similar pattern, which we interpret as the resulting model having good generalization capabilities.

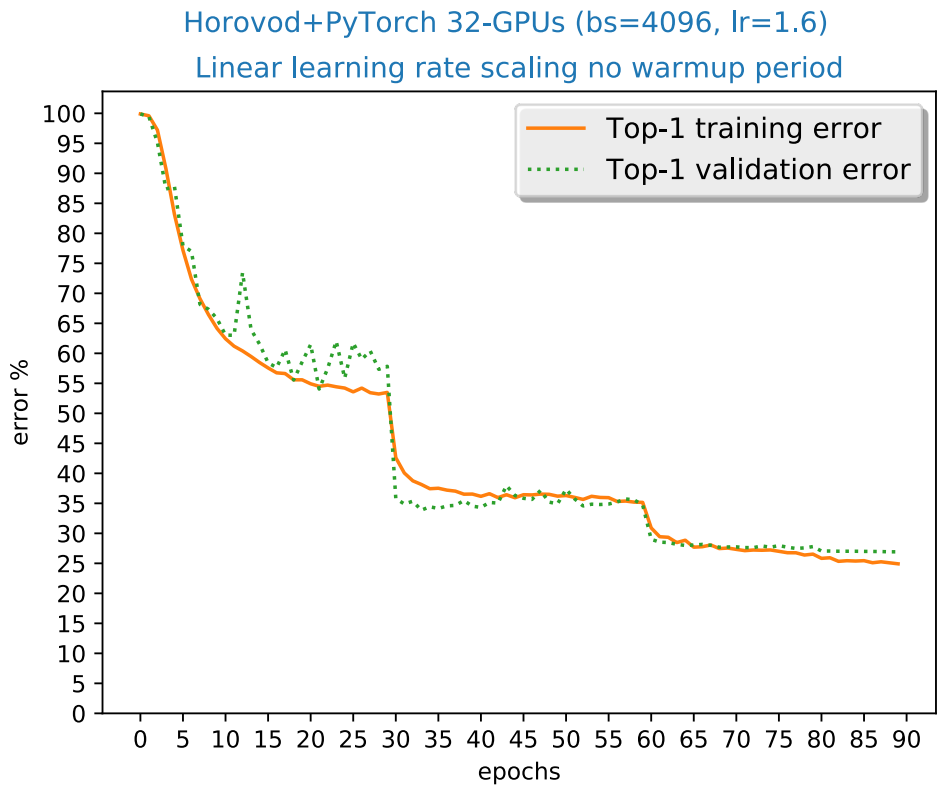

Figure 21. Plot of training and validation error. 32 GPUs linear learning rate scaling without warmup period.

5.6 Experiment 6 – Model quality – Linear learning rate scaling with 5 epoch stepping warmup period

In experiment 6 we applied linear learning rate scaling as described by Goyal et al. combined with a stepping learning rate warmup period (Goyal *et al.*, 2017). Here we trained multiple ResNet-50 architecture deep convolutional neural networks with the ILSVRC 2012 dataset for the task of 1000 class image classification. The networks were trained for 90 epochs with a different numbers of GPU workers. The per GPU worker batch size was set to the maximum value that could fit in a single Pascal P100 GPUs memory to ramp-up the global batch size and to speed up the training process. We applied the linear scaling rule to select the appropriate learning rates for the different worker counts (Goyal *et al.*, 2017). As with the previous experiment, all other hyperparameters were again left to their default values. We also applied a stepping learning rate warmup period of 5 epochs during the initial epochs of training. A pseudocode example for implementing the 5 epoch stepping learning rate warmup period can be seen in Appendix 6. We also took note of the wall clock time taken for the networks to train for 90 epochs.

Samples of the slurm batch scripts that were used for applying resources and launching the experiments can be seen in Appendixes 6 (baseline) and 8. The hyperparameters, data transforms, learning rate schedule and other relevant settings of the experiment can be seen in Appendixes 7 (baseline) and 9.

When we inspect the results for experiment 6 in Table 13, we notice that as the global batch size increases, the best Top-1/Top-5 validation accuracy no longer degrades in the same magnitude as it did with the naïve method which we utilized during experiment 4. When we compare the results of experiment 6 with the results of experiment 5, we notice that we were able to improve our best Top-1/Top-5 validation accuracy by applying a learning rate warmup period during the early stages of training. We also notice that the PyTorch Examples ImageNet training script that utilizes Horovod for distribution is able to finish 90 epochs of training faster than the PyTorch version of the training script. We regard this difference in 90 epoch training time somewhat consistent with the difference in the two frameworks throughput metric that was recorded for PyTorch and

Horovod distributed PyTorch in experiments 1-3. We also note that the wall clock training time for Horovod distributed PyTorch is very similar to the wall clock training time recorded in experiment 5. We regard this similarity in training time as the execution environment being stable.

Table 12. Linear learning rate scaling with 5 epoch stepping warmup. Validation accuracy and training time.

Deep learning framework	Linearly scaled learning rate	Batch size per GPU	Global batch size	Number of GPUs	Best Top-1/Top-5 validation accuracy	Wall clock time taken for training 90 epochs
PyTorch 1.0.0 (baseline)	0.05	128	128	1	75.028% / 92.296%	6 days 4 hours 16 minutes
Horovod 0.15.2 + PyTorch 1.0.0	0.4	128	1024	8 (2 * 4)	74.674% / 92.108%	0 days 20 hours 35 minutes
Horovod 0.15.2 + PyTorch 1.0.0	0.8	128	2048	16 (4 *4)	74.768% / 92.180%	0 days 11 hours 16 minutes
Horovod 0.15.2 + PyTorch 1.0.0	1.2	128	3072	24 (6 * 4)	74.468% / 91.880%	0 days 8 hours 7 minutes
Horovod 0.15.2 + PyTorch 1.0.0	1.6	128	4096	32 (8 * 4)	74.662% / 92.090%	0 days 6 hours 36 minutes
PyTorch 1.0.0	1.6	128	4096	32 (8 * 4)	74.604% / 92.186%	0 days 7 hours 18 minutes

As we inspect the plots of GPU CPU utilization in Figures 22 and 23, we notice that a pattern is present in the plot for the PyTorch Examples ImageNet training scripts GPU CPU utilization. In Figure 22 the first longer utilization peak in the plot represents the training epoch and the second shorter peak represents the validation epoch. We note that the same pattern is present on all the 32 GPUs. The Horovod distributed version of the PyTorch Examples ImageNet training scripts GPU CPU utilization plot in Figure 23 does not have the same pattern. The pattern in Figure 22 is consistent with the PyTorch Examples ImageNet training scripts longer data loading time observed in Experiment 3 Figure 13.

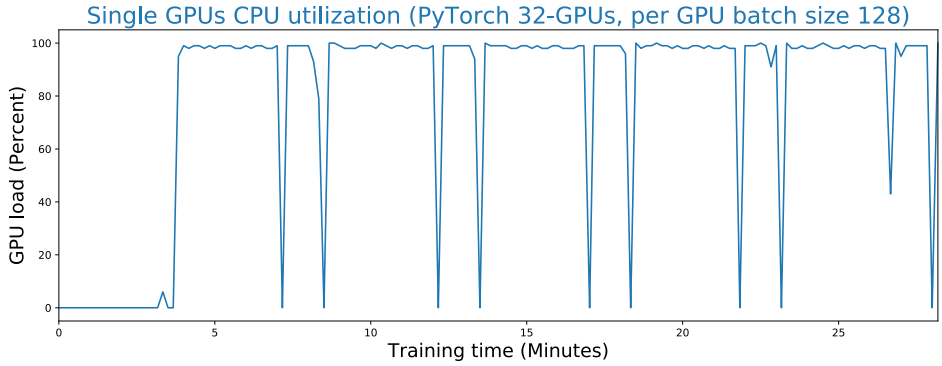

Figure 22. GPU CPU utilization on a single GPU, PyTorch 32 GPUs batch size 128.

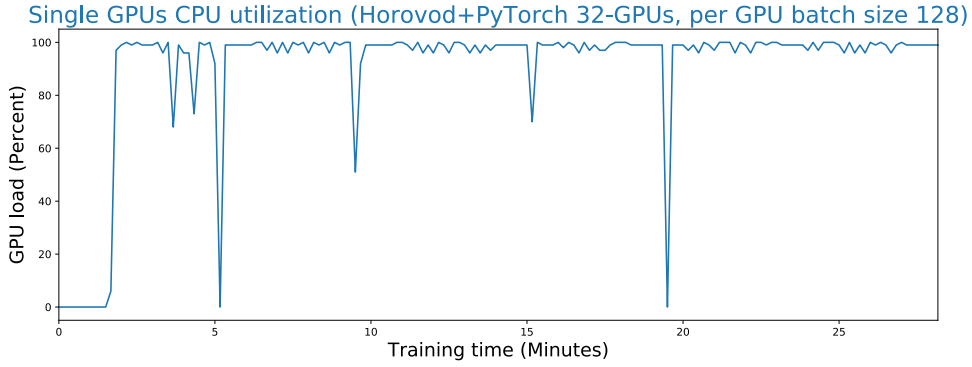

Figure 23. GPU CPU utilization on a single GPU, Horovod+PyTorch 32 GPUs batch size 128.

As we inspect the plot of training error in Figure 24, we notice that the training error curves for the different global batch sizes align, when we utilize a learning rate warmup period. According to Goyal et al. the aligning training curves for the different global batch sizes can be used as a reliable proxy for success well before training finishes, when trying out new settings (Goyal *et al.*, 2017). When we inspect the plot of validation error in the same figure, we notice that even though the training error curves for different global batch sizes do align, the validation error curves do not initially align. The misalignment in validation error dissipates during the scheduled drops in learning rate at epochs 30, 60 and 80. We interpret this as follows: The resulting models for the different global batch sizes, which utilize linear learning rate scaling and a 5 epoch stepping learning rate warmup period, have similar generalization capabilities, when the models are allowed to train for a sufficient amount of epochs.

