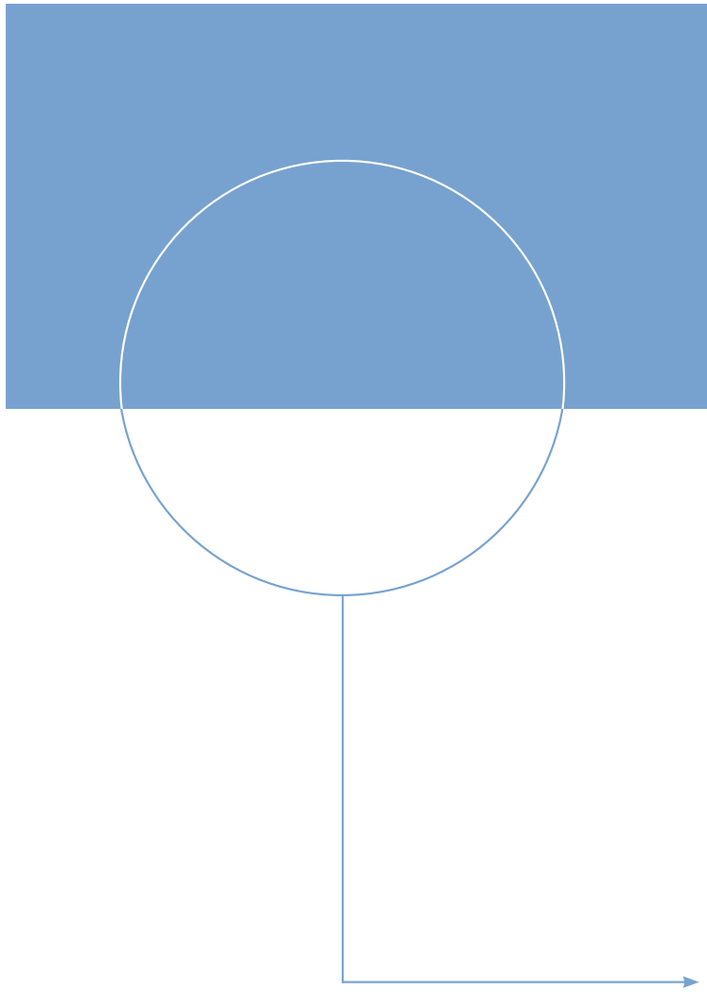


Master's thesis

2020

André Håland, Bjørnar Birkeland

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

André Håland
Bjørnar Birkeland

Exploring data assignment schemes when training deep neural networks using data parallelism

June 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Exploring data assignment schemes when training deep neural networks using data parallelism

André Håland
Bjørnar Birkeland

Computer Science
Submission date: June 2020
Supervisor: Ole Jakob Mengshoel
Co-supervisor: Lorenzo Cevolani, Graphcore
Arjun Chandra, Graphcore

Norwegian University of Science and Technology
Department of Computer Science

Abstract

The ever increasing sizes of datasets have allowed deep neural networks to excel in many difficult tasks. In addition, even bigger models have proven to improve the performance of deep neural networks. However, the huge amount of data in combination with bigger models have resulted in the training process becoming prohibitively computationally expensive for a single worker. As such, the training process is parallelized across several workers.

Through a literature review, we find that previous work mainly focus on the model, its hyperparameters and the communication method when scaling to large number of workers. We also find that there, to the best of our knowledge, does not exist any study on how the amount of data available to each worker affects the final accuracy. Thus, in this thesis, we explore the effects of different data assignment schemes when training deep neural networks using data parallelism. We find that when training fully synchronous, there is no significant difference in final accuracy between the amount of data available to each worker. When reducing the number of communication rounds, however, we find that when the batch size/learning rate relationship is altered to a certain degree, assigning overlapping data can improve the final accuracy, compared to assigning non-overlapping data.

Sammendrag

Den stadig økende størrelsen på datasett har gjort det mulig for dype nevrale nettverk å utføre mange vanskelige oppgaver. Samtidig har enda større modeller vist seg å forbedre ytelsen til dype nevrale nettverk. Derimot har den enorme mengden datainnsamling med større modeller ført til at treningen har blitt uoverkommelig beregningsdyktig for en enkelt arbeider. Som sådan blir treningen parallellisert over flere arbeidere.

Gjennom et litteraturstudie finner vi at tidligere arbeid hovedsakelig fokuserer på modellen, dens hyperparametre og kommunikasjonsmetoden når det skaleres til et stort antall arbeidere. Vi finner også at det, etter vår kunnskap, ikke eksisterer noen undersøkelse av hvordan datamengden tilgjengelig for hver enkelt arbeider påvirker nøyaktigheten. I denne oppgaven undersøker vi derfor effektene av forskjellige datafordelingstrategier når vi trener dype nevrale nettverk ved bruk av data parallelisme. Vi finner ingen signifikant forskjell i nøyaktighet mellom datamengden tilgjengelig for hver enkelt arbeider når vi synkroniserer i hvert steg. Når vi reduserer antall kommunikasjonsrunder, finner vi imidlertid at tildeling av overlappende data kan forbedre nøyaktigheten når partistørrelse/læringsrate forholdet er forskjøvet til en viss grad, sammenlignet med å tildele ikke-overlappende data.

Preface

This thesis is the product of work done as part of the subject TDT4900 at Norwegian University of Science and Technology (NTNU) during the spring of 2020. The thesis is a continuation on the work done during the autumn of 2019 as part of the subject TDT4501, in which we conducted a literature review in the field of training deep neural networks using data parallelism.

We would like to thank our supervisor Ole Jakob Mengshoel for his general guidance and follow-up meetings throughout this project. We would also like to thank our supervisors at Graphcore, Lorenzo Cevolani and Arjun Chandra, for their guidance regarding the more technical material of the project.

André Håland & Bjørnar Birkeland

Trondheim, June 10, 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine learning and especially deep learning has in recent years excelled at previously difficult tasks like speech recognition [1], image classification [2] and language processing [3]. However, with the increasing sizes of training data [4, p. 18-21], training becomes slower and in some cases infeasible on a single worker. Similarly, to increase model accuracy, a popular approach is to increase the model size, slowing down the training further. Lately, there has been a trend in research where the usage of CPUs for training neural networks has gradually shifted towards parallel accelerators [5], enabling more efficient training. Accelerators have become increasingly effective, and some manufacturers even create chips specifically designed for machine learning tasks [6], [7]. However, the massive amount of compute required to train state-of-the-art models has increased exponentially the last few years [8], rendering training with a single accelerator in reasonable time insufficient. Going forward, machine learning is going to require fast training of massive models in a way that optimizes both the accuracy of the model and the latency.

1.1 Background and Motivation

Parallelization is a natural approach to speed up the training process. Among the popular methods for parallelizing deep neural networks are model parallelism and data parallelism [9]. With model parallelism, the work is divided by the model layers. This means that each worker will have a subset of the entire model,

where training is accomplished by passing layer activations between the workers. This is beneficial whenever the model is too large to fit in memory of a single worker. Data parallelism, on the other hand, splits the work by splitting the data between the workers, where each worker has a copy of the entire model. The workers update their local models by regularly synchronizing updates.

There are two main challenges when training deep neural networks using data parallelism: (1) maintaining accuracy when the global batch size increases as an effect of scaling the number of workers, and (2) overcoming the communication bottleneck that occurs with a large number of workers. We see that a reoccurring theme in the literature for dealing with these challenges is to adjust the hyperparameters of the model when increasing the batch size, modifying parts of the model and improving the communication method to overcome the communication bottleneck. At the same time, we see that most work assign the data disjointly (*i.e.,* no overlap) between the workers with no explicit statement of how this impact the performance. Also, to the best of our knowledge, there does not exists any study of how assigning data in different ways between the workers impact the performance of training deep neural networks using data parallelism. We see this concept as interesting because it can be applied to data parallel supervised learning in general. Therefore, if an effect of overlapping data is observed with regards to model accuracy using a specific machine learning architecture and dataset, then the same effect might occur using other machine learning architectures and datasets as well.

1.2 Goals and Research Questions

We see the presented motivation in Section 1.1 as an opportunity to explore the effects of different data assignment schemes, which give rise to our research goal:

Goal *Explore different data assignment schemes and the effects of them when training deep neural networks using data parallelism*

The specifics of different data assignment schemes will be presented in Section 4.1. Further, to reach our research goal, we have created three research questions, seen below. We refer to Section 2.2.7 for a more detailed discussion on how to evaluate the performance of a data parallel system.

Research question 1 (RQ1) *In terms of performance, what are the effects of*

assigning the data in different ways between the workers?

Research question 2 (RQ2) *Can overlapping data make the system more resilient to communication reduction? If so, in what circumstances?*

Research question 3 (RQ3) *What is the optimal way to assign data between workers?*

1.3 Research Method

The work in this thesis is a continuation of a literature review performed as part of the subject TDT4501 - Computer Science, Specialization Project in the autumn of 2019. The main findings, as has been mentioned previously in this chapter, lead us to the derivation of a research goal and a set of research questions. To address the goal, most of the work in this thesis will be experimental and analytical work. We will use a known deep neural network architecture to establish a baseline, and further use the workload (architecture, dataset, optimizer and hyperparameters) in this baseline with minor modifications to address our goal. The typical process will be to conduct experiments, analyse them, and design new experiments based on the analysis. Since our research goal is to explore a certain subfield within machine learning, and the fact that machine learning is an empirical field of study, we see this as the natural choice of research method.

1.4 Thesis Structure

This rest of this report are structured as follows:

- Chapter 2 presents necessary background theory for the rest of this report. This includes a brief description of deep learning, as well as a more in depth look at using data parallelism for training deep neural networks.
- Chapter 3 takes a look on work that is related to this project.
- Chapter 4 presents a high-level description of the main concepts experimented with in this project. This chapter also presents the scope of the project and a description of how we conduct experiments.

- Chapter 5 presents the results for experiments conducted in this project. Every experiment will be presented together with a specific goal, the method and data used, as well as a discussion of the results.
- Chapter 6 concludes this project by evaluating the key results, and answering the research questions stated in Section 1.2. This chapter will also include a discussion of limitations in this work, leading to suggestions for future work.

Note that Chapter 1, Chapter 2 and Chapter 3 are based on but revised from chapters in the specialization project [10].

Chapter 2

Background Theory

In this chapter we will provide sufficient background theory to understand how data parallelism can be used to speed up the training of a deep neural network (DNN). We will first see a typical construction of a DNN, as well as a common method for training them. Then, we will describe how a typical data parallel training process is performed. This includes a look at main design features and how they impact performance and runtime.

2.1 Deep Learning

In this section we will see what constitutes the main building blocks of an artificial neural network (ANN), and how these building blocks, when arranged in multiple layers, forms a DNN. We will also see how training these networks can be seen as an optimization problem. At last, we will describe a special kind of ANN, called convolutional neural network (CNN), which is commonly used to extract features from data with spatial information.

2.1.1 Artificial Neural Networks

An ANN is a network of artificial neurons normally organized in layers. Figure 2.1 illustrates an artificial neuron. The output of one layer becomes the input

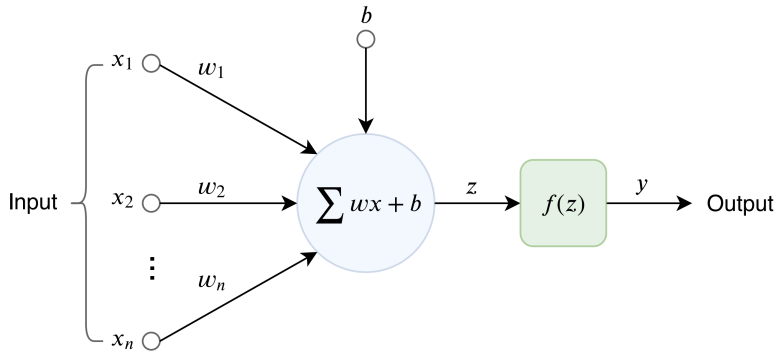


Figure 2.1: An artificial neuron. It receives an array of inputs, where each input x_i is multiplied with the corresponding weight w_i. Then the weighted inputs are summed with a bias b and the result z is fed through the activation function f, creating the output y. Some notable activation functions are ReLU, sigmoid, and tanh, where ReLU is currently the most popular [11].

of the next layer, where the first layer, the input layer, consists of the input data. This system creates a function when passing through the layers, known as the forward propagation function [4, p. 200]. When all units in one layer is connected to every unit in the next layer, the layers are fully connected. Figure 2.2 displays a network where every layer is fully connected. Between the input layer and the last layer, the output layer, there may be one or several layers called hidden layers. Modern ANNs typically have several hidden layers, and empirical results show that deeper networks generalize better [4, p. 194-200]. The popular terms "deep neural network", or "deep learning" refers to ANNs that have several hidden layers.

2.1.2 Training Artificial Neural Networks

When training an ANN, the weights and biases, called parameters, are adjusted in a way such that the forward propagation function approximates a goal function. This task is an optimization problem, as we want to find the parameters w that minimize the distance between the goal function and the ANN. When we have a set of data samples with the corresponding goal output, this set can be used for training by applying our ANN to each data sample and calculating the distance between the prediction made by our ANN and the goal output using a loss function. This process is described as supervised learning, and can be

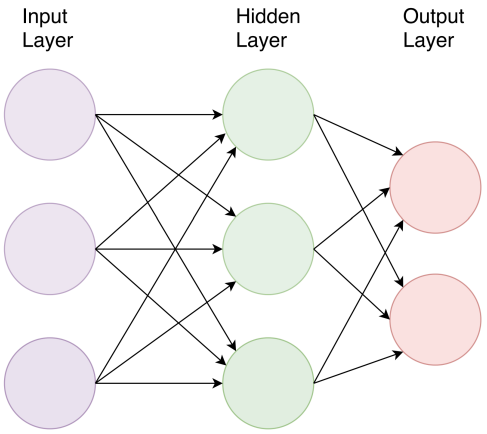


Figure 2.2: A fully connected artificial neural network. The network consists of an input layer, a single hidden layer and an output layer. Each artificial neuron in one layer is connected to every neuron in the next layer, making the artificial neural network fully connected.

described by

$$\min_{w \in R^d} L(w) \text{ where } L(w) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^{n} L_i(w). \tag{2.1}$$

The loss function $L(w)$ is the approximated distance between the goal function and the function achieved with our ANN with current parameters w. This loss is found by calculating the mean loss of each data sample in a dataset with n data samples. Some examples of loss functions used with ANNs are mean squared error for use in regression, and cross-entropy for classification. The optimization problem can be approached in several ways, including: first-order optimization, second-order optimization, or search using evolutionary algorithms, where the popular way of optimizing is first-order optimization using a variant of gradient descent. Gradient descent iteratively adjusts the parameters w in order to minimize a loss function. In each iteration the adjustment is proportional to the gradient of the loss function with respect to the parameters at the current iteration t. This gradient is often calculated using the backpropagation algorithm [12]. In addition, the adjustment is scaled by a learning rate η. This equation is given by

$$w_{t+1} = w_t - \eta \nabla L(w_t), \tag{2.2}$$

where $\nabla L(w_t)$ is the gradient of the loss function. Computing the gradient of the loss function for the whole dataset for each parameter adjustment can be

prohibitively computationally expensive. One way to tackle this problem is to use stochastic gradient descent (SGD) [13], where instead of computing the gradient based on the whole dataset, it is approximated using a randomly selected subset b of the dataset. This technique is also known as mini-batch stochastic gradient descent. The equation for this variant is given by

$$w_{t+1} = w_t - \eta \nabla L_b(w_t) \text{ where } \nabla L_b(w_t) = \frac{1}{b} \sum_{i=1}^{b} \nabla L_i(w_t). \tag{2.3}$$

Here, $\nabla L_b(w_t)$ is the gradient of the loss function with respect to the parameters w at time t, calculated using a batch with b samples. Other notable variations of SGD include momentum [12], RMSProp [14], Adam [15] and LAMB [16].

2.1.3 Convolutional Neural Networks

The CNN [17] is similar to the ANN described in Section 2.1.1, as they are composed of neurons structured in layers, and can still be trained using SGD. Some notable differences between an ANN using fully-connected layers and a CNN are the usage of convolutional layers and pooling layers. In a convolutional layer, neurons are structured as a set of filters. When it is applied to a black and white image, the image is represented as a 2D matrix. This matrix is convolved with each filter in the layer, creating a separate matrix for each filter. Then an activation function is applied to each of these matrices, typically ReLU, resulting in a set of activation maps; the activation volume. This operation is shown in Figure 2.3. The number of activation maps that make up the activation volume is referred to as its depth. In subsequent convolutional layers, the entire activation

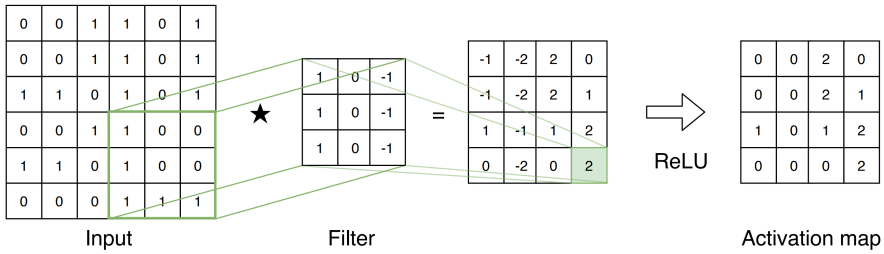


Figure 2.3: The convolution operation. Here a single filter of size 3x3 containing 9 weights is convolved with the input matrix. After this, the ReLU activation function is applied, resulting in an activation map.

volume is convolved with each filter, creating only one activation map for each

filter. In between a sequence of convolutional layers it is common to use a pooling layer. Pooling layers reduce the dimensions of an activation map by combining neighbouring values into single values. This is done for each activation map in the volume, and as such, the depth of the activation volume is not affected. The most popular way of combining these values is to use max-pooling, in which only the highest value is kept. Despite often being referred to as a layer, the pooling layer does not have any parameters that requires training.

CNNs perform particularly well on image processing [2] and other natural signals. This is due to a few aspects. First, the convolutional layer use local connections that take advantage of the spatial information found in images. Second, the convolutional layers use shared weights. This is helpful as it decreases the number of parameters that needs to be trained, and weights that are able to detect certain patterns in one part of an image can be used to find the same pattern in different areas in an image [11]. Third, by reducing the activation dimensions, the pooling layer introduces invariance to translation to the network, as the output of the layer is less dependent of the exact position of a feature [4, p. 335-339]. In addition, by reducing the size of the activation map, this layer also reduces the amount of compute required in subsequent layers.

2.2 Data Parallelism

In this section we will describe the parallelization strategy in which the work is split across the data dimension, namely *data parallelism* [9]. For M workers, this parallelization strategy can be illustrated as seen in Figure 2.4. One of the advantages of data parallelism is that it is model agnostic, meaning that it is applicable to any machine learning architecture. Since the workers in a data parallel system must synchronize their computed gradients or parameters with each other, data parallelism is especially beneficial for models with high compute and fewer parameters (*e.g.,* CNNs) [18]. When scaling to multiple workers, however, there are mainly two challenges with data parallelism. The first challenge is related to the runtime of the system. When the communication-to-computation ratio gets large, the desired speedup of parallel training is degraded. This can happen for instance when the total number of workers increases or when the size of the model parameters increases. The second challenge is related to the performance of the system. As the global batch size often increases with the number of workers, the model is vulnerable to the generalization gap [19] (see Section 2.2.1). For the rest of this section, we will describe some design features for a data parallel system and how they impact runtime and performance.