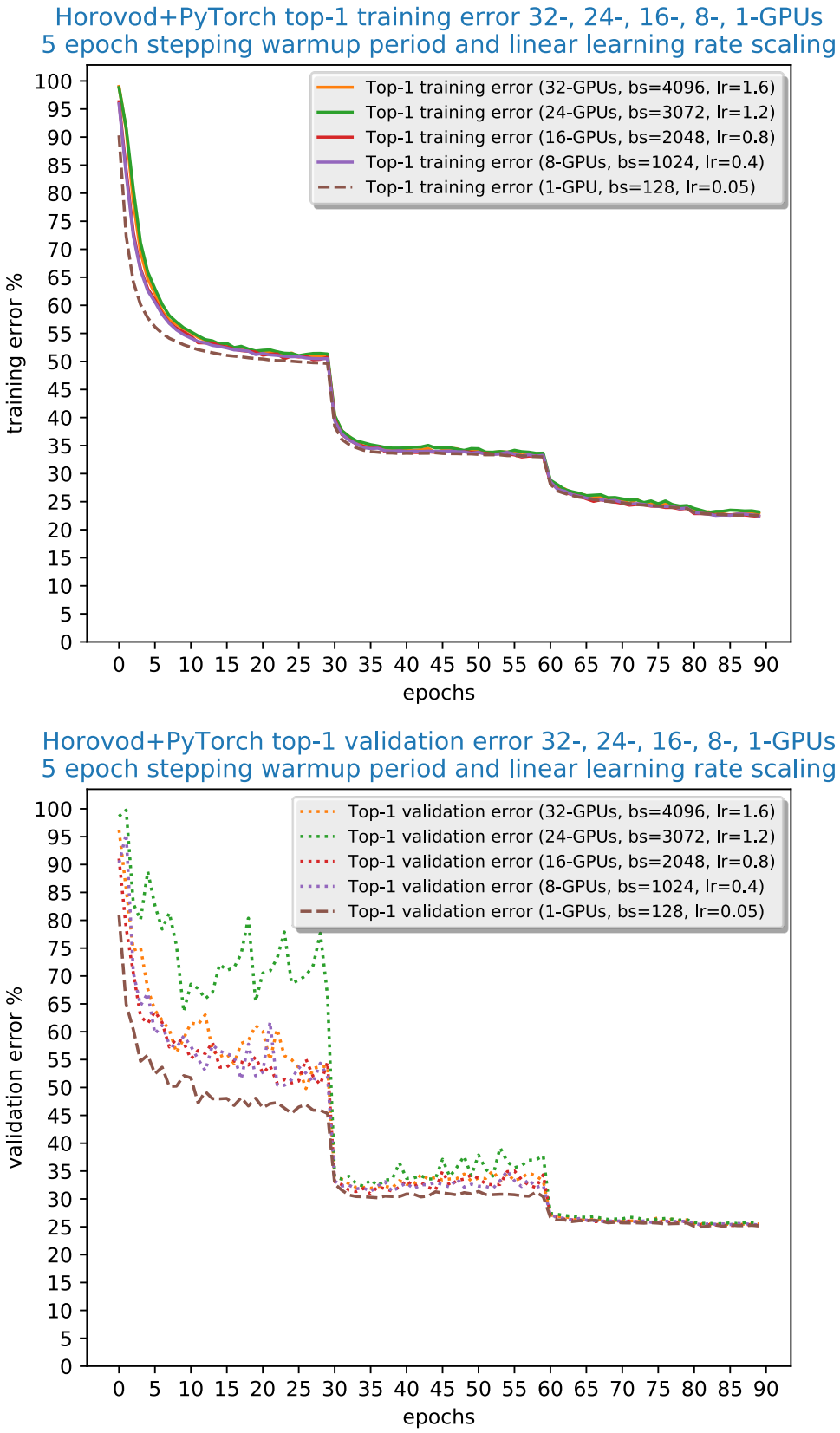

Figure 24. Plot of training and validation error. Linear learning rate scaling + 5 epoch stepping warmup

As we inspect the plot of training and validation error in Figure 25, we notice that the training and validation error curves do not align in similar fashion as they did when the models were trained utilizing the naïve method experiment 4, where we trained with a smaller learning rate of 0.1. We also notice that there are again potential generalization issues during the early stages of training for batch size 3072, which dissipate during the first scheduled drop in learning rate at epoch 30.

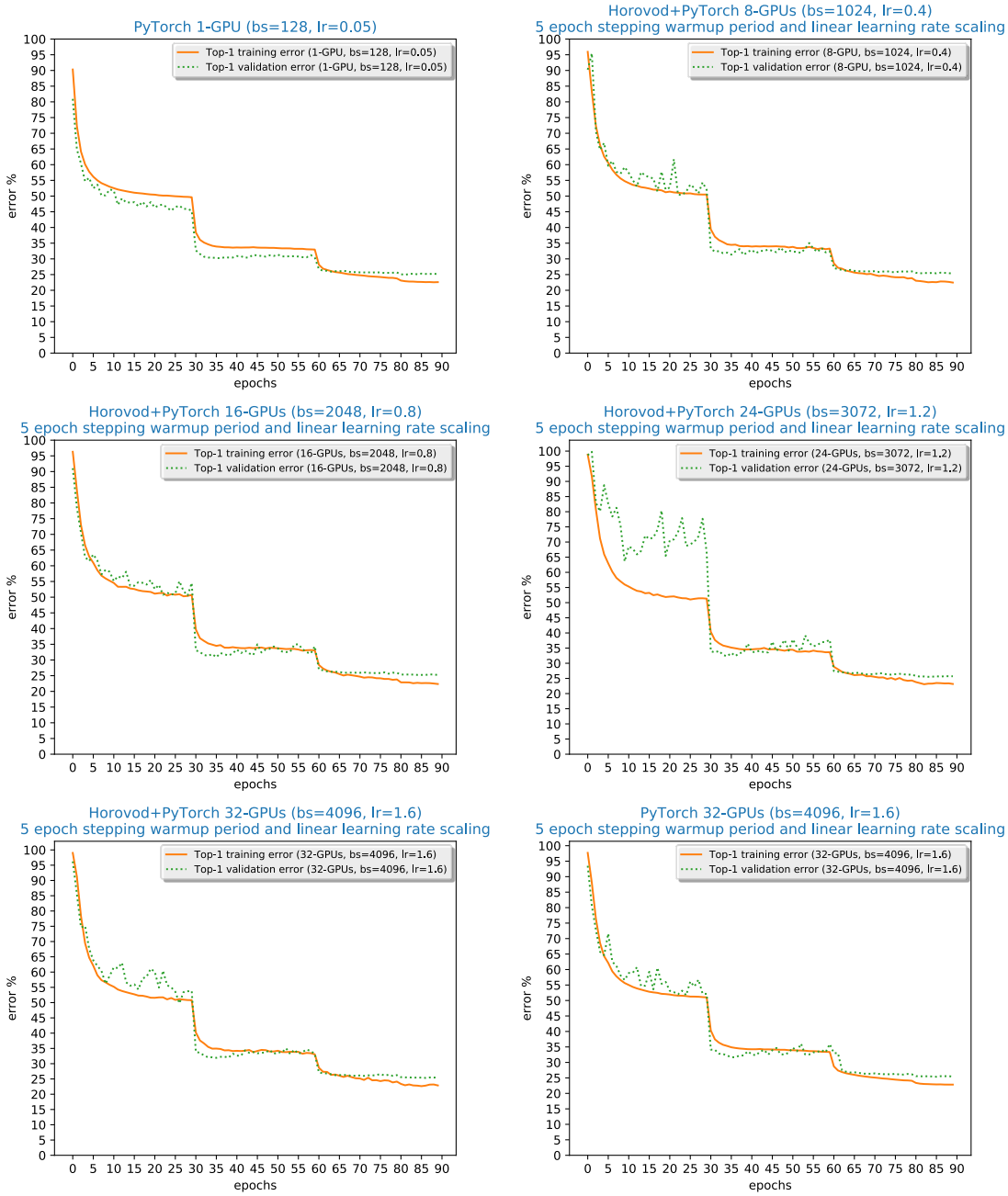

Figure 25. Grid plot of training and validation error. Linear learning rate scaling + 5 epoch stepping warmup.

5.7 Experiment 7 – Model quality – Linear learning rate scaling with 5 epoch stepping warmup period and reduced batch size

In experiment 7 we trained a ResNet-50 architecture deep convolutional neural network with the ILSVRC 2012 dataset for the task of 1000 class image classification. We reduced the per worker batch size to 32 to reduce the global batch size. The network was trained for 90 epochs while utilizing 32 GPU workers. We applied the linear scaling rule to select the appropriate learning rate for 32 GPU workers with a per worker batch size of 32 (Goyal *et al.*, 2017). As with previous experiments, all other hyperparameters were kept intact. We applied a stepping learning rate warmup period of 5 epochs during the start of training and measured the wall clock time taken to train the network.

The slurm batch script that were used for applying resources and launching the experiment can be seen in Appendix 10. The hyperparameters, data transforms, learning rate schedule and other relevant settings of the experiment can be seen in Appendix 11.

When we inspect the results of experiment 7 in Table 14, we notice that the best Top-1/Top-5 validation accuracy is on par with the best Top-1/Top-5 validation accuracies that were reported in Table 13 for networks that were trained with the per worker batch size of 128. When comparing the results for 32 GPUs per worker batch size 32 in Table 14 with the corresponding results for 32 GPUs per worker batch size 128 in Table 13, we also notice that the wall clock time for training the network for 90 epochs on 32 GPUs has gone up from 6 hours 36 minutes to 8 hours.

Table 13. Linear learning rate scaling with 5 epoch stepping warmup and reduced per GPU worker batch size. Validation accuracy and training time.

Deep learning framework	Linearly scaled learning rate	Batch size per GPU	Global batch size	Number of GPUs	Best Top-1/Top-5 validation accuracy	Wall clock time taken for training 90 epochs
Horovod 0.15.2 + PyTorch 1.0.0	0.4	32	1024	32 (8 * 4)	74.766% / 92.204%	0 days 8 hours 0 minutes

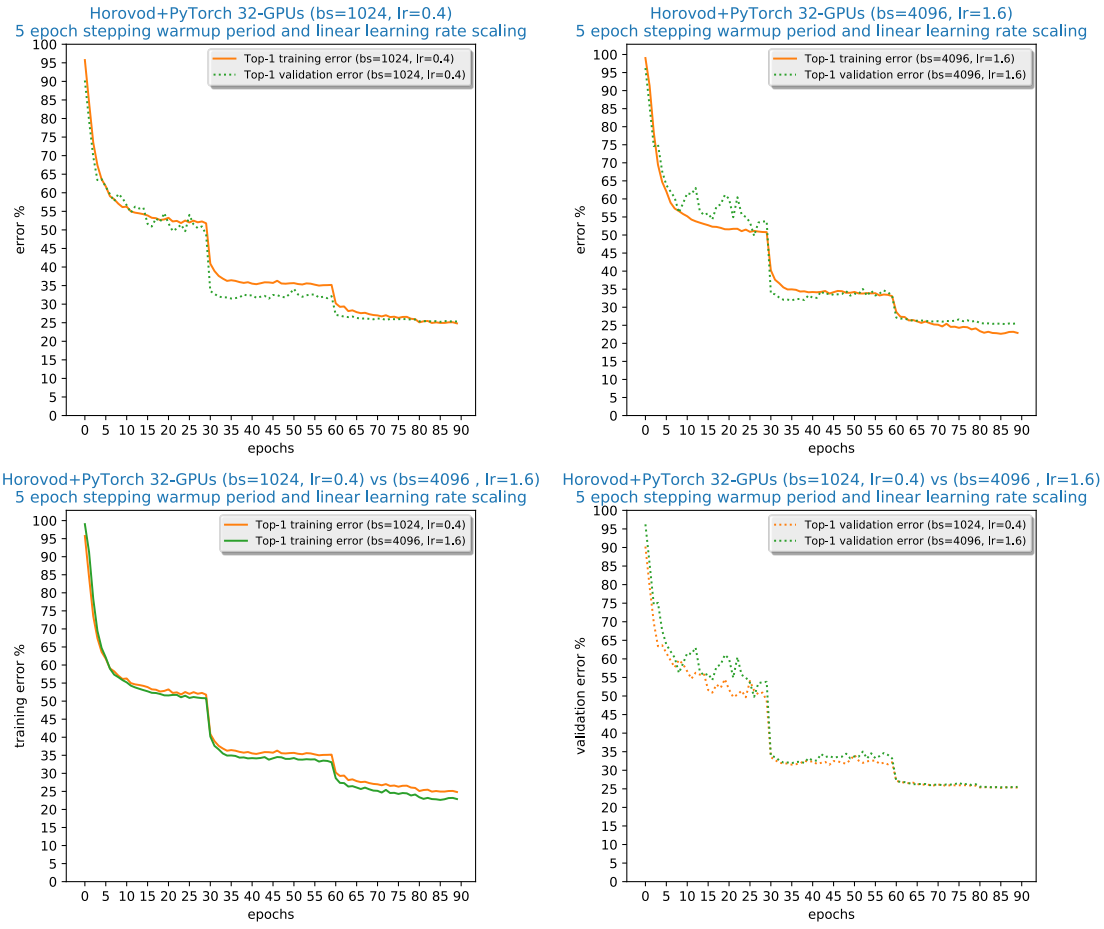

Figure 26. Grid plot of training and validation error. Per GPU worker batch size 32 vs. batch size 128.

When we inspect the plots of validation and training error in Figure 26, we notice that the network that was trained with the smaller per worker batch size of 32, has fewer generalization issues during the early stages of training. This is expected, since the global batch size and linearly scaled learning rate is smaller when we utilize the per

worker batch size of 32. We also notice that the validation error curve for per worker batch size of 32 matches the validation error curve for per worker batch size 128 after the second scheduled drop in learning rate at epoch 60. We also notice that the training error curves not completely align. According to Goyal et al., the aligning training error curves can be utilized as a reliable proxy for success well before training finishes, when modifying settings (Goyal *et al.*, 2017). Apparently this statement does not hold true when the modified setting is the per GPU worker batch size. In their paper, Goyal et al. conducted their experiments utilizing a constant per GPU worker batch size of 32 (Goyal *et al.*, 2017).