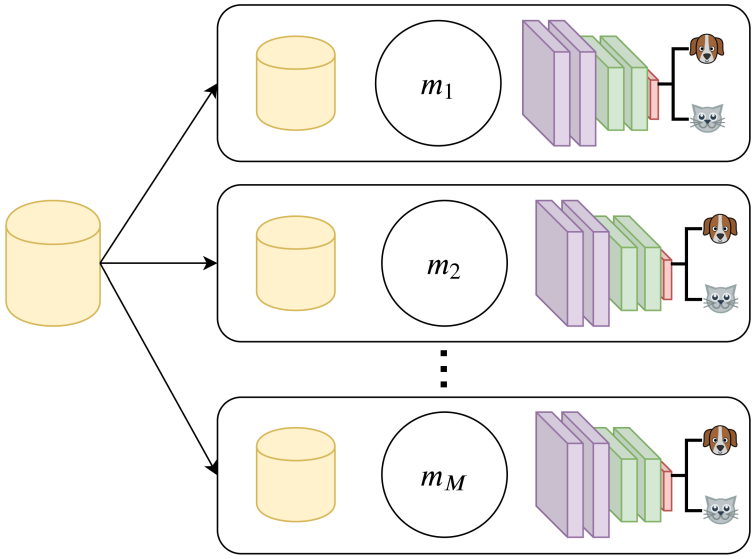


Figure 2.4: Data parallelism. In a data parallel system, the model is replicated across multiple workers, and each worker use some part of the dataset to train. In the general case, each worker computes gradients using a part of their allocated data and shares either the model parameters or gradients with the other workers to update the local models.

2.2.1 Global- and Local Batches

Data parallel training of DNNs is typically done by defining a *global batch* with size B, and letting every worker compute the gradient for a *local batch* with size $b = B/M$. When adding workers to the system, one could either increase the global batch size or decrease the local batch size (or a combination of both). With today's highly parallel hardware, it is desirable to use a large enough b to utilize the computational resources at every worker. However, when using large batches, DNNs tend to converge to sharp minimizers of the training function which leads to the generalization gap [19], meaning it performs worse on unseen data (typically on a held-out test set) than the training data. For models that incorporate Batch Normalization (BatchNorm) [20], we also see a drop in accuracy when the local batch size gets too small.[1] Wu *et al.* [21] note that this can be explained by the inaccurate batch statistics estimation, and show that when training ResNet-

[1] Exactly how small will depend on the dataset.

50 [22] on ImageNet [23], the accuracy starts to drop when the local batch size falls below 16. They further propose Group Normalization as a solution to this problem. With Group Normalization, the channels are divided into groups, where the mean and variance for each group are used for normalization. Due to its independence on the batch dimension, Group Normalization does not suffer with small local batch sizes.

2.2.2 Centralization

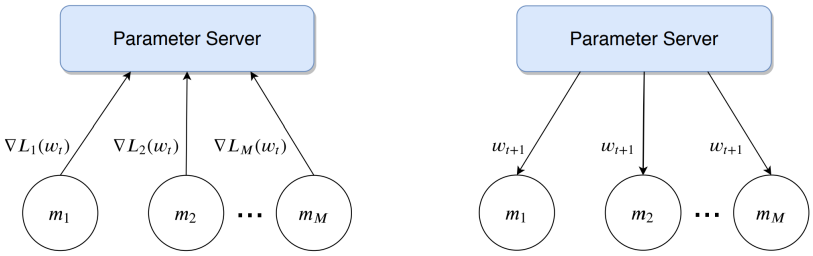
A central data parallel system consists of a centralized server, often referred to as a *parameter server* [9], [24].[2] The parameter server contains the global parameter, and communicates with all workers to update this parameter. A typical workflow consist of the workers *pushing* calculated gradients/parameters to the parameter server, the parameter server updating the global parameter, followed by the workers *pulling* the newest parameters. The push and pull operations for a synchronous (see Section 2.2.3) parameter server are illustrated in Figure 2.5a and 2.5b, respectively. The push operation consists of M workers calculating the local gradients $\nabla L_1(w_t), \nabla L_2(w_t), \ldots, \nabla L_M(w_t)$ at time t and sending them to the parameter server. After the parameter server has received the gradients from all of the workers, it updates the global parameter

$$w_{t+1} = w_t - \eta \frac{1}{M} \sum_{i=1}^{M} \nabla L_i(w_t), \tag{2.4}$$

where η is the learning rate. In the pull operation, every worker pulls down this new global parameter. Parameter servers can also be used to implement asynchronous systems (see Section 2.2.3): after a worker has sent its local gradient/parameter, the parameter server immediately updates the global parameter and sends it to the worker. As we will see in Section 2.2.3, this will result in staleness.

In a decentralized system there is no central server. This means that the workers must communicate with each other to achieve a shared global parameter. MapReduce [25] is a popular distributed data processing model, but it has been observed that it is not well suited for iterative problems as often found in machine learning [26]. Moreover, a common approach to achieve a global parameter with decentralized data parallelism is by aggregating the local gradients with the use of high-performance communication interfaces, such as Message Passing Interface (MPI) [27]. The most common MPI operation used in decentralized data parallel

[2] Does not have to be a single machine. Can for instance be a sharded server [9].



(a) Push operation (b) Pull operation

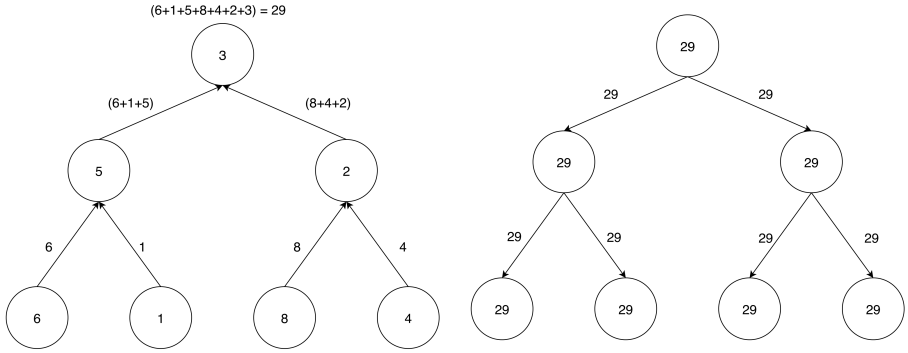
Figure 2.5: A parameter server. It contains the global parameters. Workers can push (Figure 2.5a) their new parameters to the parameter server where they are aggregated, updating the global parameters. The new parameters can then be pulled (Figure 2.5b) by the workers to update the parameters in each worker's local model.

systems is *AllReduce*: every worker i starts with a local gradient $\nabla L_i(w_t)$, and ends up with the sum of all local gradients across all workers. This sum can then be divided by the number of workers M to get the average gradient

$$\nabla L(w_t) = \frac{1}{M} \sum_{i=1}^{M} \nabla L_i(w_t). \tag{2.5}$$

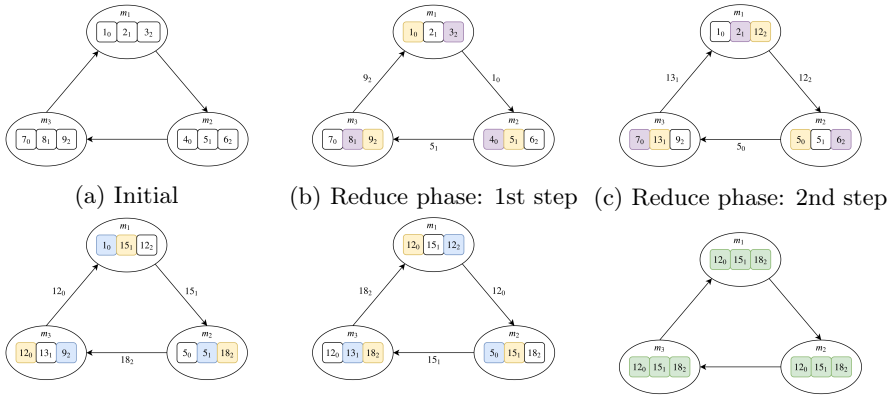
The AllReduce operation can for instance be implemented in a tree structure as illustrated in Figure 2.6. Another AllReduce variant is the Ring-AllReduce [28] shown in Figure 2.7. Here, a vector of numbers is split into M chunks. As seen in the figure, since there are three workers, the vector is split into three parts.[3] The chunks are sent through the ring until every worker contains one chunk that is summed across all workers (Figure 2.7b and 2.7c). For example, in Figure 2.7d, worker m_1 has the complete sum of index 1, m_2 of index 2 and m_3 of index 0. With M workers, this phase requires $M - 1$ steps. When this phase is complete, the sums are broadcast around the ring to ensure that every worker contains the complete sum for all chunks (Figure 2.7d and 2.7e). This phase also requires $M - 1$ steps, resulting in a total of $2(M - 1)$ steps for the entire procedure.

[3] As with Tree-AllReduce, the chunks will in most cases contain more than a single number, and hence, a element-wise addition will be used instead of scalar addition.



(a) Reduce (b) Broadcast

Figure 2.6: Tree-AllReduce. In the reduce phase (Figure 2.6a), the aggregated numbers are sent from the leaf nodes and upwards in the tree until the root node has received all aggregated numbers. The root node then calculates the total sum, before it is broadcast down the tree (Figure 2.6b). Note that even though the illustrations use single numbers, this operation can be applied to vectors of numbers with the use element-wise addition.



(a) Initial (b) Reduce phase: 1st step (c) Reduce phase: 2nd step

(d) Broadcast phase: 1st step (e) Broadcast phase: 2nd step (f) Final

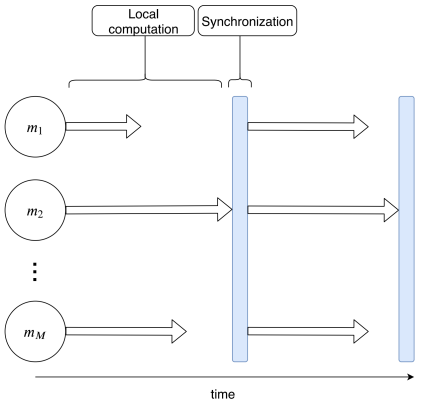
Figure 2.7: Ring-AllReduce for 3 workers. The yellow boxes indicate numbers to be sent in current step. For the first two steps: purple boxes represents numbers that will be *aggregated* with a received number. For the last two steps: blue represents numbers that will be *replaced* by a received number

2.2.3 Synchronization

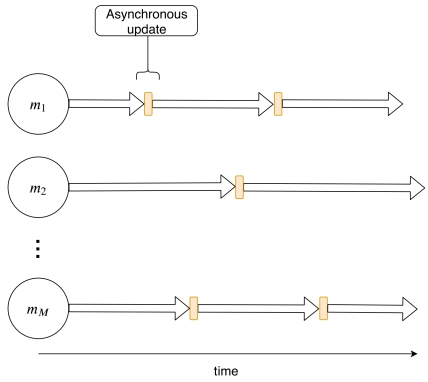
Another design choice for a data parallel system is its level of synchronization. In a fully synchronous system the workers must wait for every other worker to finish their local computations so that the calculated parameters or gradients can be exchanged, as shown in Figure 2.8a. In such a system, the up-to-date parameters are observed by everyone, which we refer to as a *consistent model*. In a fully asynchronous system, on the other hand, the workers can continue their computations independently of other workers process, leading to an *inconsistent model*. This is shown in Figure 2.8b. A data parallel system can also be somewhere in between these two extremes. For instance, the system may only allow the slowest worker be a maximum amount of steps behind the fastest worker. This approach is referred to as stale-synchronous parallelism [29] and is shown in Figure 2.8c. This figure shows that the workers can do asynchronous updates until the max staleness (which in the figure is 2) is reached. The system must then wait for all workers to finish its local computation before a global synchronization is performed.