When we inspect the plot of GPU CPU utilization in Figure 27, we notice that we are not able to fully saturate the GPUs CPU when we reduce the per worker batch size to 32.

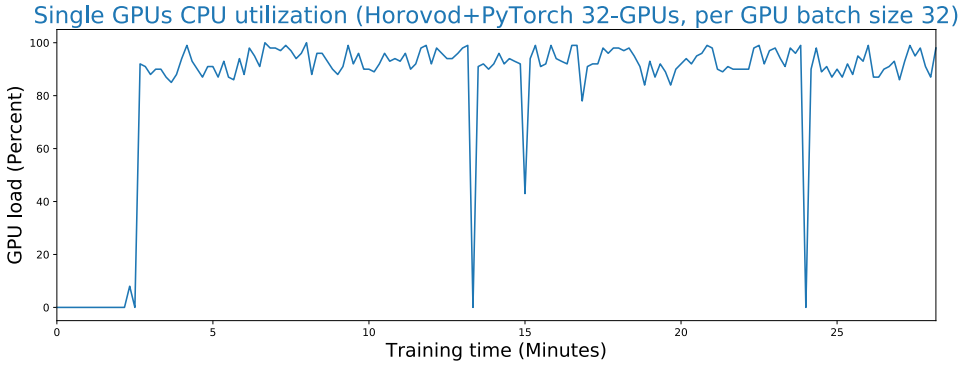

Figure 27.GPU CPU utilization on a single GPU, Horovod+PyTorch 32 GPUs batch size 32.

5.8 Experiment 8 – Model quality – Linear learning rate scaling with 5 epoch gradual warmup period

In experiment 8 we trained three ResNet-50 architecture deep convolutional neural network with the ILSVRC 2012 dataset for the task of 1000 class image classification.
The networks were trained for 90 epochs with a different numbers of GPU workers. The per GPU worker batch size was set to the maximum value that could fit in a single Pascal P100 GPUs memory to ramp-up the global batch size and to speed up the training process. We applied the linear scaling rule to select the appropriate learning rates for the different worker counts (Goyal *et al.*, 2017). As with the previous experiment, all other hyperparameters were again left to their default values. We also applied a gradual learn-

ing rate warmup period of 5 epochs during the initial epochs of training (Goyal *et al.*, 2017). A pseudocode example for implementing the 5 epoch gradual learning rate warmup period can be seen in Appendix 13. We also took note of the wall clock time taken for the networks to train for 90 epochs.

Samples of the slurm batch scripts that were used for applying resources and launching the experiments can be seen in Appendixes 6 (baseline) and 12. The hyperparameters, data transforms, learning rate schedule and other relevant settings of the experiment can be seen in Appendixes 7 (baseline) and 13.

When we inspect the results of experiment 8 in Table 15, we notice that the training times and best Top-1/Top-5 validation accuracies are on par with the corresponding results in experiment 6, where we utilized the linear scaling rule combined with a 5 epoch stepping warmup period.

Table 14. Linear learning rate scaling with 5 epoch gradual warmup. Validation accuracy and training time.

Deep learning framework	Linearly scaled learning rate	Batch size per GPU	Global batch size	Number of GPUs	Best Top-1/Top-5 validation accuracy	Wall clock time taken for training 90 epochs
PyTorch 1.0.0 (baseline)	0.05	128	128	1	75.028% / 92.296%	6 days 4 hours 16 minutes
Horovod 0.15.2 + PyTorch 1.0.0	0.8	128	2048	16 (4 * 4)	74.736% / 92.206%	0 days 11 hours 15 minutes
Horovod 0.15.2 + PyTorch 1.0.0	1.2	128	3072	24 (6 * 4)	74.446% / 92.096%	0 days 8 hours 8 minutes
Horovod 0.15.2 + PyTorch 1.0.0	1.6	128	4096	32 (8 * 4)	74.298% / 91.956%	0 days 6 hours 33 minutes

As we inspect the plot of training error in Figure 28, we notice that the 5 epoch gradual warmup scheme produces very similar training and validation error curves when compared to the 5 epoch stepping warmup scheme, which we experimented with in experiment 6. We again notice, that the training curves for different global batch sizes align, when a learning rate warmup period is applied during training. The misalignment in validation error again dissipates, during the scheduled drops in learning rate at epochs 30, 60 and 80. We interpret this as follows: We gain no additional benefits from switching the warmup scheme from 5 epoch stepping warmup to 5 epoch gradual warmup, when training with a global batch sizes of 4096 or less. We note that gradual warmup increases the learning rate gradually in smaller steps, which may provide additional benefits when training with even larger global batch sizes, where we tend to utilize even larger learning rates. According to Goyal et al., gradually ramping up the learning rate avoids a sudden increase of the learning rate, which allows for healthier convergence during the initial phases of training when network is changing rapidly (Goyal *et al.*, 2017).

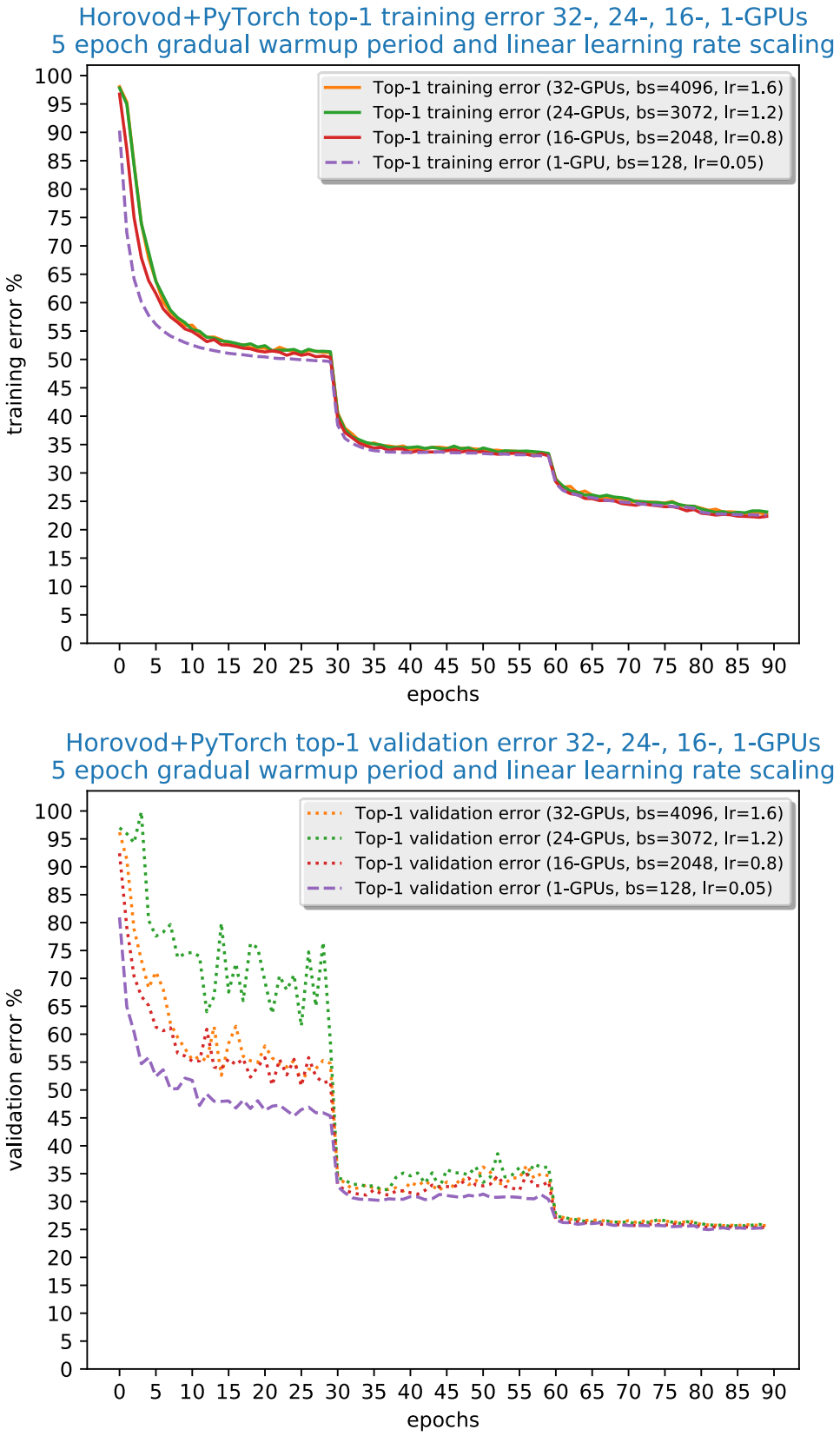

Figure 28. Plot of training and validation error. Linear learning rate scaling + 5 epoch gradual warmup.

As we inspect the plot of training and validation error in Figure 29, we notice a small spike in validation error when the 5 epoch gradual warmup period ends. An exception to this is batch size 3072, where we again notice potential generalization issues during the early stages of training, which dissipate during the first scheduled drop in learning rate at epoch 30.

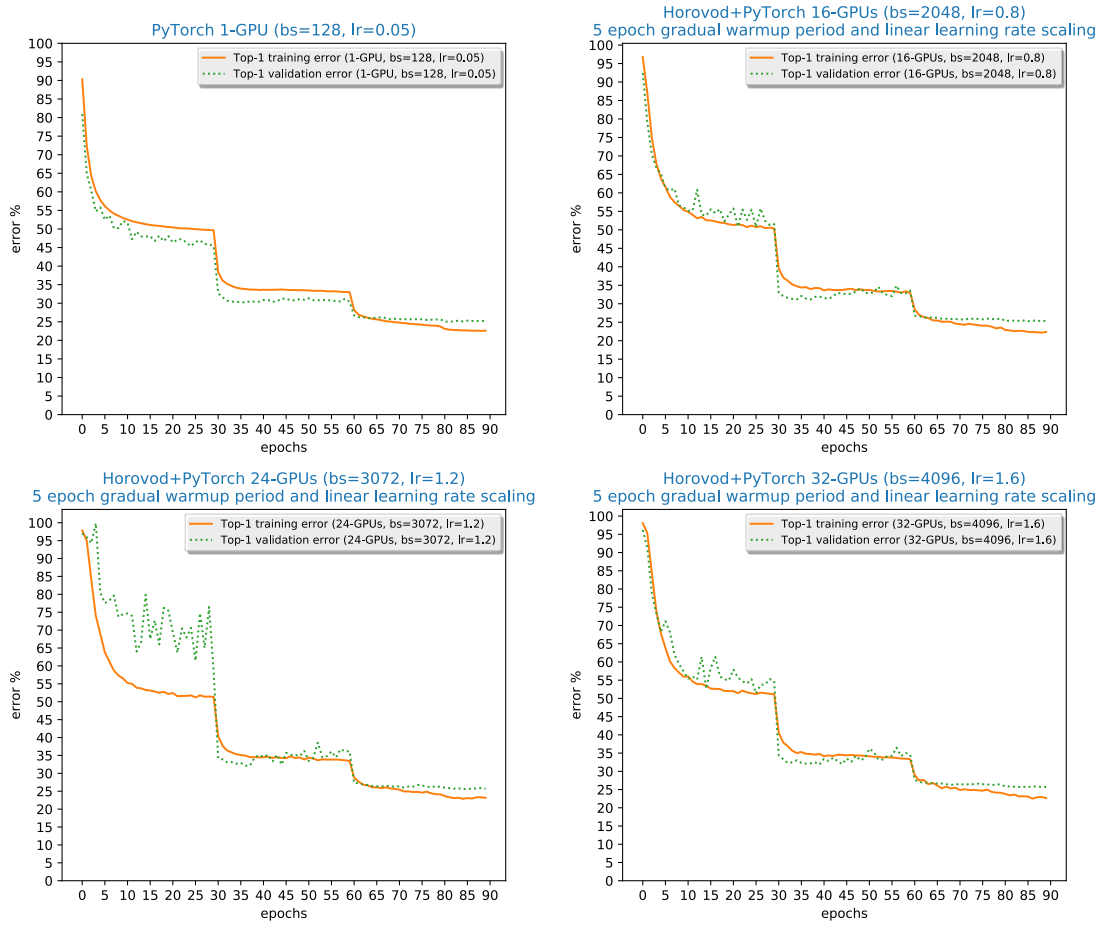

Figure 29. Grid plot of training and validation error. Linear learning rate scaling + 5 epoch gradual warmup.

5.9 Experiment 9 – Model quality – Linear learning rate scaling with 10 epoch gradual warmup period

In experiment 9 we trained two ResNet-50 architecture deep convolutional neural network with the ILSVRC 2012 dataset for the task of 1000 class image classification.
The two networks were trained for 90 epochs both utilizing a different numbers of GPU workers. The per GPU worker batch size was set to the maximum value that could fit in a single Pascal P100 GPUs memory to ramp-up the global batch size and to speed up the training process. We applied the linear scaling rule to select the appropriate learning

rates for the two different worker counts (Goyal *et al.*, 2017). As with the previous experiment, all other hyperparameters were again left to their default values. We also applied a gradual learning rate warmup period of 10 epochs during the initial epochs of training (Goyal *et al.*, 2017). A pseudocode example for implementing the 10 epoch gradual learning rate warmup period can be seen in Appendix 15. We also took note of the wall clock time taken for the networks to train for 90 epochs.

The slurm batch scripts that were used for applying resources and launching the experiments can be seen in Appendix 14. The hyperparameters, data transforms, learning rate schedule and other relevant settings of the experiment can be seen in Appendix 15.

When inspecting the results of experiment 9 in Table 16, we notice that the best Top-1/Top-5 validation accuracy and the wall clock time taken for training the network for 90 epochs is on par with the corresponding values in experiments 6 and 8. We interpret these similarities in best validation accuracy as the resulting model and training process not gaining any additional benefits from the longer gradual warmup period.

Table 15. Linear learning rate scaling with 10 epoch gradual warmup. Validation accuracy and training time.

Deep learning framework	Linearly scaled learning rate	Batch size per GPU	Global batch size	Number of GPUs	Best Top-1/Top-5 validation accuracy	Wall clock time taken for training 90 epochs
Horovod 0.15.2 + PyTorch 1.0.0	1.2	128	3 072	24 (6 * 4)	74.546% / 92.250%	0 days 8 hours 9 minutes
Horovod 0.15.2 + PyTorch 1.0.0	1.6	128	4096	32 (8 * 4)	74.638% / 92.180%	0 days 6 hours 33 minutes

As we inspect the plot of training and validation error in Figure 30, we notice small spikes in validation error when the 5 epoch and 10 epoch gradual warmup periods end. After the first scheduled drop in learning rate at epoch 30, the plots for training and validation error seem to follow a similar pattern, which we interpret as the resulting models having good generalization capabilities. We notice that batch size 3072 again displays potential generalization issues during early stages of training, which are not mitigated by the longer 10 epoch gradual warmup period. We interpret the similarity in the validation and training error curves for the two different warmup schemes as the resulting models and the training process not gaining any additional benefits from a longer gradual warmup period.

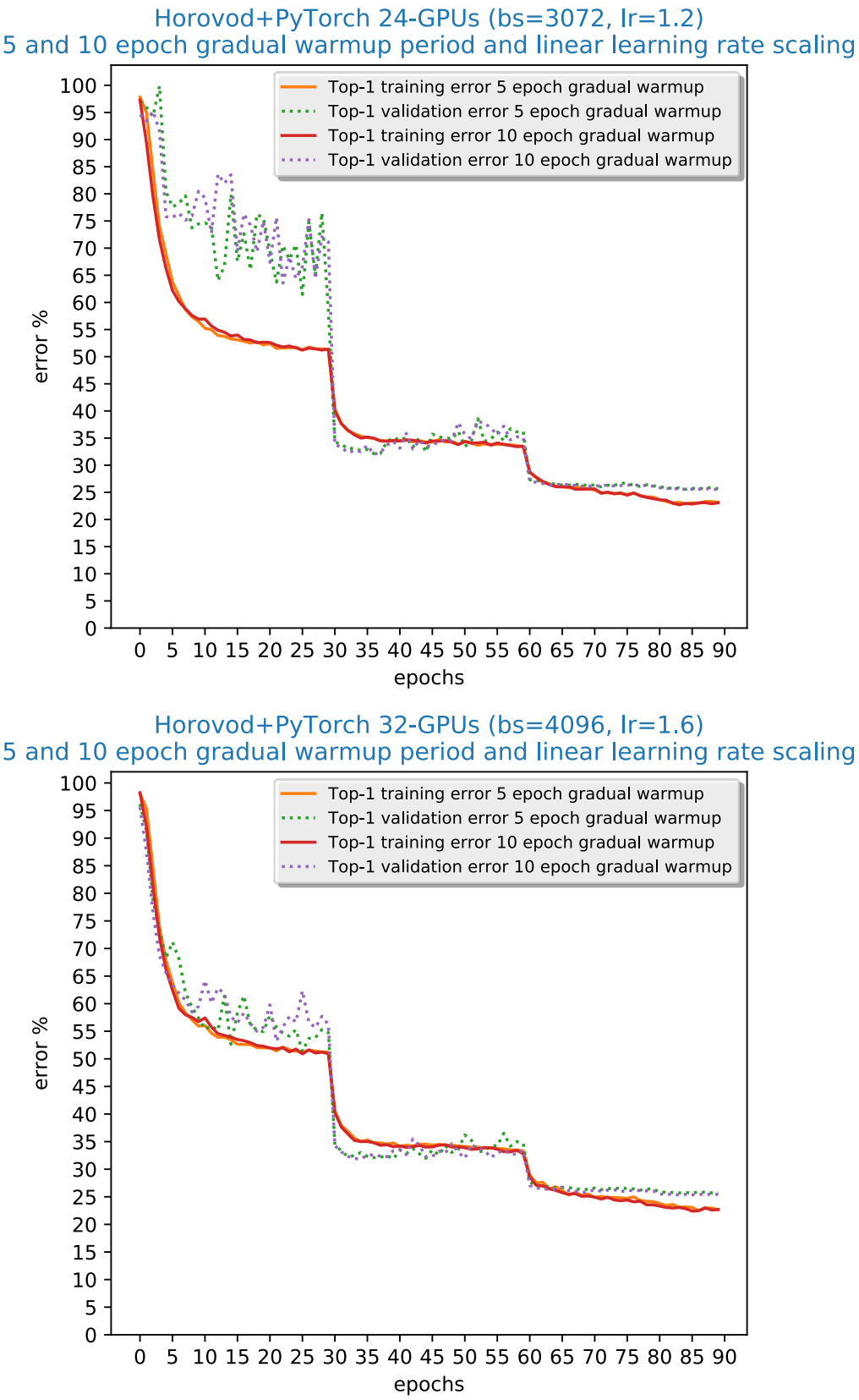

Figure 30. Plot of training and validation error. 5 epoch gradual warmup vs. 10 epoch gradual warmup.

6 DISCUSSION AND CONCLUSIONS

In this chapter, we first provide a short summary of our work. Then we provide the answers for our research questions. In the end of this chapter, we discuss interesting avenues for further research.

This thesis was an integral part of an internal competence development effort at CSC in distributed deep learning. With the attained knowledge, we aim to better serve researchers who wish to scale out their deep learning research on the CSC provided infrastructure.

We were able to gain better insight and hands-on experience on how to scale out in deep learning to multiple nodes and multiple GPUs while utilizing the linear learning rate scaling and learning rate warmup methods. The methods outlined by Goyal et al. enabled us to reduce training times by scaling out to multiple cluster nodes, without degrading the resulting models quality in the process (Goyal *et al.*, 2017). We were also able to measure the scaling performance of our distributed programs, and to utilize the Horovod distributed training framework in the provided execution environment. The experiments in this thesis were performed and documented in a way, that they can easily be reproduced or used as source material for a guide on how to scale out in deep learning, in the provided execution environment.

6.1 Summary of Research Questions

A summary of the answers to our research questions is presented below in the format of main key points and conclusions.

RQ1: *What are the implications of using smaller per GPU worker batch sizes vs. using larger per GPU worker batch sizes in terms of scaling performance, training time (wall clock) and efficient resource utilization in the provided execution environment?*

In experiments 1 to 3, we were able to observe a reduction in scaling performance when the training was conducted with a smaller per GPU worker batch size. In experiments 6 and 7, we were able to observe degraded resource utiliza-

tion and a longer 90 epoch training time, when the training was conducted with a smaller per GPU worker batch size. On the other hand, in experiment 7 the model trained with a smaller per GPU worker batch size displayed better generalization capabilities during the early stages of training.

Based on our experiments we make the following conclusions: Using larger per GPU worker batch sizes enable us to reduce training time (wall clock) and to better utilize the available hardware. Using smaller per GPU worker batch sizes enables us to reduce the global batch size; The reduced global batch size enables us to scale out training to a larger number of GPU workers, without reaching the upper bound for the dataset's global batch size. For discussion about the global batch size upper bound, please refer to the discussion in RQ3 below.

RQ2: *Does the Horovod distributed training framework provide a speedup in the provided execution environment when scaling out training?*

In experiments 1 to 3, we were able to measure the scaling performance of the PyTorch Examples ImageNet training script and the scaling performance of the Horovod distributed version of the same PyTorch Examples ImageNet training script. In experiment 6, we measured the 90 epoch training time (wall clock) of the two training scripts, when training was scaled out to 32 GPU workers.

Based on our experiments we make the following conclusions: The speedup provided by the Horovod framework is evident when observing the scale out performance of the PyTorch Examples ImageNet training script and the reported 90 epoch training time (wall clock). Although we do not have a similar reference point for Horovod distributed TensorFlow, the throughput results provided by the tf_cnn_benchmarks script can be used to benchmark the provided execution environment with other execution environments which have performed the same benchmark.

RQ3: *What are the model quality implications of larger global batch sizes when utilizing methods such as linear learning rate scaling and gradual learning rate warmup?*

In experiment 4, we were able to show the negative implications of large global batch sizes with regard to model accuracy, when scaling out training using a naïve method. In experiments 5 to 9, we were able to show how linear learning rate

scaling combined with a learning rate warmup period can be applied with large global batch sizes to mitigate model quality issues.

Based on our experiments we make the following conclusions:

Linear learning rate scaling combined with a learning rate warmup provide a means for easily scaling out training to multiple nodes and multiple GPU workers, without a significant decrease in validation accuracy. The linear scaling rule is easy to understand and implement, since it only involves adjusting one hyperparameter (learning rate) when scaling out training. Adjusting the learning rate without implementing and applying a learning rate warmup period has a negative effect on the resulting model's validation accuracy. Both methods should be applied when scaling out for training. Global batch size has an upper bound which is dataset dependent. When this upper bound for the global batch size is reached, the models quality begins to degrade, even though we implement learning rate scaling and a learning rate warmup period. For the ILSVRC 2012 dataset the global batch size upper bound is batch size 8192 (Goyal *et al.*, 2017). Thus, for the ILSVRC 2012 dataset, using smaller per GPU worker batch sizes (e.g. 32) may be beneficial when scaling out training beyond 64 GPU workers.

6.2 Future research

In our research, we measured the throughput, scaling performance, training time (wall clock) and model quality of a ResNet-50 architecture deep convolutional neural network trained with the ILSVRC 2012 dataset for the task of 1000 class image classification on multiple GPUs while utilizing the Horovod, TensorFlow and PyTorch frameworks. It would be interesting to perform similar measurements for different deep neural network architectures and deep learning frameworks, to gain better insight on the unique characteristics of different deep neural network architectures and different deep learning frameworks.