With a fully synchronous system, the *straggler problem* [30] is introduced. This happens as a result of a small amount of the workers taking longer to finish a given task. The majority of the workers must then wait for the slowest worker before they can continue with the next task, resulting in low utilization. In an asynchronous system, the model parameters can be updated without any synchronization, solving the straggler problem. However, this means that a worker can be computing with outdated parameters. The gradients calculated with the outdated parameters are called *stale* gradients, and its *staleness* is defined as the number of updates that have happened to the global parameters since start of computation at the local worker [31]. More formally, a worker i at time t has a copy of the parameters w_i^τ with $\tau \leq t$, where $t - \tau$ is the staleness of m. Performance is shown to degrade as staleness increases [31].

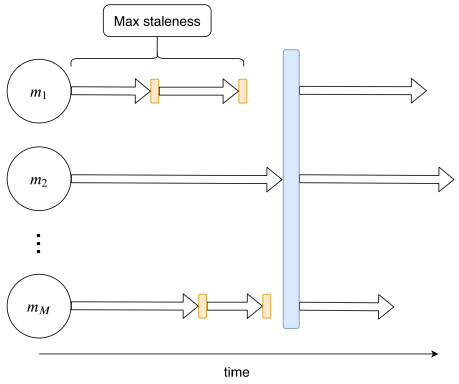
When doing synchronous data parallelism, the local parameters [32], [33] or the computed local gradients [9] must be averaged between all workers to obtain a consistent model. We will refer to these two cases as *parameter averaging* and *gradient averaging*, respectively. In the case of gradient averaging in a centralized system, the workers send their gradients directly to the parameter server, followed by the parameter server aggregating all gradients and then using it to update the global parameter (described by Equation (2.4) and illustrated in Figure 2.5). The other alternative for a centralized system is for every worker to use their gradients to update their local parameters and then send these newly calculated parameters to the parameter server. The parameter server can then average all



(a) Synchronous data parallelism



(b) Asynchronous data parallelism



(c) Stale-synchronous data parallelism

Figure 2.8: Synchronization. In this figure, the arrows represent local computation, the blue rectangle represent a global synchronization between all workers, and the yellow rectangles represent an asynchronous update. Synchronous (Figure 2.8a) and asynchronous (Figure 2.8b) can be viewed as two extremes, where stale-synchronous (Figure 2.8c) is somewhere in between. In Figure 2.8c the max staleness is 2.

the local parameters to obtain a new global parameter.

One also has to consider the frequency of the synchronization. Averaging the gradients after every batch has been calculated is referred to as *mini-batch aver-*

aging [34].[4] With the assumption that every worker has disjoint partitions of the dataset (see Section 4.1), mini-batch averaging is conceptually similar to training on a single worker. Another approach is to allow every worker optimize the objective function locally, and then average all parameters at the end of optimization. This approach is referred to as *one-shot averaging* [35], [36]. A middle ground of one-shot averaging and mini-batch averaging is *local SGD* [37], [38]. With local SGD, each worker runs independently for a certain amount of iterations before the parameters are averaged between the workers. A variant to local SGD is to have more frequent averaging at the initial phase of training [34]. Previous work on local SGD will be further studied in Chapter 3.

The effects of the synchronization frequency is twofold. Naturally, the frequency has an impact on the communication in the system. More frequent synchronization result in more communication, thus decreasing the computation-to-communication ratio. On the other hand, less frequent synchronization could negatively impact the model performance. For certain non-convex problems (such as DNNs), Zhang *et al.* [34] illustrate that one-shot averaging can negatively impact the accuracy. The intuition is that workers could end up converging to different local minima with the average of the minima not being a minimum. This is illustrated for a single-value parameter w in Figure 2.9 where the x-axis consist of the parameter and the y-axis is the loss value with this parameter. They further show that more frequent averaging can be used to regain the accuracy, and conclude that one-shot averaging is not suitable for non-convex problems.

There also exist some methods that extends direct parameter averaging. One of them is *elastic averaging* [39], an averaging method where the workers use an elastic force based on a global parameter stored at a parameter server to update their local parameters. This enables the workers to perform more exploration without fluctuating too much from the global parameter. The global parameter is updated as a moving average of the parameters computed by the workers. Another method, called *gossip averaging* [40], allows the amount of information exchanged between the workers to be tuned. After updating local parameters, each worker draws a random Bernoulli variable with expectancy p that decides whether the worker will share its information with another uniformly drawn worker. This implies that at each round, every worker will send their parameters at most once, but can receive parameters from several others. Higher p results in the workers having more similar weights, but on the other hand requires more communication. A lower p means less communication, but could result in the workers diverging.

[4] Note that gradient averaging only makes sense in the case of mini-batch averaging.

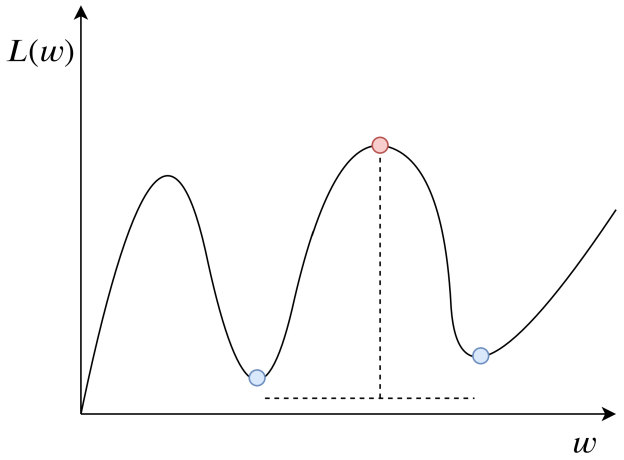


Figure 2.9: Average of two local minima. Blue dots represent the local minima, while the red dot represent the average of the minima

2.2.4 Quantization and Sparsification

Since large scale distributed systems are often limited by a communication bottleneck, it is desirable to reduce the communication. This can for instance be accomplished by limiting the frequency of synchronization, as described in Section 2.2.3. Researchers has also studied other methods to reduce communication. Giving the gradient a lower precision with *gradient quantization* is one such method. By sending the quantization errors from one gradient quantiziation of one batch to the next batch and adding it to the gradient before quantization, Seide *et al.* [41] was able to quantize the gradients to 1-bit per value with nearly no loss in accuracy in their experiments. Quantized Stochasitc Gradient Descent [42] allow users to trade of accuracy and runtime by adjusting the precision of the quantization. Another method for reducing communication is by limiting which parts of the gradient that are communicated with *gradient sparsification*. Heuristics are often used to decide which parts should be sent. For instance, Strom [43] uses a threshold in which only the gradient elements larger than the specified threshold will be communicated. With *gradient dropping* [44], a dropping rate R is used to drop the $R\%$ smallest gradient elements by absolute value. Common for these two sparsification methods is the local accumulation of gradient elements that were not included in the communicated gradient. When the accumulated gradient elements gets larger than the threshold or in the $(1 - R)\%$ biggest gradient elements, they will be communicated.

2.2.5 Sampling

When using SGD to train a neural network, there are essentially two strategies for sampling the data. The first strategy, called *with-replacement*, picks data samples from the entire dataset at random and computes the gradient which is used to update the parameters. The samples are then placed back into the dataset and the process is repeated. The other strategy is called *without-replacement*: all data samples are put into a pool, and whenever a data sample has been used to update the parameters, this data sample is removed from the pool. When all samples in the pool has been processed, an *epoch* is completed. The number of iterations in an epoch is N/B where N is the number of data samples in the dataset and B is the batch size.[5] When an epoch is completed, all samples are put back into the pool and the process is repeated.

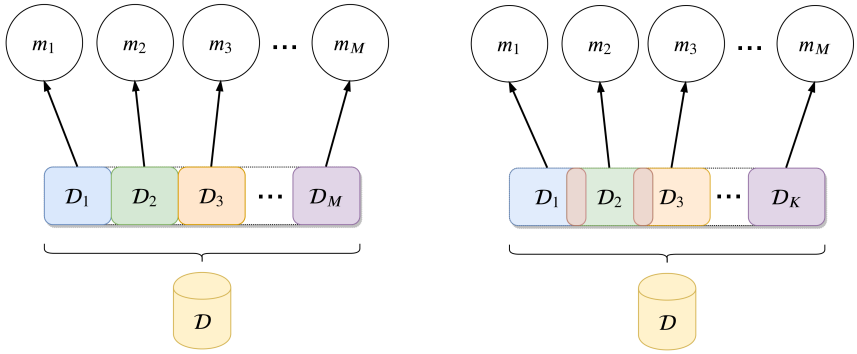
There are, however, variants [45] of without-replacement that should be discussed. These variants are concerned with whether or not the dataset is shuffled before it is put into the pool before an epoch. One approach is to shuffle the dataset only before the first epoch and then go through the samples in the same sequence for every epoch. Another approach is to shuffle the dataset before every epoch. This implies that the batches will, with very high probability when $B \ll N$, contain different samples in every epoch. When it comes to distributed SGD, there are even more variants [46] to without-replacement. Since the data is assigned between multiple workers, the data can be shuffled either locally or globally. With *local shuffling*, the workers only shuffle their locally assigned data, while with *global shuffling*, the entire dataset is shuffled before it is redistributed to the workers.