In their paper Goyal et al. show that the linear learning rate scaling and gradual warmup methods are also applicable to other problems in the computer vision domain, such as object detection and image segmentation (Goyal *et al.*, 2017). An interesting avenue for

future research would be to gain better insight on whether the methods outlined by Goyal et al. are applicable to problems outside the computer vision domain.

In our research, our models were trained with the floating point precision of 32 bits (FP32). It would be interesting to reconduct our experiments in half-precision (FP16), to be able to measure the speedup gain and model quality implications of conducting training in half-precision (Svyatkovskiy, Kates-Harbeck and Tang, 2017).

During our experiments, we observed reoccurring generalization issues, during the initial stages of training, when training was conducted with 24 GPU workers and the global batch size of 3072. We were not able to identify what was causing these generalization issues. Future research could be conducted in finding out the cause for these reoccurring issues.

At the time of writing CSC is renewing its HPC environment. The new HPC environment will include a new GPU partition with top-of-the-line hardware. It would be interesting to conduct the same experiments in the new HPC environment, to see what kind of throughput, scaling performance and training times (wall clock) we are able to achieve on the new hardware.

REFERENCES

Akiba, T., Fukuda, K. and Suzuki, S. (2017) 'ChainerMN: Scalable Distributed Deep Learning Framework'. Available at: http://arxiv.org/abs/1710.11351.

Akiba, T., Suzuki, S. and Fukuda, K. (2017) 'Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes'. Available at: http://arxiv.org/abs/1711.04325.

Ben-Nun, T. and Hoefler, T. (2018) 'Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis'. Available at: http://arxiv.org/abs/1802.09941.

Chen, C. *et al.* (2016) 'Stochastic Gradient MCMC with Stale Gradients', (Nips). Available at: http://arxiv.org/abs/1610.06664.

Chilimbi, T. *et al.* (2014) 'Project Adam: Building an Efficient and Scalable Deep Learning Training System', *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, pp. 571–582. Available at: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi.

Chollet, François. (2017). Deep Learning with Python, 1st edition, Manning Publications, 384 pages.

COCO. (2018). COCO - Common Objects in Context. Available from: http://cocodataset.org/#home Accessed 11.5.2019

Dai, W. *et al.* (2014) 'High-Performance Distributed ML at Scale through Parameter Server Consistency Models', pp. 1–19. Available at: http://arxiv.org/abs/1410.8043.

Dutta, S. *et al.* (2018) 'Slow and Stale Gradients Can Win the Race: Error-Runtime Trade-offs in Distributed SGD'. Available at: http://arxiv.org/abs/1803.01113.

Géron, Aurélien. (2017). Hands-On Machine Learning with Scikit-Learn & TensorFlow, 1st edition, O'Reilly Media, Inc., 574 pages.

Goyal, P. *et al.* (2017) 'Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour'. Available at: http://arxiv.org/abs/1706.02677.

Gupta, S., Zhang, W. and Milthorpe, J. (2015) 'Model Accuracy and Runtime Tradeoff in Distributed Deep Learning', *arXiv e-prints*, p. 15. doi: 10.1109/ICDM.2016.122.

He, K. *et al.* (2015) 'Deep Residual Learning for Image Recognition'. Available at: http://arxiv.org/abs/1512.03385.

Hegde, V. and Usmani, S. (2016) ‘Parallel and Distributed Deep Learning’, *Tech Report*. doi: 10.1039/b103896f.

Horovod. (2019). GitHub - horovod/horovod: Distributed training framework for TensorFlow, Keras, PyTorch, and Apache MXNet. Available from: https://github.com/horovod/horovod Accessed 11.5.2019

Horovod FP16. (2018). What is the difference between FP16 and FP32 when doing deep learning?. Available from: https://www.quora.com/What-is-the-difference-between-FP16-and-FP32-when-doing-deep-learning Accessed 11.5.2019

Horovod PyTorch Examples ImageNet. (2019). dl-experiments/pytorch-horovod-imagenet.py at da06c8b6776f8852cb883fee30fc8ac61e25646b · juhany/dl-experiments. Available from: https://github.com/juhany/dl-experiments/blob/da06c8b6776f8852cb883fee30fc8ac61e25646b/pytorch-horovod-imagenet.py Accessed 12.5.2019

Iandola, F. N. *et al.* (2015) ‘FireCaffe: near-linear acceleration of deep neural network training on compute clusters’, pp. 1–13. Available at: http://arxiv.org/abs/1511.00175.

ILSVRC. (2015). ImageNet Large Scale Visual Recognition Competition (ILSVRC). Available from: http://www.image-net.org/challenges/LSVRC/ Accessed 11.5.2019

ImageNet. (2016). ImageNet. Available from: http://www.image-net.org/about-overview Accessed 11.5.2019

Ioffe, S. and Christian Szegedy (2017) ‘Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift Sergey’, *Journal of Molecular Structure*. doi: 10.1016/j.molstruc.2016.12.061.

Keskar, N. S. *et al.* (2016) ‘On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima’, pp. 1–16. Available at: http://arxiv.org/abs/1609.04836.

Krizhevsky, A., Sutskever, I. and Hinton., G. E. (2012) ‘Imagenet classification with deep convolutional neural networks. In Advances in neural information’, *Advances in neural information processing systems*. doi: 10.1016/B978-008046518-0.00119-7.

Lecun, Y., Bengio, Y. and Hinton, G. (2015) ‘Deep learning’. doi: 10.1038/nature14539.

Ma, M. *et al.* (2018) ‘Democratizing Production-Scale Distributed Deep Learning’. Available at: http://arxiv.org/abs/1811.00143.

Message Passing Interface Forum (2012) ‘MPI: A Message-Passing Interface Standard Version 3.0’, p. 852.

Nesterov, Y. E. (1983) ‘A Method of Solving A Convex Programming Problem with Convergence Rate O(1/k2)’, *Soviet Mathematics Doklady*.

Patarasuk, P. and Yuan, X. (2009) ‘Bandwidth optimal all-reduce algorithms for clusters of workstations’, *Journal of Parallel and Distributed Computing*, 69(2), pp. 117–124. doi: 10.1016/j.jpdc.2008.09.002.

Polyak, B. T. (1964) ‘Some methods of speeding up the convergence of iteration methods’, *USSR Computational Mathematics and Mathematical Physics*. doi: 10.1016/0041-5553(64)90137-5.

Prace. (2019). Best Practice Guide - Deep Learning, February 2019 - PRACE Research Infrastructure. Available from: http://www.prace-ri.eu/best-practice-guide-deep-learning Accessed 11.5.2019

PyTorch Examples ImageNet. (2019). examples/main.py at 447974f6337543d4de6b888e244a964d3c9b71f6 · pytorch/examples · GitHub. Available from: https://github.com/pytorch/examples/blob/447974f6337543d4de6b888e244a964d3c9b71f6/imagenet/main.py Accessed 11.5.2019

PyTorch ImageNet resize. (2015). imagenet-multiGPU.torch/README.md at 0b7632948af04d84211733e0aa1b018706760e5d · soumith/imagenet-multiGPU.torch · GitHub. Available from: https://github.com/soumith/imagenet-multiGPU.torch/blob/0b7632948af04d84211733e0aa1b018706760e5d/README.md Accessed 11.5.2019

PyTorch ImageNet valprep. (2015). imagenetloader.torch/valprep.sh at 12be602fc76b55808fb0bd775b46acebc3a5aeb9 · soumith/imagenetloader.torch · GitHub. Available from: https://github.com/soumith/imagenetloader.torch/blob/12be602fc76b55808fb0bd775b46acebc3a5aeb9/valprep.sh Accessed 11.5.2019

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986) ‘Learning representations by back-propagating errors’, *Nature*. doi: 10.1038/323533a0.

Russakovsky, O. *et al.* (2015) ‘ImageNet Large Scale Visual Recognition Challenge’, *International Journal of Computer Vision*, 115(3), pp. 211–252. doi: 10.1007/s11263-015-0816-y.

sbatch. (2019). Slurm Workload Manager - sbatch. Available from: https://slurm.schedmd.com/sbatch.html Accessed 11.5.2019

Schmidhuber, J. (2015) ‘Deep Learning in neural networks: An overview’, *Neural Networks*, 61, pp. 85–117. doi: 10.1016/j.neunet.2014.09.003.

Senior, A. *et al.* (2013) ‘An empirical study of learning rates in deep neural networks for speech recognition’, in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*. doi: 10.1109/ICASSP.2013.6638963.

Sergeev, A. and Del Balso, M. (2018) ‘Horovod: fast and easy distributed deep learning in TensorFlow’, (September). Available at: http://arxiv.org/abs/1802.05799.

Svyatkovskiy, A., Kates-Harbeck, J. and Tang, W. (2017) ‘Training distributed deep recurrent neural networks with mixed precision on GPU clusters’, in. doi: 10.1145/3146347.3146358.

Taito-GPU hardware. (2018). CSC - 6.1 Taito-GPU hardware and operating system. Available from: https://research.csc.fi/taito-gpu-hardware Accessed 11.5.2019

Teng, M. and Wood, F. (2018) ‘High Throughput Synchronous Distributed Stochastic Gradient Descent’. Available at: http://arxiv.org/abs/1803.04209.

tf_cnn_benchmarks. (2018). benchmarks/scripts/tf_cnn_benchmarks at cnn_tf_v1.12_compatible · tensorflow/benchmarks · GitHub. Available from: https://github.com/tensorflow/benchmarks/tree/cnn_tf_v1.12_compatible/scripts/tf_cnn_benchmarks Accessed 11.5.2019

TFRecord. (2017). models/download_and_preprocess_imagenet.sh at master · tensorflow/models · GitHub. Available from: https://github.com/tensorflow/models/blob/master/research/inception/inception/data/download_and_preprocess_imagenet.sh Accessed 11.5.2019

Using Taito-GPU. (2018). CSC - 6. Using Taito-GPU. Available from: https://research.csc.fi/taito-gpu Accessed 11.5.2019

WordNet. (2019). WordNet | A Lexical Database for English. Available from: https://wordnet.princeton.edu/ Accessed 11.5.2019

Xiang, J. *et al.* (2014) ‘Using extreme learning machine for intrusion detection in a big data environment’, in. doi: 10.1145/2666652.2666664.

You, Y., Gitman, I. and Ginsburg, B. (2017) ‘Large Batch Training of Convolutional Networks’, pp. 1–8. Available at: http://arxiv.org/abs/1708.03888.