2.2.6 Data assignment

A less discussed part of data parallelism is how to assign the data between the workers. For practical reasons, common practice [9], [32], [47]–[49] is to divide the total size of the dataset N on the amount of workers M and assign each worker a N/M partition with no overlap between the partitions. This assignment scheme is illustrated in Figure 2.10a. It is, however, to the best of our knowledge, not clear what the effect of this partitioning scheme has on the final accuracy of the model. An alternative assignment scheme could for instance be to give overlapping partitions of the dataset, as illustrated in Figure 2.10b. Different

[5] In the case of $N \mod B \neq 0$, the last iteration typically uses a smaller batch size to complete the epoch.

ways of assigning overlapping data will be presented in Section 4.1.



(a) Disjoint data shards, *i.e.,* $\mathcal{D}_i \cap \mathcal{D}_j = \emptyset$ for all $(i, j) \in \mathcal{D}$ where $i \neq j$

(b) Overlapping data shards with red sections indicating an overlap. Note that even though the figure illustrates overlap between two adjacent data shards, the overlap can be between multiple shards

Figure 2.10: Data assignment

2.2.7 Evaluation

There are mainly two measurements under consideration when evaluating a data parallel system. First of all, we want to achieve a good performance measure. By performance, we mean the accuracy on a test or valdiation set. The exact metric used (*e.g.,* top-1 validation/test accuracy, top-5 validation/test accuracy, etc.) will depend on the model-dataset combination, and what is most commonly used in the literature. For instance, as we will see in Section 3.1, for ResNet-50 [22] trained on ImageNet [23], the commonly used metric is top-1 accuracy. Second, we are concerned with the runtime of the system. This can for example be wall-clock time to finish a fixed amount of epochs or number of iterations to reach a target accuracy. It should be noted that wall-clock time is, however, highly dependent on the hardware, making it harder to compare results across different work. Another metric to evaluate the runtime is the amount of FLOPs. This is dependent on the model, dataset and number of iterations, but independent of the hardware.

Chapter 3

Related Work

In this chapter we will look at three ways of applying data parallelism to train DNNs. We will first look at previous work on *large scale data parallelism*, in which the general focus is on modifying parts of the model, its hyperparameters or the communication method when scaling up to a large number of workers. Next, we will look at some recent studies of *local SGD*. Local SGD is when workers do not synchronize for every iteration, but instead update their model locally and only periodically synchronize between each other. Finally, we will see how *distillation* [50] can be used to scale data parallelism by assigning the data to different groups with a method called *codistillation* [51].

3.1 Large Scale Data Parallelism

In this section we will look at work that has successfully scaled data parallel systems to a large number of workers. Asynchronous methods have historically been popular [9], but the trend in state-of-the-art systems is to use synchronous SGD due to its preferable time to convergence and validation accuracy when scaling to many workers [31]. All of the presented work is evaluated using ResNet-50 [22] on ImageNet [23], a dataset consisting of 1.28 million training images and 50,000 validation images split across 1,000 classes. Table 3.1 serves as an overview, and the rest of this section will introduce the main contributions of the different work. Note that wall-clock time in the table is the time to train 90 epochs unless specified otherwise in the respective paragraphs below.

Main contributions	**Workers**	**Batch size**	**Top-1 Validation accuracy**	**Time**
Linear scaling rule & learning rate warmup phase (Goyal *et al.*, 2017)	256 Tesla P100	8k	76.3%	60 min
Dynamically increase batch size (Smith *et al.*, 2017)	TPU v2 (128 chips)	8k→16k	76.1%	30 min
Final collapse & collapsed ensemble learning rate schedule (Codreanu *et al.*, 2017)	512	10k	76.4%	82 min
	512	16k	76.26%	74 min
	1024	32k	75.31%	42 min
	1536 Intel Knights Landing (KNL) nodes	48k	74.6%	28 min
RMSprop warmup, Slow-start learning rate schedule & Batch Normalization without moving averages (Akiba *et al.*, 2017)	1024 Tesla P100 GPUs	32k	74.9%	15 min
Layer-wise adaptive rate scaling (You *et al.*, 2017), (You *et al.*, 2018)	2048 KNL nodes	32k	75.4%	20 min
		32k	74.9%	14 min
Mixed-precision training with LARS & hybrid AllReduce, (Jia *et al.*, 2018)	2048 Tesla P40	64k	75.8%	6.6 min
Distributed Batch Normalization, 2D AllRedcue & input pipeline optimization (Ying *et al.*, 2018)	TPU v3 (1024 chips)	32k	76.3%	2.2 min
		64k	75.2%	1.8 min
Batch-size control, label smoothing & 2D-Torus AllReduce (Mikami *et al.*, 2018)	3456 Tesla V100	54k	75.29%	2 min

Table 3.1: Overview of large scale data parallel systems using ResNet-50 on ImageNet.

Linear Scaling Rule & Learning Rate Warmup Phase Goyal *et al.* [52] introduce a *linear scaling rule* in which the learning rate scales linearly with the batch size. The logic behind this rule is that if the batch size is increased by k while keeping the amount of epochs fixed, there are k fewer weight updates. Thus, it seems natural to take k bigger steps when updating the weights. Applying this technique without any other additional techniques, however, resulted in some instability at the initial phase of training. To overcome this, a *learning rate warmup phase* was proposed in which the learning rate is gradually increased until it reaches the target. Using these two techniques they are able to scale up to a batch size of 8k with ResNet-50 on ImageNet while maintaining 76.3% validation accuracy. With 256 Tesla P100 GPUs, this is accomplished in 60 minutes.

Increase Batch Size During Training Smith *et al.* [53] demonstrate that increasing the batch size can have same impacts on the validation accuracy as decaying the learning rate has.[1] They show that one can increase the batch size using constant learning rate until $B \sim N/10$, after which they start decaying the learning rate. Here, B is the global batch size and N is the size of the dataset. By increasing the batch size while keeping the number of epochs fixed, they also reduce the total amount of parameter updates. This is shown in an experiment conducted with Wide ResNet [61] on CIFAR-10 [62] where the decaying learning rate and increasing batch size reaches the same validation, but increasing batch size does so in fewer parameter updates. They also show that this applies for different optimizers such as plain SGD, SGD with momentum, Nesterov momentum and Adam. In an experiment where they train the first 30 epochs with batch size 8k and the last 60 epochs with 16k, they are able to train ResNet-50 on ImageNet with a 128-chip TPU v2 in 30 minutes without loosing accuracy. They compare this with another experiment in which they double the initial learning rate and use a constant batch size of 16k for the entire training. In this experiment, they loose 1.1 percentage points in validation accuracy.

Final Collapse & Collapsed Ensemble Learning Rate Schedule Based on experiments similar to those performed by Goyal *et al.* [52], Codreanu *et al.* [54] also noticed that the validation accuracy starts decreasing when the batch size exceeds 8k. They find that one of the reasons for this is a too large weight decay, particularly in the first phase of training when the learning rate is large. By using a smaller initial weight decay, as well as dynamically decreasing it until the last phase of training in which it is increased again, they are able to improve

[1] They do, however, note that this only apply when $B \ll N$.

the validation accuracy. They also implement a *final collapse* for the learning rate. The learning rate is decayed linearly until the last phase of training, in which it is decayed with a power of 2. This happens at the same time as they increase the weight decay. Using these two techniques, they are able to achieve a validation accuracy of 76.6% with batch size 8k. They also show that increasing the batch size to 16k only reduces the validation accuracy by 0.34 percentage points.

A more complex learning rate schedule, called *collapsed ensemble learning rate schedule*, is also introduced by Codreanu *et al.* With this schedule, the training is split into cycles. In their experiments, the first cycle starts after 45 epochs in which the learning rate goes from linearly decaying to a power-2 polynomial decay. After a few epochs, the learning rate is linearly increased by a factor of 3 for a couple of epochs. This cycle of polynomial decay followed by a linear increase is then repeated 4 more times. They also create snapshot ensembles called *collapsed ensembles* at the end of every polynomial decay in each of the cycles. Training for 120 epochs, they ensemble 5 models and achieve a validation accuracy of 77.5%. Stopping at 75 epochs, they achieved a single-model accuracy of 76.5%.

RMSprop Warmup, Slow-start Learning Rate Schedule & Batch Normalization without Moving Averages Akiba *et al.* [55] found that the early optimization difficulty when training could be addressed by starting the training using RMSprop, and then gradually transitioning to SGD with momentum. They do this by defining a custom update rule

$$w_t = w_{t-1} + \eta \triangle_t, \tag{3.1}$$

where

$$\begin{aligned} \triangle_t &= \mu_1 \triangle_{t-1} - \left(\alpha_{SGD} + \frac{\alpha_{RMSprop}}{\sqrt{m_t} + \epsilon} \right) \nabla L(w_t), \\ m_t &= \mu_2 m_{t-1} + (1 - \mu_2) \nabla L(w_t)^2. \end{aligned}$$

The momentum term $\triangle_t$ allows for adjusting the balance between SGD with momentum and RMSprop through α_{SGD} and $\alpha_{RMSprop}$. For instance, with $\alpha_{RMSprop} = 0$ and $\alpha_{SGD} = 1$, the update is only using SGD with momentum. The hyperparameters μ_1 and μ_2 determines the amount of momentum. In their experiments they use a function similar to the exponential linear unit (ELU) activation function to slowly transition from RMSprop to SGD with momentum.

By using this update rule, in combination with a slightly modified learning rate schedule of the one described by Goyal *et al.* [52] and performing BatchNorm without moving averages, they are able to train ResNet-50 on ImageNet in 15 minutes, achieving a top-1 validation accuracy of 74.9%.

Layer-wise Adaptive Rate Scaling You *et al.* [56] observed that if, for some layer, the learning rate is large compared to the ratio between the L2-norm of weights $||w||$ and update $||\nabla L(w_t)||$, the training can become unstable. Motivated by this, they introduce *Layer-wise Adaptive Rate Scaling* (LARS), a technique which uses separate local learning rates for every layer. The local learning rates for layer l is then defined as

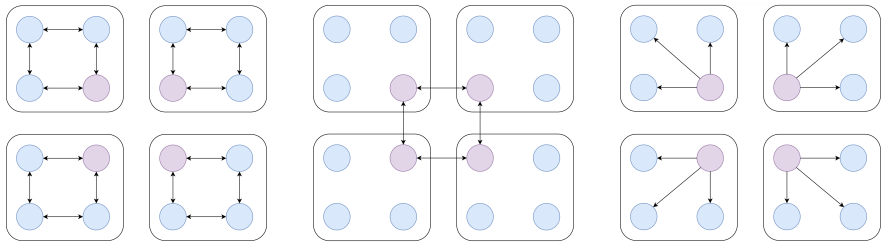
$$\eta^l = \lambda \times \frac{||w^l||}{||\nabla L(w^l)||}, \tag{3.2}$$

where $\lambda < 1$ is a "trust" coefficient to control the magnitude of the update.[2] Using LARS, they are able to scale ResNet-50 to a batch size of 32k with only a small loss in accuracy (-0.7%). To confirm that LARS can be used to scale to a large number of workers, You *et al.* [57] trained ResNet-50 on ImageNet with 2048 KNLs and finished 90 epochs in 20 minutes with a validation accuracy of 75.4%. When stopping after 64 epochs, they achieved 74.9% accuracy in 14 minutes. In these experiments they also adopted the learning rate warmup scheme [52].

Mixed-precision Training with LARS & Hybrid AllReduce Jia *et al.* [58] use a couple of techniques to scale up to 2048 workers. First of all, they use mixed-precision training with LARS. This is done as follows: (1) perform forward and backward pass using 16-bit floating points (16FP), (2) cast the weights and gradients to single-precision format (32FP), (3) apply LARS on 32FP, then (4) cast back to 16FP. They show that training ResNet-50 on ImageNet with a batch size of 64k when using LARS with mixed-precision achieves 76.2% accuracy, compared to 73.2% without LARS. Second, they optimize the communication method. They note that when scaling to a large amount of workers, Ring-AllReduce fails to utilize the full network bandwidth as the data is split into M chunks (see Section 2.2.2). They address this problem with two strategies. With *tensor fusion*, they pack multiple small tensors together before AllReduce. This ensures better bandwidth utilization, and thus also higher throughput. Since the higher throughput increases the latency, they also implement a *hiearchical AllReduce* where the workers are split into groups with one master each, as shown in

[2] You *et al.* uses λ to denote learning rate and η to denote the LARS coefficient, but we will use opposite notation for consistency with the rest of this report.

Figure 3.1. In the first phase (Figure 3.1a), each group performs a local reduce, and the master temporarily stores the result. Then, in phase 2 (Figure 3.1b), the masters from each group do a Ring-AllReduce to share their results. At last (Figure 3.1c), the masters share the final result to the workers of their respective groups. With k groups, this algorithm reduces the running steps from $2(M-1)$ to $4(k-1)+2(\frac{M}{k}-1)$. They note that this algorithm is best suited for small tensor sizes (*e.g.,* weights in a convolutional layer), and does not perform as well for large tensor sizes (*e.g.,* weights in a fully-connected layer). To have good performance in both cases, they design a *hybrid AllReduce* in which they can switch between Ring-AllReduce and Hierachical AllReduce based on the size of the tensor.



(a) Intra-group reduce (b) Inter-group AllReduce (c) Intra-group broadcast

Figure 3.1: Hierarchical AllReduce. A square represents a group. Purple workers are the assigned masters for their respective groups.

Distributed Batch Normalization, 2D AllReduce & Input Pipeline Optimization To overcome the issue with small local batch size when using Batch Norm (see Section 2.2.1), Ying *et al.* [59] design a *distributed batch normalization* algorithm in which the mean and variance of a batch are computed across a small subset of the workers. To further facilitate scaling, they optimize the communication method by extending the traditional 1D Ring-AllReduce to a 2D mesh algorithm in which the reductions is computed in two phases, one per dimension. They show that, on a 256 chip TPU v2, the 2D mesh algorithm is faster than 1D. They also note that enabling torus links provides even better performance by approximately halving the distributed sum time compared to the traditional 1D AllReduce when using 256 TPU v2 chips.[3] At last, they optimize the input pipeline. This includes enabling more efficient access patterns to the data,

[3]Naturally, this only works on hardware that has this ability. In their experiments in which the different communication methods were tested, they used a full TPU v2 Pod which has the ability to enable torus links.

prefetching the next batch during computing of current batch, more efficient decoding of datasets (*e.g.,* only decoding the part of an image that will be used after data augmentation methods such as cropping), and parallelizing the input pipeline across several CPU threads.[4]

Using all of the above described methods in addition to previously introduced methods such as LARS, mixed-precision, linear learning rate scaling, learning rate warm-up phase and learning rate decay scheduling, they are able to train ResNet-50 on ImageNet with batch size 32k and 64k in 2.2 and 1.8 min, respectively. The 32k run achieves a validation accuracy of 76.3%, while the 64k run achieves 75.2%.

Batch-size Control, Label smoothing & 2D-Torus AllReduce Mikami *et al.* [60] address the generalization gap when training with large mini-batches with two main techniques. First, they implement a batch size scheduling similar to the one proposed by Smith *et al.* [53], *i.e.,* they dynamically increase the batch size during training. Second, they regularize the model with a technique called *label smoothing* [63] where the probability value of a true label is decreased and the probability of a false label is increased. Finally, they address the communication overhead by using a 2D-Torus topology where the $X \cdot Y = M$ workers are arranged in a $X \times Y$ grid as shown in Figure 3.2. Compared to standard Ring-AllReduce which uses $2(M-1)$ steps, 2D-Torus AllReduce only use $2(X-1)$ steps. The complete AllReduce algorithm is then implemented as follows:

1. Reduce-scatter in horizontal direction
2. AllReduce in vertical direction
3. AllGather in horizontal direction

They show that in an experiment with 4096 Tesla V100 GPUs, using a dynamic batch-size 34k $\rightarrow$ 119k, they achieve a 75.23% validation accuracy with ResNet-50 on ImageNet in 129 seconds. Their best achievement, however, was achieved without batch size control. With a constant batch size of 54k, they achieve a 75.29% validation accuracy in 122 seconds with 3456 Tesla V100 GPUs.

[4]When training on a single worker, the entire ImageNet dataset of 1.28 million images does not fit into memory, and must be read from disk during training. With a large amount of workers, however, the data partitions get smaller and are more likely to fit into local memory.

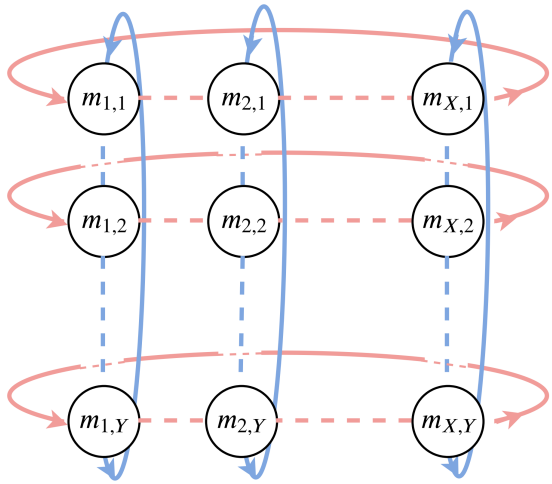


Figure 3.2: 2D-Torus AllReduce. M workers are arranged in a $X \times Y$ grid. The workers reduce-scatter in horizontal direction (red lines), then AllReduce in vertical direction (blue lines) and lastly AllGather in horizontal direction.

3.2 Local SGD

In this section we look at some of the recent progress regarding local SGD. One common reason to use local SGD is to address the communication overhead that synchronous data parallel training incurs. There is a trade-off however, as naively doing so will negatively impact model accuracy. With local SGD, each worker runs independently and updates its local model parameters for a certain amount of steps before the parameters are synchronized between the workers at a world synchronization period L_W. This concept is also described in the perspective of different synchronization strategies in Section 2.2.3.

Post-local SGD & Hierarchical Local SGD Lin *et al.* [38] introduce Post-local SGD, a two-phased training setup, where the training is synchronous in the first phase, and local SGD is used in the second phase, with 8 local steps between synchronizations. With this setup they were able to retain the baseline accuracy while significantly decreasing the communication overhead in the second phase. They also show that increasing the batch size decreases accuracy, despite linearly scaling the learning rate and employing a learning rate gradual warm-up scheme. Their baseline achieved an accuracy of 93%, and using a 4 times larger batch size

achieved 89%, while using local SGD instead with 4 local steps resulted in 92%. As such they propose to use local SGD instead of increasing batch size.

Lin *et al.* [38] also describe Hierarchical local SGD, which is a way local SGD can be used in each level in a hierarchy of devices. Typically hardware is organized as layers in a hierarchy, like accelerators in a node, nodes in a rack and rack in a data-center. Each of these layers has different communication bandwidth. For example the communication between accelerators and a CPU is faster than the network between nodes. Hierarchical local SGD takes advantage of this hierachy by synchronizing the workers within a node at one period, and synchronizing between all nodes at a lower period. In one of their experiments they simulated using many devices by artificially delaying the global synchronization, showing a drastic runtime improvement when employing this setup.

Adaptive synchronization period Wang *et al.* [64] explores the error-runtime trade-off related to local SGD, and also introduces AdaComm, an adaptive communication strategy. Based on their analysis of the error-runtime trade-off, they propose an expression of the optimal communication period. AdaComm, the scheme they developed, starts the training with local model updates for a long period between each synchronization, and over the course of the training linearly decreases the period between synchronizations. Their experiments with this scheme achieved up to a 3x improvement in runtime without degrading the accuracy compared to a synchronous baseline.

3.3 Codistillation

Distillation [50] is the process of transferring the knowledge of either an ensemble [65] or a large regularized model (*teacher model*), to a smaller, distilled model (*student model*). This is typically done in two phases: first training the teacher, followed by training the student model using the *soft targets* generated by the teacher.[5] The soft targets are a smoothed probability distribution obtained by adding a temperature T to the softmax function in the last layer of a DNN, in which a higher temperature leads to a smoother distribution. The softmax function with temperature is given by

$$q_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}, \tag{3.3}$$

[5] One could also incorporate the hard target to the training of the student model. In this case, the objective function containing the hard labels would typically get a lower weight.

where T is the temperature, q_i is the probability for class i and z_i is the i^{th} element of the input vector to the softmax function. With the use of soft targets instead of hard targets (one-hot targets from the dataset), the student model tries to learn the generalization from the teacher.