APPENDIX 1. SLURM BATCH SCRIPTS FOR EXPERIMENT 1

Example of sbatch script for launching tf_cnn_benchmarks on one cluster node while reserving one Pascal P100 GPU. The per GPU batch size is set to 32.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --tasks-per-node 1
#SBATCH --cpus-per-task 7
#SBATCH -D .
#SBATCH -J 1-node-1-gpu-tf-horovod-bs32-benchmark
#SBATCH -o 1-node-1-gpu-tf-horovod-bs32-benchmark.out.%j
#SBATCH -e 1-node-1-gpu-tf-horovod-bs32-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:1
#SBATCH -t 01:00:00
#SBATCH --mem=200G
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Launch training script.
mpirun -np 1 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
      --model resnet50 \
      --batch_size 32 \
      --variable_update horovod \
      --data_dir $TMPDIR/ilsvrc2012/ \
      --data_name imagenet \
      --num_batches=200
```

Example of sbatch script for launching PyTorch Examples ImageNet training script on one cluster node while reserving one Pascal P100 GPU. The per GPU batch size is set to 128.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --tasks-per-node 1
#SBATCH --cpus-per-task 7
#SBATCH -D .
#SBATCH -J 1-node-1-gpu-pytorch-bs-128-benchmark
#SBATCH -o 1-node-1-gpu-pytorch-bs-128-benchmark.out.%j
#SBATCH -e 1-node-1-gpu-pytorch-bs-128-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:1
#SBATCH -t 01:00:00
#SBATCH --mem=60G
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Launch training script.
srun python ./main.py -a resnet50 --print-freq 1 --batch-size 128 --lr "0.05" \
--world-size 1 --rank 0 --epochs 1 --workers 4 $TMPDIR/ilsvrc2012-torch/
```

Example of sbatch script for launching Horovod+PyTorch Examples ImageNet training script on one cluster node while reserving one Pascal P100 GPU. The per GPU batch size is set to 128.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --tasks-per-node 1
#SBATCH --cpus-per-task 7
#SBATCH -D .
#SBATCH -J 1-node-1-gpu-pytorch-horovod-bs-128-benchmark
#SBATCH -o 1-node-1-gpu-pytorch-horovod-bs-128-benchmark.out.%j
#SBATCH -e 1-node-1-gpu-pytorch-horovod-bs-128-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:1
#SBATCH -t 01:00:00
#SBATCH --mem=60G
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-torch-resized-new.tar -C $TMPDIR/

# Launch training script.
mpirun -np 1 -bind-to none -map-by slot \
       -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
       -mca pml ob1 -mca btl ^openib \
       -oversubscribe \
       python3.6 pytorch-horovod-imagenet.py --bs 128 --lr 0.05 --epochs 1 \
       --data $TMPDIR/ilsvrc2012-torch/
```

APPENDIX 2. SLURM BATCH SCRIPTS FOR EXPERIMENT 2

Example of sbatch script for launching tf_cnn_benchmarks on one cluster node while reserving two Pascal P100 GPU. The per GPU batch size is set to 32.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --tasks-per-node 2
#SBATCH --cpus-per-task 7
#SBATCH -D .
#SBATCH -J 1-node-2-gpu-tf-horovod-bs32-benchmark
#SBATCH -o 1-node-2-gpu-tf-horovod-bs32-benchmark.out.%j
#SBATCH -e 1-node-2-gpu-tf-horovod-bs32-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:2
#SBATCH -t 01:00:00
#SBATCH --mem=200G
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Launch training script.
mpirun -np 2 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
      --model resnet50 \
      --batch_size 32 \
      --variable_update horovod \
      --data_dir $TMPDIR/ilsvrc2012/ \
      --data_name imagenet \
      --num_batches=200
```

Example of sbatch script for launching tf_cnn_benchmarks on one cluster node while reserving four Pascal P100 GPU. The per GPU batch size is set to 32.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --tasks-per-node 4
#SBATCH --cpus-per-task 7
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 1-node-4-gpu-tf-horovod-bs32-benchmark
#SBATCH -o 1-node-4-gpu-tf-horovod-bs32-benchmark.out.%j
#SBATCH -e 1-node-4-gpu-tf-horovod-bs32-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 01:00:00
#SBATCH --mem=0
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Launch training script.
mpirun -np 4 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
      --model resnet50 \
      --batch_size 32 \
      --variable_update horovod \
      --data_dir $TMPDIR/ilsvrc2012/ \
      --data_name imagenet \
      --num_batches=200
```

Example of sbatch script for launching PyTorch Examples ImageNet training script on one cluster node while reserving two Pascal P100 GPU. The per GPU batch size is set to 128.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --tasks-per-node 1
#SBATCH --cpus-per-task 7
#SBATCH -D .
#SBATCH -J 1-node-2-gpu-pytorch-bs-128-benchmark
#SBATCH -o 1-node-2-gpu-pytorch-bs-128-benchmark.out.%j
#SBATCH -e 1-node-2-gpu-pytorch-bs-128-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:2
#SBATCH -t 01:00:00
#SBATCH --mem=60G
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRK-DIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Launch training script.
srun python ./main.py -a resnet50 --print-freq 1 --batch-size 128 --lr "0.05" \
--world-size 1 --rank 0 --epochs 1 --workers 4 $TMPDIR/ilsvrc2012-torch/
```

Example of sbatch script for launching PyTorch Examples ImageNet training script on one cluster node while reserving four Pascal P100 GPU. The per GPU batch size is set to 128.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --tasks-per-node 1
#SBATCH --cpus-per-task 7
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 1-node-4-gpu-pytorch-bs-128-benchmark
#SBATCH -o 1-node-4-gpu-pytorch-bs-128-benchmark.out.%j
#SBATCH -e 1-node-4-gpu-pytorch-bs-128-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 01:00:00
#SBATCH --mem=0
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRK-DIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Launch training script.
srun python ./main.py -a resnet50 --print-freq 1 --batch-size 128 --lr "0.05" \
--world-size 1 --rank 0 --epochs 1 --workers 4 $TMPDIR/ilsvrc2012-torch/
```

Example of sbatch script for launching Horovod+PyTorch Examples ImageNet training script on one cluster node while reserving two Pascal P100 GPU. The per GPU batch size is set to 128.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --tasks-per-node 2
#SBATCH --cpus-per-task 7
#SBATCH -D .
#SBATCH -J 1-node-2-gpu-pytorch-horovod-bs-128-benchmark
#SBATCH -o 1-node-2-gpu-pytorch-horovod-bs-128-benchmark.out.%j
#SBATCH -e 1-node-2-gpu-pytorch-horovod-bs-128-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:2
#SBATCH -t 01:00:00
#SBATCH --mem=60G
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRK-DIR/DONOTREMOVE/DATA/ilsvrc2012-torch-resized-new.tar -C $TMPDIR/

# Launch training script.
mpirun -np 2 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 pytorch-horovod-imagenet.py --bs 128 --lr 0.05 --epochs 1 \
      --data $TMPDIR/ilsvrc2012-torch/
```

Example of sbatch script for launching Horovod+PyTorch Examples ImageNet training script on one cluster node while reserving four Pascal P100 GPU. The per GPU batch size is set to 128.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --tasks-per-node 4
#SBATCH --cpus-per-task 7
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 1-node-4-gpu-pytorch-horovod-bs-128-benchmark
#SBATCH -o 1-node-4-gpu-pytorch-horovod-bs-128-benchmark.out.%j
#SBATCH -e 1-node-4-gpu-pytorch-horovod-bs-128-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 01:00:00
#SBATCH --mem=0
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-torch-resized-new.tar -C $TMPDIR/

# Launch training script.
mpirun -np 4 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 pytorch-horovod-imagenet.py --bs 128 --lr 0.2 --epochs 1 \
      --data $TMPDIR/ilsvrc2012-torch/
```

APPENDIX 3. SLURM BATCH SCRIPTS FOR EXPERIMENT 3

Example of sbatch script for launching tf_cnn_benchmarks on two cluster nodes while reserving eight Pascal P100 GPU. The per GPU batch size is set to 32.

```
#!/bin/bash
#SBATCH -N 2
#SBATCH --tasks-per-node 4
#SBATCH --cpus-per-task 7
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 2-node-4-gpu-tf-horovod-bs32-benchmark
#SBATCH -o 2-node-4-gpu-tf-horovod-bs32-benchmark.out.%j
#SBATCH -e 2-node-4-gpu-tf-horovod-bs32-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 01:00:00
#SBATCH --mem=0
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Launch training script.
mpirun -np 8 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
      --model resnet50 \
      --batch_size 32 \
      --variable_update horovod \
      --data_dir $TMPDIR/ilsvrc2012/ \
      --data_name imagenet \
      --num_batches=200
```

Example of sbatch script for launching tf_cnn_benchmarks on four cluster nodes while reserving sixteen Pascal P100 GPU. The per GPU batch size is set to 32.

```
#!/bin/bash
#SBATCH -N 4
#SBATCH --tasks-per-node 4
#SBATCH --cpus-per-task 7
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 4-node-4-gpu-tf-horovod-bs32-benchmark
#SBATCH -o 4-node-4-gpu-tf-horovod-bs32-benchmark.out.%j
#SBATCH -e 4-node-4-gpu-tf-horovod-bs32-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 01:00:00
#SBATCH --mem=0
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Launch training script.
mpirun -np 16 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
      --model resnet50 \
      --batch_size 32 \
      --variable_update horovod \
      --data_dir $TMPDIR/ilsvrc2012/ \
      --data_name imagenet \
      --num_batches=200
```

Example of sbatch script for launching tf_cnn_benchmarks on six cluster nodes while reserving twenty-four Pascal P100 GPU. The per GPU batch size is set to 32.

```
#!/bin/bash
#SBATCH -N 6
#SBATCH --tasks-per-node 4
#SBATCH --cpus-per-task 7
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 6-node-4-gpu-tf-horovod-bs32-benchmark
#SBATCH -o 6-node-4-gpu-tf-horovod-bs32-benchmark.out.%j
#SBATCH -e 6-node-4-gpu-tf-horovod-bs32-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 01:00:00
#SBATCH --mem=0
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Launch training script.
mpirun -np 24 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
      --model resnet50 \
      --batch_size 32 \
      --variable_update horovod \
      --data_dir $TMPDIR/ilsvrc2012/ \
      --data_name imagenet \
      --num_batches=200
```

Example of sbatch script for launching tf_cnn_benchmarks on eight cluster nodes while reserving thirty-two Pascal P100 GPU. The per GPU batch size is set to 32.

```
#!/bin/bash
#SBATCH -N 8
#SBATCH --tasks-per-node 4
#SBATCH --cpus-per-task 7
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 8-node-4-gpu-tf-horovod-bs32-benchmark
#SBATCH -o 8-node-4-gpu-tf-horovod-bs32-benchmark.out.%j
#SBATCH -e 8-node-4-gpu-tf-horovod-bs32-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 01:00:00
#SBATCH --mem=0
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Launch training script.
mpirun -np 32 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
      --model resnet50 \
      --batch_size 32 \
      --variable_update horovod \
      --data_dir $TMPDIR/ilsvrc2012/ \
      --data_name imagenet \
      --num_batches=200
```

Example of sbatch script for launching PyTorch Examples ImageNet training script on two cluster nodes while reserving eight Pascal P100 GPUs. The training script utilizes the PyTorch frameworks Shared file-system initialization feature to set up communication between the cluster nodes. The per GPU batch size is set to 128. The sbatch scripts for four cluster nodes sixteen GPUs, six cluster nodes twenty-four GPUs and eight cluster nodes thirty-two GPUs follow the same convention as two cluster nodes eight GPUs and can be derived from the example below.

```
#!/bin/bash
#SBATCH -N 2
#SBATCH --tasks-per-node 1
#SBATCH --cpus-per-task 28
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 2-node-8-gpu-pytorch-bs-128-benchmark
#SBATCH -o 2-node-8-gpu-pytorch-bs-128-benchmark.out.%j
#SBATCH -e 2-node-8-gpu-pytorch-bs-128-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 10:00:00
#SBATCH --mem=0
#SBATCH --wait-all-nodes=1
#SBATCH

# PyTorch Shared file-system initialization file.
# Note! You must use the full to path to the file.
INITIALIZATION_FILE=/homeappl/home/nyholmju/INITIALIZATION_FILE
# Remove $INITIALIZATION_FILE on startup.
rm -f $INITIALIZATION_FILE

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRK-DIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Since each node contains 4 GPUs, set the training scripts batch size to 512.
# Per GPU worker batch size = 512 / 4 = 128

# Start the job on node 0. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 0 python ./main.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "0.4" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 2 --rank 0 \
--epochs 1 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# Start the job on node 1. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 1 python ./main.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "0.4" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 2 --rank 1 \
--epochs 1 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# ‘wait’ is needed when running jobs in background with &
wait
```

Example of sbatch script for launching Horovod+PyTorch Examples ImageNet training script on two cluster nodes while reserving eight Pascal P100 GPU. The per GPU batch size is set to 128. The sbatch scripts for four cluster nodes sixteen GPUs, six cluster nodes twenty-four GPUs and eight cluster nodes thirty-two GPUs follow the same convention as two cluster nodes eight GPUs and can be derived from the example below.