Using distillation as a backbone, Anil *et al.* propose *Codistillation* [51], an online distillation method that trains n copies of a model in parallel. Codistillation differs from standard distillation in three primary ways: (1) every trained model uses the same architecture, (2) the distillation is performed during training before any of the models has converged, and (3) every model acts as both teacher and student. Since the distillation happens during training, the two phases in standard distillation are merged together into a single phase. Each worker is given a disjoint partition of the dataset which it uses to train a local model for a set amount of iterations before the workers start occasionally sharing their model parameters with their neighbours. From here on out, every worker use the latest received model parameters w_j as teacher models when training. This is done by adding a distillation loss ψ to the update of student i's parameters w_t^i at iteration t. Using the same notation as in (2.2), the expanded update rule is given by

$$w_{t+1}^i = w_t^i - \eta \nabla \left(L(w_t^i, \mathbf{x}) + \psi \left(\frac{1}{|NB|} \sum_{j \in NB} L(w^j), \mathbf{x}), pred(w_t^i, \mathbf{x}) \right) \right), \quad (3.4)$$

where $\mathbf{x}$ is the input sample(s), NB is the set of neighbours for the student, w^j is the parameters of teacher model j and $pred(w_t^i, \mathbf{x})$ is the prediction made on $\mathbf{x}$ by the student model. The distillation loss ψ can for instance be the cross entropy error where the inputs are the student prediction and the soft targets generated by the teacher models. Note that the teacher predictions will be stale if the parameters is not exchanged after every update. Anil *et al.*, however, contemplate that stale predictions has less impact on model quality than stale gradients.

Anil *et al.* further argue that codistillation can be used to speed up the training by dividing M workers into n groups of K workers where each group is given a disjoint partition of the dataset, as illustrated by Figure 3.3. This means that the groups will train independent models, where the gradients for a model are communicated within a group (synchronous data parallelism with mini-batch averaging, see Section 2.2.3) and the model parameters are exchanged occasionally between the groups. In their experiments, they use two groups of 128 workers ($n = 2$ and $K = 128$) which they base on previous experiments showing that synchronous data parallelism with mini-batch averaging has diminishing returns in terms of runtime when exceeding 128 workers. Thus, instead of using the extra

workers to train the same model, they decrease training time by assigning disjoint parts of the dataset to groups which train different models. For the groups to learn from each others data, they occasionally exchange model parameters to update the teacher models.[6] They further show that by assigning the same data to both groups (*i.e.,* full overlap), the accuracy is degraded compared to when the data is assigned with no overlap.

Even though codistillation requires a forward pass on the teacher models, they note that this computation can be overlapped with the forward pass of the student model.[7] With this setup, they are able to achieve the baseline (128 worker synchronous data parallelism with mini-batch averaging) validation error in 2x fewer steps. They also show that by training longer, codistillation with two groups is able to reach a better final validation error.

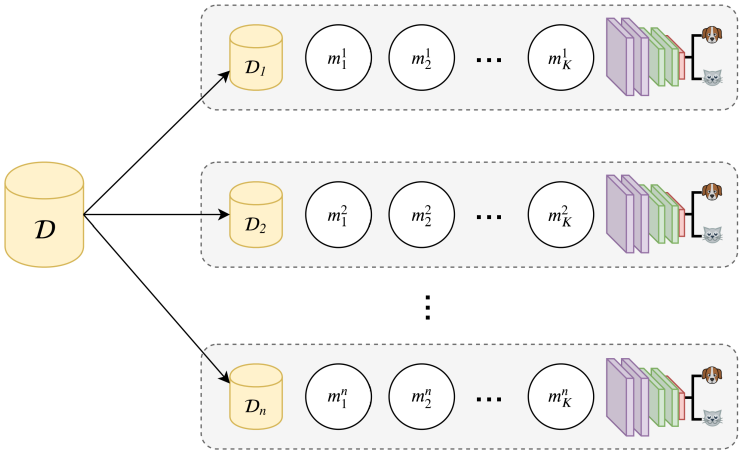


Figure 3.3: Codistillation with n groups of K workers. Different groups are given disjoint data partitions, *i.e.,* $\mathcal{D}_1 \cap \mathcal{D}_2 \cap \cdots \cap \mathcal{D}_n = \emptyset$

[6] Anil *et al.* only experimented with two groups, but they suggest that if scaled to more than two groups, the groups might benefit from being arranged in a sparse topology (*e.g.,* a ring) instead of a fully connected graph, as very dense topologies might lead to the models becoming too similar.

[7] Note that this can be another motivating factor for using sparse topologies to connect the groups when using several groups.

3.4 Summary & Motivation

In this chapter we have seen three ways of training DNNs using data parallelism. In Section 3.1, we see that the focus is on modifying the model, its hyperparameters or optimizing the communication method When scaling to a large number of workers. We have also seen some techniques that allows for increasing the batch size without loosing significant performance (*e.g.,* the linear scaling rule, learning rate warmup phase and LARS). Further, in Section 3.2 we have seen that local SGD can be used to reduce the number of communication rounds. Techniques for handling the loss in performance when reducing the communication rounds have also been presented. At last, we have seen in Section 3.3 how different groups of workers train independent models on their local data, and through exchanging teacher models, the information from separate parts of the dataset are shared between the groups. Here, we see an explicit statement of how assigning the data in different ways between two groups impact the performance. However, this only applies when training using codistillation, and not data parallelism with either gradient or parameter averaging (see Section 2.2.3). This lack of any study on how assigning the data in different ways impact the performance will be the motivating factor for the work in this project.

It should also be noted that we will use some of the presented techniques in this chapter to further expand upon initial experiments, enabling deeper analyses for some of the experiments. More specifically, we will use the linear scaling rule and learning rate warmup scheme to alter the learning rate. We will also experiment with local SGD in which we study how assigning the workers different amount of data impacts the performance. This study will also be expanded upon in which we will use the concept of hierarchical local SGD to further apply different data assignment schemes when reducing the number of communication rounds (as we will see in Chapter 4, and more specifically Section 4.2, we will refer to this concept as households and neighbourhoods).

Chapter 4

Methodology

In this chapter we will describe the main concepts that will be experimented with in Chapter 5. This includes a look at different ways of assigning data between multiple workers with varying degrees of overlap. We will also use the concept of hierarchical local SGD [38] to describe a communication reduction method in which the workers are grouped into what we refer to as *households* and *neighbourhoods*. At last, we will conclude this chapter with a description of how we conduct experiments with focus on the experimental process and scope of the project.

4.1 Data assignment

When utilizing data parallel training (see Section 2.2), a dataset is typically distributed evenly between a set of workers with no overlap of data between them. Here we propose an approach where some workers may or may not share a portion of the dataset. In other words it causes a non-zero intersection of data samples between certain workers. We refer to this concept as overlap of data between workers.

Before we describe different ways of assigning the data to the workers, we will look at how the dataset is split into *shards* and *samples*: after shuffling the dataset, we divide the dataset $\mathcal{D}$ into S shards, where one shard consists of j data samples. Each sample s is associated with input data x and target label y,

and is referred to as $s = (x, y)$.[1] Formally, we have $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_S\}$ where $\mathcal{D}_i = \{s_{i,1}, s_{i,2}, \ldots, s_{i,j}\}$ for all i. An example of data assignment is shown in Figure 4.1. In standard distributed data parallel training using M workers, we will create $S = M$ shards, and assign each worker a unique shard. This situation is illustrated in Figure 4.2a. At the other end of the spectrum, one could assign all shards to all workers, which would result in full overlap between the workers, as illustrated in Figure 4.2b.

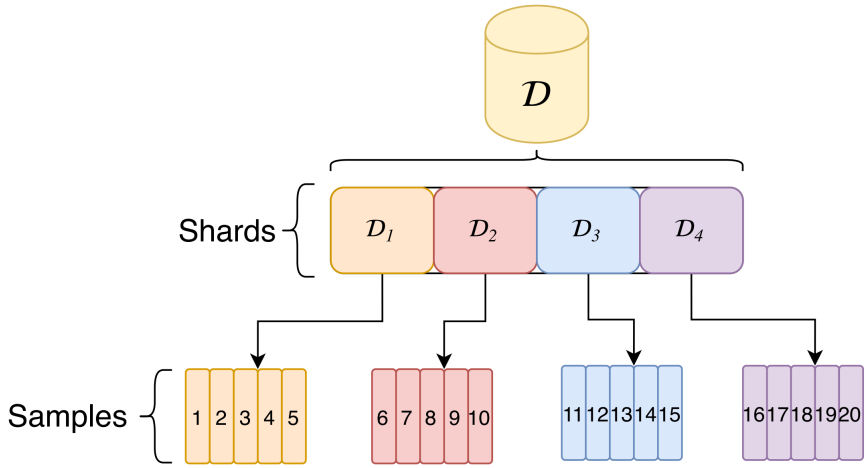
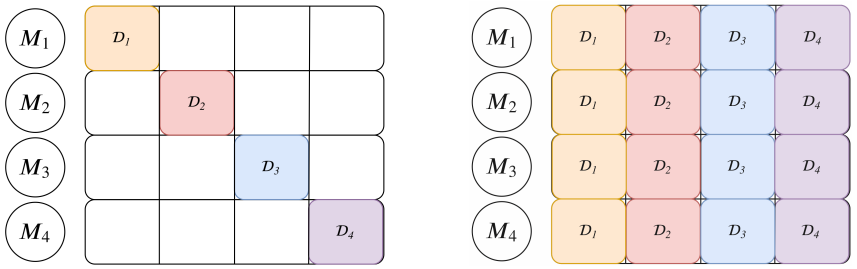


Figure 4.1: A dataset $\mathcal{D}$ divided into shards $\mathcal{D}_i$, where a shard consists of multiple samples. In this figure there are 20 data samples in the dataset divided into 4 shards and each shard contains 5 unique samples.

One way to quantify the amount of overlap is to look at how many times each data sample is assigned. We refer to this quantity as number of copies C of every data sample, where $1 \leq C \leq M$. We see that for $C = 1$, we have disjoint data assignment (Figure 4.2a), while for $C = M$, we have full overlap data assignment (Figure 4.2b). When $1 < C < M$, there are several ways to assign the data. For our experiments we have used the following:

- **Coarse sharding**: With this strategy, we create $1 < S < M$ overlapping shards, where worker M_i gets shard $S_j = i \mod S$. This sharding scheme results in groups of $\frac{M}{S}$ workers having equal data (which, naturally, is also the number of copies C). Figure 4.3a illustrates an example of coarse sharding where $S = 2$ and $M = 4$.