```
#!/bin/bash
#SBATCH -N 2
#SBATCH --tasks-per-node 4
#SBATCH --cpus-per-task 7
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 2-node-8-gpu-pytorch-horovod-bs-128-benchmark
#SBATCH -o 2-node-8-gpu-pytorch-horovod-bs-128-benchmark.out.%j
#SBATCH -e 2-node-8-gpu-pytorch-horovod-bs-128-benchmark.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 01:00:00
#SBATCH --mem=0
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-torch-resized-new.tar -C $TMPDIR/

# Launch training script.
mpirun -np 8 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 pytorch-horovod-imagenet.py --bs 128 --lr 0.4 --epochs 1 \
      --data $TMPDIR/ilsvrc2012-torch/
```

APPENDIX 4. SLURM BATCH SCRIPTS FOR EXPERIMENT 4

1 GPU and 2 GPUs

Example of sbatch script for launching PyTorch Examples ImageNet training script on one cluster node while reserving one Pascal P100 GPU. The script trains for 10 epochs and saves the resulting model in a checkpoint file which is utilized for resuming training. The one cluster node two GPUs sbatch script follow the same convention and can be derived from the examples below.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --tasks-per-node 1
#SBATCH --cpus-per-task 7
#SBATCH -D .
#SBATCH -J 1-node-1-gpu-pytorch-bs-128-checkpoint-0-10
#SBATCH -o 1-node-1-gpu pytorch-bs-128-checkpoint-0-10.out.%j
#SBATCH -e 1-node-1-gpu-pytorch-bs-128-checkpoint-0-10.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:1
#SBATCH -t 20:00:00
#SBATCH --mem=60G
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRK-DIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Launch training script.
srun python ./main.py -a resnet50 --print-freq 1 --batch-size 128 --lr "0.1" \
--world-size 1 --rank 0 --epochs 10 --workers 4 $TMPDIR/ilsvrc2012-torch/
```

1 GPU continued

Example of sbatch script for launching PyTorch Examples ImageNet training script on one cluster node while reserving one Pascal P100 GPU. The script resumes training from a checkpoint file and trains for 10 epochs. The resulting model is saved in a checkpoint file which is utilized for resuming training. We repeat the restore from checkpoint and train procedure until we have trained for 90 epochs.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --tasks-per-node 1
#SBATCH --cpus-per-task 7
#SBATCH -D .
#SBATCH -J 1-node-1-gpu-pytorch-bs-128-checkpoint-10-20
#SBATCH -o 1-node-1-gpu-pytorch-bs-128-checkpoint-10-20.out.%j
#SBATCH -e 1-node-1-gpu-pytorch-bs-128-checkpoint-10-20.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:1
#SBATCH -t 20:00:00
#SBATCH --mem=60G
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRK-DIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Launch training script.
srun python ./main.py -a resnet50 --print-freq 1 --batch-size 128 --lr "0.1" \
--world-size 1 --rank 0 --epochs 20 --workers 4  \
--resume ./checkpoint.pth.tar $TMPDIR/ilsvrc2012-torch/
```

32 GPUs / 24 GPUs / 16 GPUs

Example of sbatch script for launching PyTorch Examples ImageNet training script on eight cluster nodes while reserving thirty-two Pascal P100 GPUs. The training script utilizes the PyTorch frameworks Shared file-system initialization feature to set up communication between the cluster nodes. The four cluster nodes sixteen GPUs and six cluster nodes twenty-four GPUs sbatch scripts follow the same convention and can be derived from the eight cluster nodes thirty-two GPUs example below.

```
#!/bin/bash
#SBATCH -N 8
#SBATCH --tasks-per-node 1
#SBATCH --cpus-per-task 28
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 8-node-32-gpu-pytorch-bs-128
#SBATCH -o 8-node-32-gpu-pytorch-bs-128.out.%j
#SBATCH -e 8-node-32-gpu-pytorch-bs-128.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 10:00:00
#SBATCH --mem=0
#SBATCH --wait-all-nodes=1
#SBATCH

# PyTorch Shared file-system initialization file.
# Note! You must use the full to path to the file.
INITIALIZATION_FILE=/homeappl/home/nyholmju/INITIALIZATION_FILE
# Remove $INITIALIZATION_FILE on startup.
rm -f $INITIALIZATION_FILE

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRK-DIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Since each node contains 4 GPUs, set the training scripts batch size to 512.
# Per GPU worker batch size = 512 / 4 = 128
# Start the job on node 0. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 0 python ./main.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "0.1" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 0 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# Start the job on node 1. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 1 python ./main.py -a resnet50 --print-freq 1\
```

```
--batch-size 512 --lr "0.1" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 1 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# Start the job on node 2. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 2 python ./main.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "0.1" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 2 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# Start the job on node 3. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 3 python ./main.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "0.1" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 3 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# Start the job on node 4. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 4 python ./main.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "0.1" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 4 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# Start the job on node 5. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 5 python ./main.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "0.1" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 5 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# Start the job on node 6. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 6 python ./main.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "0.1" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 6 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# Start the job on node 7. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 7 python ./main.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "0.1" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 7 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# ‘wait’ is needed when running jobs in background with &
wait
```

APPENDIX 5. SETTINGS AND HYPERPARAMETERS FOR EXPERIMENT 4

We utilized the same hyperparameters and settings for 1 GPU / 2 GPUs / 16 GPUs / 24 GPUs / 32 GPUs:

Deep learning framework: PyTorch 1.0.0 utilizing the NCCL backend and Shared file-system initialization.

Network architecture: ResNet-50

Training time: 90 epochs

Data loader threads per worker: 4

Per GPU worker batch size: 128

Learning rate: 0.1 (not scaled linearly)

No learning rate warmup period was applied

Momentum: 0.9

Weight decay: 1e-4

Data transforms and augmentations:

During training:

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

RandomResizedCrop(224)

RandomHorizontalFlip()

During validation:

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

Resize(256)

CenterCrop(224)

Learning rate schedule: The learning rate is reduced by 1/10 every 30 epochs

APPENDIX 6. SLURM BATCH SCRIPTS FOR EXPERIMENT 5

1 GPU baseline

Example of sbatch script for launching PyTorch Examples ImageNet training script on one cluster node while reserving one Pascal P100 GPU. The script trains for 10 epochs and saves the resulting model in a checkpoint file which is utilized for resuming training.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --tasks-per-node 1
#SBATCH --cpus-per-task 7
#SBATCH -D .
#SBATCH -J 1-node-1-gpu-pytorch-bs-128-lsr-baseline-checkpoint-0-10
#SBATCH -o 1-node-1-gpu pytorch-bs-128-lsr-baseline-checkpoint-0-10.out.%j
#SBATCH -e 1-node-1-gpu-pytorch-bs-128-lsr-baseline-checkpoint-0-10.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:1
#SBATCH -t 20:00:00
#SBATCH --mem=60G
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRK-DIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Launch training script.
srun python ./main.py -a resnet50 --print-freq 1 --batch-size 128 --lr "0.05" \
--world-size 1 --rank 0 --epochs 10 --workers 4 $TMPDIR/ilsvrc2012-torch/
```

1 GPU baseline continued

Example of sbatch script for launching PyTorch Examples ImageNet training script on one cluster node while reserving one Pascal P100 GPU. The script resumes training from a checkpoint file and trains for 10 epochs. The resulting model is saved in a checkpoint file which is utilized for resuming training. We repeat the restore from checkpoint and train procedure until we have trained for 90 epochs.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --tasks-per-node 1
#SBATCH --cpus-per-task 7
#SBATCH -D .
#SBATCH -J 1-node-1-gpu-pytorch-bs-128-lsr-baseline-checkpoint-10-20
#SBATCH -o 1-node-1-gpu-pytorch-bs-128-lsr-baseline-checkpoint-10-20.out.%j
#SBATCH -e 1-node-1-gpu-pytorch-bs-128-lsr-baseline-checkpoint-10-20.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:1
#SBATCH -t 20:00:00
#SBATCH --mem=60G
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRK-DIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Launch training script.
srun python ./main.py -a resnet50 --print-freq 1 --batch-size 128 --lr "0.05" \
--world-size 1 --rank 0 --epochs 20 --workers 4  \
--resume ./checkpoint.pth.tar $TMPDIR/ilsvrc2012-torch/
```

32 GPUs

Example of sbatch script for launching Horovod+PyTorch Examples ImageNet training script on eight cluster nodes while reserving thirty-two Pascal P100 GPUs. We scale the learning rate linearly but do not apply a learning rate warmup period.

```
#!/bin/bash
#SBATCH -N 8
#SBATCH --tasks-per-node 4
#SBATCH --cpus-per-task 7
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 8-node-32-gpu-pytorch-horovod-bs-128-lsr-no-wu
#SBATCH -o 8-node-32-gpu-pytorch-horovod-bs-128-lsr-no-wu.out.%j
#SBATCH -e 8-node-32-gpu-pytorch-horovod-bs-128-lsr-no-wu.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 10:00:00
#SBATCH --mem=0
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-torch-resized-new.tar -C $TMPDIR/

# Launch training script.
mpirun -np 32 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 pytorch-horovod-imagenet-no-warmup.py --bs 128 --lr 1.6 --epochs 90 \
      --data $TMPDIR/ilsvrc2012-torch/
```

APPENDIX 7. SETTINGS AND HYPERPARAMETERS FOR EXPERIMENT 5

Settings and hyperparameters for 1 GPU baseline:

Deep learning framework: PyTorch 1.0.0 utilizing the NCCL backend

Network architecture: ResNet-50

Training time: 90 epochs

Data loader threads per worker: 4

Per GPU worker batch size: 128

Linearly scaled learning rate for 1 GPU: 0.05

No learning rate warmup period was applied

Momentum: 0.9

Weight decay: 1e-4

Data transforms and augmentations:

During training:

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

RandomResizedCrop(224)

RandomHorizontalFlip()

During validation:

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

Resize(256)

CenterCrop(224)

Learning rate schedule: The learning rate is reduced by 1/10 at the 30-th, 60-th, and 80-th epoch

Settings and hyperparameters for 32 GPUs:

Deep learning framework: Horovod 0.15.2 + PyTorch 1.0.0

Network architecture: ResNet-50

Training time: 90 epochs

Data loader threads per worker: 4

Per GPU worker batch size: 128

Linearly scaled learning rate for 32 GPUs: 1.6

No learning rate warmup period was applied

Momentum: 0.9

Weight decay: 1e-4

Data transforms and augmentations:

During training:

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

RandomResizedCrop(224)

RandomHorizontalFlip()

During validation:

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

Resize(256)

CenterCrop(224)

Learning rate schedule: The learning rate is reduced by 1/10 at the 30-th, 60-th, and 80-th epoch

APPENDIX 8. SLURM BATCH SCRIPTS FOR EXPERIMENT 6

Example of sbatch script for launching Horovod+PyTorch Examples ImageNet training script on two cluster nodes while reserving eight Pascal P100 GPUs. The sbatch scripts for four cluster nodes sixteen GPUs, six cluster nodes twenty-four GPUs and eight cluster nodes thirty-two GPUs follow the same convention and can be derived from the example below.

```
#!/bin/bash
#SBATCH -N 2
#SBATCH --tasks-per-node 4
#SBATCH --cpus-per-task 7
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 2-node-8-gpu-pytorch-horovod-bs-128-lsr-stepping-wu
#SBATCH -o 2-node-8-gpu-pytorch-horovod-bs-128-lsr-stepping-wu.out.%j
#SBATCH -e 2-node-8-gpu-pytorch-horovod-bs-128-lsr-stepping-wu.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 24:00:00
#SBATCH --mem=0
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-torch-resized-new.tar -C $TMPDIR/

# Launch training script.
mpirun -np 8 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 pytorch-horovod-imagenet-stepping-warmup-py --bs 128 --lr 0.4 \
      --epochs 90 --data $TMPDIR/ilsvrc2012-torch/
```

Example of sbatch script for launching PyTorch Examples ImageNet training script on eight cluster nodes while reserving thirty-two Pascal P100 GPUs. The training script utilizes the PyTorch frameworks Shared file-system initialization feature to set up communication between the cluster nodes.

```
#!/bin/bash
#SBATCH -N 8
#SBATCH --tasks-per-node 1
#SBATCH --cpus-per-task 28
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 8-node-32-gpu-pytorch-bs-128
#SBATCH -o 8-node-32-gpu-pytorch-bs-128.out.%j
#SBATCH -e 8-node-32-gpu-pytorch-bs-128.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 10:00:00
#SBATCH --mem=0
#SBATCH --wait-all-nodes=1
#SBATCH

# PyTorch Shared file-system initialization file.
# Note! You must use the full to path to the file.
INITIALIZATION_FILE=/homeappl/home/nyholmju/INITIALIZATION_FILE
# Remove $INITIALIZATION_FILE on startup.
rm -f $INITIALIZATION_FILE

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRK-DIR/DONOTREMOVE/DATA/ilsvrc2012-tf.tar -C $TMPDIR/

# Since each node contains 4 GPUs, set the training scripts batch size to 512.
# Per GPU worker batch size = 512 / 4 = 128
# Start the job on node 0. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 0 python ./main-stepping-warmup.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "1.6" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 0 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# Start the job on node 1. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 1 python ./main-stepping-warmup.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "1.6" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 1 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &
```

```
# Start the job on node 2. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 2 python ./main-stepping-warmup.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "1.6" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 2 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# Start the job on node 3. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 3 python ./main-stepping-warmup.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "1.6" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 3 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# Start the job on node 4. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 4 python ./main-stepping-warmup.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "1.6" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 4 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# Start the job on node 5. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 5 python ./main-stepping-warmup.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "1.6" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 5 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# Start the job on node 6. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 6 python ./main-stepping-warmup.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "1.6" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 6 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# Start the job on node 7. Remember to add & since srun is a blocking command.
srun -lN1 -n 1 -c 28 -r 7 python ./main-stepping-warmup.py -a resnet50 --print-freq 1\
--batch-size 512 --lr "1.6" --dist-backend 'nccl' --multiprocessing-distributed \
--dist-url "file://$INITIALIZATION_FILE" --world-size 8 --rank 7 \
--epochs 90 --workers 4 $TMPDIR/ilsvrc2012-torch/ &

# ‘wait’ is needed when running jobs in background with &
wait
```

APPENDIX 9. SETTINGS AND HYPERPARAMETERS FOR EXPERIMENT 6

Settings and hyperparameters for 8 GPUs / 16 GPUs / 24 GPUs / 32 GPUs:

Deep learning frameworks: Horovod 0.15.2 + PyTorch 1.0.0 and PyTorch 1.0.0 utilizing the NCCL backend and Shared file-system initialization.

Network architecture: ResNet-50

Training time: 90 epochs

Data loader threads per worker: 4

Per GPU worker batch size: 128

Linearly scaled learning rate for **8 GPUs**: 0.4

Linearly scaled learning rate for **16 GPUs**: 0.8

Linearly scaled learning rate for **24 GPUs**: 1.4

Linearly scaled learning rate for **32 GPUs**: 1.6

5 epoch learning rate warmup period (stepping)

Momentum: 0.9

Weight decay: 1e-4

Data transforms and augmentations:

During training:

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

RandomResizedCrop(224)

RandomHorizontalFlip()

During validation:

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

Resize(256)

CenterCrop(224)

Learning rate schedule: The learning rate is reduced by 1/10 at the 30-th, 60-th, and 80-th epoch.

Pseudocode for the 5 epoch stepping learning rate warmup period.

```
warmup_epochs = 5
per_worker_learning_rate = 0.05
number_of_workers = 32
global_learning_rate =  number_of_workers * per_worker_learning_rate

for epoch in range(0, 90):
  if (epoch < warmup_epochs):
    learning_rate = global_learning_rate * ((epoch + 1) / (warmup_epochs + 1))
  else:
    learning_rate = global_learning_rate
```

APPENDIX 10. SLURM BATCH SCRIPTS FOR EXPERIMENT 7

Example of sbatch script for launching Horovod+PyTorch Examples ImageNet training script on eight cluster nodes while reserving thirty-two Pascal P100 GPUs. The per GPU worker batch size is set to 32.

```
#!/bin/bash
#SBATCH -N 8
#SBATCH --tasks-per-node 4
#SBATCH --cpus-per-task 7
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 8-node-32-gpu-pytorch-horovod-bs-32-lsr-stepping-wu
#SBATCH -o 8-node-32-gpu-pytorch-horovod-bs-32-lsr-stepping-wu.out.%j
#SBATCH -e 8-node-32-gpu-pytorch-horovod-bs-32-lsr-stepping-wu.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 10:00:00
#SBATCH --mem=0
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-torch-resized-new.tar -C $TMPDIR/

# Launch training script.
mpirun -np 32 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 pytorch-horovod-imagenet-stepping-warmup-py --bs 32 --lr 0.4 \
      --epochs 90 --data $TMPDIR/ilsvrc2012-torch/
```

APPENDIX 11. SETTINGS AND HYPERPARAMETERS FOR EXPERIMENT 7

Settings and hyperparameters for 32 GPUs with per GPU worker batch size 32:

Deep learning framework: Horovod 0.15.2 + PyTorch 1.0.0

Network architecture: ResNet-50

Training time: 90 epochs

Data loader threads per worker: 4

Per GPU worker batch size: 32

Linearly scaled learning rate for 32 GPUs: 0.4

5 epoch learning rate warmup period (stepping)

Momentum: 0.9

Weight decay: 1e-4

Data transforms and augmentations:

During training:

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

RandomResizedCrop(224)

RandomHorizontalFlip()

During validation:

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

Resize(256)

CenterCrop(224)

Learning rate schedule: The learning rate is reduced by 1/10 at the 30-th, 60-th, and 80-th epoch.

Pseudocode for the 5 epoch stepping learning rate warmup period:

```
warmup_epochs = 5
per_worker_learning_rate = 0.05
number_of_workers = 32
global_learning_rate =  number_of_workers * per_worker_learning_rate

for epoch in range(0, 90):
  if (epoch < warmup_epochs):
    learning_rate = global_learning_rate * ((epoch + 1) / (warmup_epochs + 1))
  else:
    learning_rate = global_learning_rate
```

APPENDIX 12. SLURM BATCH SCRIPTS FOR EXPERIMENT 8

Example of sbatch script for launching Horovod+PyTorch Examples ImageNet training script on four cluster nodes while reserving sixteen Pascal P100 GPUs. We scale the learning rate linearly and apply a 5 epoch gradual learning rate warmup period. The eight cluster nodes thirty-two GPUs and the six cluster nodes twenty-four GPUs sbatch scripts follow the same convention and can be derived from the four nodes sixteen GPUs example below.

```
#!/bin/bash
#SBATCH -N 4
#SBATCH --tasks-per-node 4
#SBATCH --cpus-per-task 7
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 4-node-16-gpu-pytorch-horovod-bs-128-lsr-gradual-wu-5
#SBATCH -o 4-node-16-gpu-pytorch-horovod-bs-128-lsr-gradual-wu-5.out.%j
#SBATCH -e 4-node-16-gpu-pytorch-horovod-bs-128-lsr-graduel-wu-5.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 13:00:00
#SBATCH --mem=0
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-torch-resized-new.tar -C $TMPDIR/

# Launch training script.
mpirun -np 16 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 pytorch-horovod-imagenet.py --bs 128 --lr 0.8 \
      --epochs 90 --num_warmup_epochs 5 \
      --data $TMPDIR/ilsvrc2012-torch/
```

APPENDIX 13. SETTINGS AND HYPERPARAMETERS FOR EXPERIMENT 8

Settings and hyperparameters for 16 GPUs / 24 GPUs / 32 GPUs:

Deep learning framework: Horovod 0.15.2 + PyTorch 1.0.0

Network architecture: ResNet-50

Training time: 90 epochs

Data loader threads per worker: 4

Per GPU worker batch size: 128

Linearly scaled learning rate for **16 GPUs**: 0.8

Linearly scaled learning rate for **24 GPUs**: 1.2

Linearly scaled learning rate for **32 GPUs**: 1.6

5 epoch learning rate warmup period (gradual)

Momentum: 0.9

Weight decay: 1e-4

Data transforms and augmentations:

During training:

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

RandomResizedCrop(224)

RandomHorizontalFlip()

During validation:

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

Resize(256)

CenterCrop(224)

Learning rate schedule: The learning rate is reduced by 1/10 at the 30-th, 60-th, and 80-th epoch.

Pseudocode for 5 epoch gradual learning rate warmup period:

```
warmup_epochs = 5
per_worker_batch_size = 128
linearly_scaled_learning_rate = 1.6
base_learning_rate = (per_worker_batch_size / 256) * 0.1
step_size = (linearly_scaled_learning_rate - base_learning_rate) / warmup_epochs

for epoch in range(0, 90):
  if (epoch < warmup_epochs):
    learning_rate = base_learning_rate + (step_size * epoch)
  else:
    learning_rate = linearly_scaled_learning_rate
```

APPENDIX 14. SLURM BATCH SCRIPTS FOR EXPERIMENT 9

Example of sbatch script for launching Horovod+PyTorch Examples ImageNet training script on six cluster nodes while reserving twenty-four Pascal P100 GPUs. We scale the learning rate linearly and apply a 10 epoch gradual learning rate warmup period. The eight cluster nodes thirty-two GPUs sbatch script follows the same convention and can be derived from the six cluster nodes twenty-four GPUs example below.

```
#!/bin/bash
#SBATCH -N 6
#SBATCH --tasks-per-node 4
#SBATCH --cpus-per-task 7
#SBATCH --exclusive
#SBATCH -D .
#SBATCH -J 6-node-24-gpu-pytorch-horovod-bs-128-lsr-gradual-wu-10
#SBATCH -o 6-node-24-gpu-pytorch-horovod-bs-128-lsr-gradual-wu-10.out.%j
#SBATCH -e 6-node-24-gpu-pytorch-horovod-bs-128-lsr-graduel-wu-10.err.%j
#SBATCH -p gpu
#SBATCH --gres=gpu:p100:4
#SBATCH -t 12:00:00
#SBATCH --mem=0
#SBATCH

# Create directory on the nodes to hold the training data.
pdsh -w $SLURM_JOB_NODELIST mkdir -p $TMPDIR/

# Extract training data on nodes.
srun --ntasks=$SLURM_NNODES --ntasks-per-node=1 tar xf \
$WRKDIR/DONOTREMOVE/DATA/ilsvrc2012-torch-resized-new.tar -C $TMPDIR/

# Launch training script.
mpirun -np 24 -bind-to none -map-by slot \
      -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH \
      -mca pml ob1 -mca btl ^openib \
      -oversubscribe \
      python3.6 pytorch-horovod-imagenet.py --bs 128 --lr 1.2 \
      --epochs 90 --num_warmup_epochs 10 \
      --data $TMPDIR/ilsvrc2012-torch/
```

APPENDIX 15. SETTINGS AND HYPERPARAMETERS FOR EXPERIMENT 9

Settings and hyperparameters for 24 GPUs / 32 GPUs:

Deep learning framework: Horovod 0.15.2 + PyTorch 1.0.0

Network architecture: ResNet-50

Training time: 90 epochs

Data loader threads per worker: 4

Per GPU worker batch size: 128

Linearly scaled learning rate for **24 GPUs**: 1.2

Linearly scaled learning rate for **32 GPUs**: 1.6

10 epoch learning rate warmup period (gradual)

Momentum: 0.9

Weight decay: 1e-4

Data transforms and augmentations:

During training:

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

RandomResizedCrop(224)

RandomHorizontalFlip()

During validation:

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

Resize(256)

CenterCrop(224)

Learning rate schedule: The learning rate is reduced by 1/10 at the 30-th, 60-th, and 80-th epoch.

Pseudocode for 10 epoch gradual learning rate warmup period:

```
warmup_epochs = 10
per_worker_batch_size = 128
linearly_scaled_learning_rate = 1.6
base_learning_rate = (per_worker_batch_size / 256) * 0.1
step_size = (linearly_scaled_learning_rate - base_learning_rate) / warmup_epochs

for epoch in range(0, 90):
  if (epoch < warmup_epochs):
    learning_rate = base_learning_rate + (step_size * epoch)
  else:
    learning_rate = linearly_scaled_learning_rate
```