[1] Here we assume supervised learning.

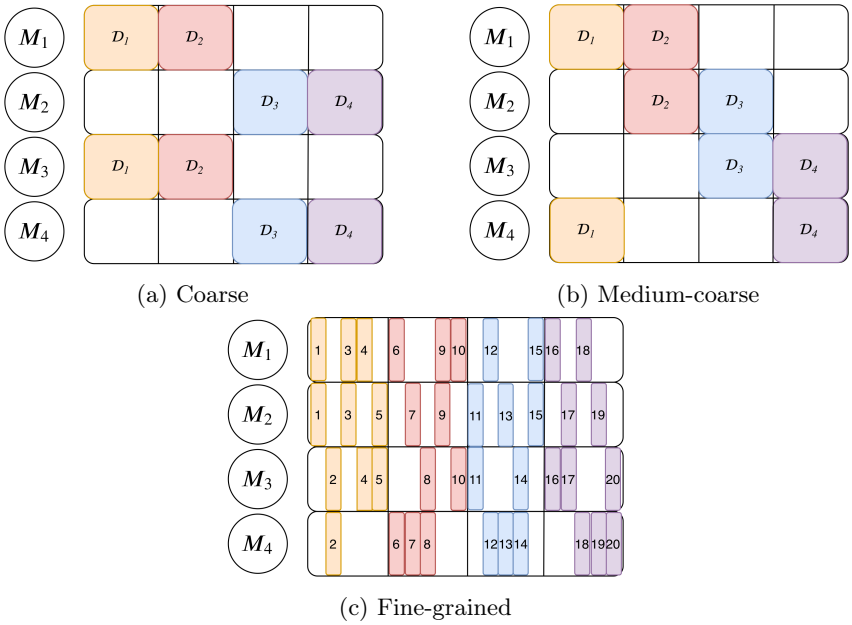


(a) No overlap (b) Full overlap

Figure 4.2: Assignment of shards to four workers. Circles represent the workers, and each row represent the dataset. The presence of a shard at a row represent the assignment to the respective worker. Figure 4.2a shows the assignment of disjoint shards, Figure 4.2b shows the assignment of full overlapping shards.

- **Medium-coarse sharding**: With this strategy we first define a number of copies C. Next, the dataset is split into M shards. Worker M_i then gets C shards; $S_i, S_{(i+1) \mod M}, \ldots, S_{(i+C-1) \mod M}$. With this sharding scheme, $2C - 1$ workers will have intersecting data, but no workers will have equal data. Figure 4.3b illustrates an example of medium-coarse sharding where $C = 2$ and $M = 4$.
- **Fine-grained sample assignment**: With this strategy we assign samples rather than shards. This is accomplished by defining a number of copies $1 < C < M$, where every sample in the dataset is assigned to C workers with the aim of having intersection between all M workers when all samples are assigned. With N total samples in the dataset, each worker gets a total of $\frac{C \cdot N}{M}$ samples. Figure 4.3c illustrates an example of fine-grained sharding where $C = 2$, $N = 20$ and $M = 4$.

The main difference between these three data assignment schemes is the resulting intersection of data between workers. We can see that with coarse and medium-coarse sharding, there will be zero intersection between some of the workers. With fine-grained sample assignment, on the other hand, we can assign samples in such a way that makes sure that all workers have intersecting data.



(a) Coarse

(b) Medium-coarse

(c) Fine-grained

Figure 4.3: Different sharding strategies

4.2 Households & Neighbourhoods

Instead of synchronizing between all workers at every iteration (or local SGD with $L_W > 1$, see Section 3.2), we can divide the workers into $H < M$ *households*, and alternate between synchronizing within households (*household synchronization)* and between households (*world synchronization*). This alternation of synchronization is accomplished by setting a household synchronization period L_H and a world synchronization period L_W, where $L_H \neq L_W$ (typically with $L_H < L_W$). Figure 4.4 illustrates an example of household arrangement.

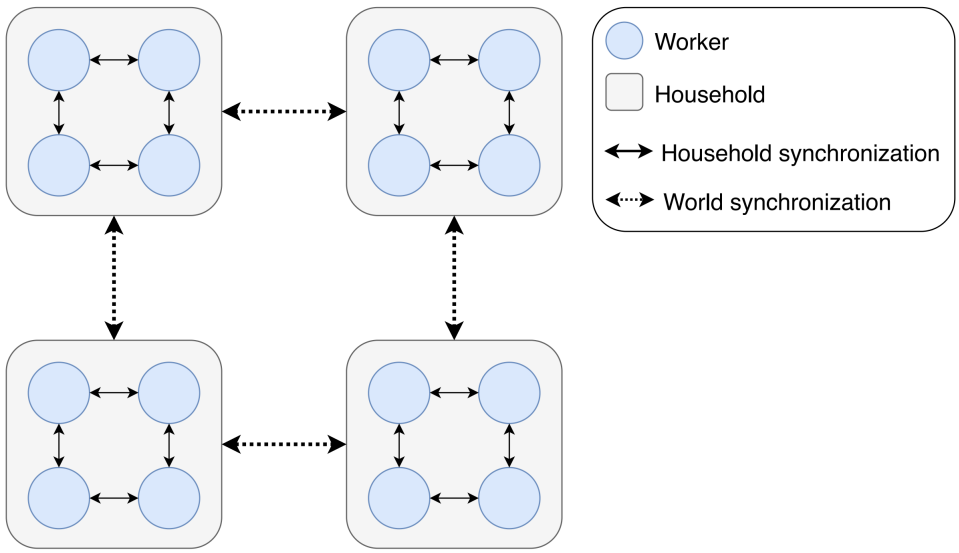


Figure 4.4: Arrangement of workers into households. In this figure there are a total of 16 workers arranged in 4 households.

We can also add another level of grouping by dividing households into $NB < H$ *neighbourhoods* and setting a neighbourhood synchronization period L_N, with $L_N \neq L_H \neq L_W$ (typically also with $L_H < L_N < L_W$). Figure 4.5 illustrates an example of neighbourhood arrangement.

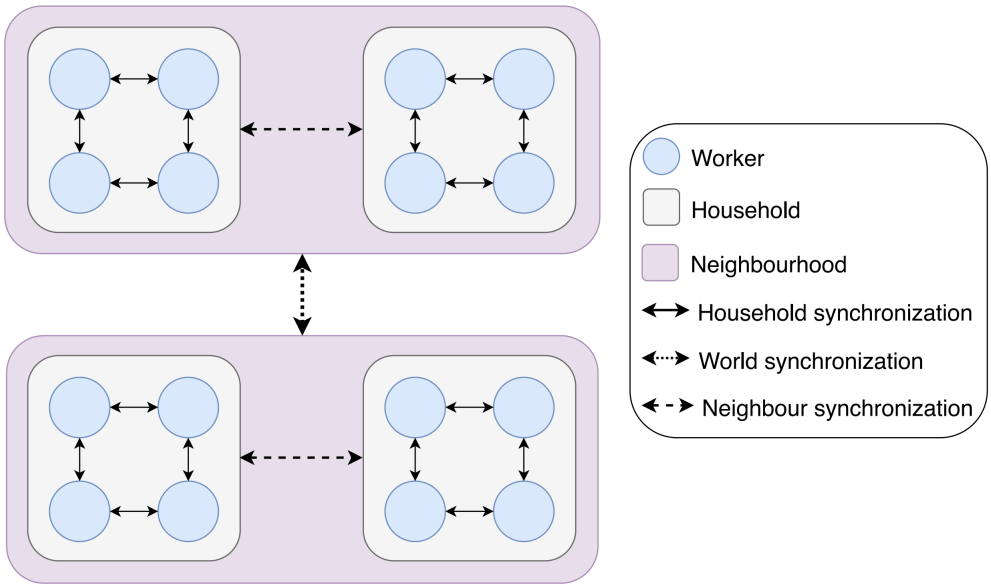


Figure 4.5: Arrangement of households into neighbourhoods. Here there are 16 workers, 4 households and 2 neighbourhoods.

4.2.1 Household shards

The straightforward way to assign data when arranging the workers into households and neighbourhoods is to give each worker a unique shard of the dataset (*i.e.,* no overlap, see Figure 4.2a). This results in each household having $\frac{1}{H}$ of the dataset. We refer to the data assigned to a household as a *household shard*, and to remove ambiguity we refer to shards at a worker level for *worker shards*, as illustrated in Figure 4.6. We see that if we set the number of household shards S_H equal to the number of households H, we have disjoint household shards. If $S_H < H$, on the other hand, we have overlapping household shards.

An example of overlapping household shards is seen in Figure 4.7, where we have M=16, H=4 and S_H=2. We see that $\frac{H}{S_H} = \frac{4}{2} = 2$ households have equal data.[2] Another example of assigning overlapping household shards is to create one neighbourhood for each copy of the dataset. Figure 4.8 shows an example with the same data assignment as in Figure 4.7a, but with one neighbourhood

[2]When we have equal data at different households we give unique initialization seeds to the households to ensure that they do not process the same batches.

per copy of the dataset.

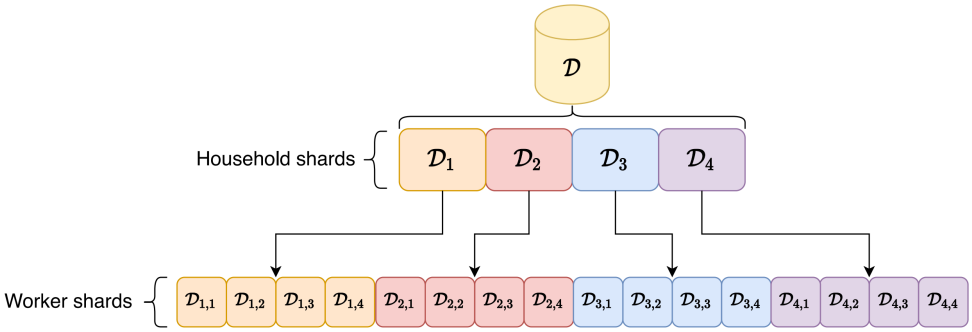
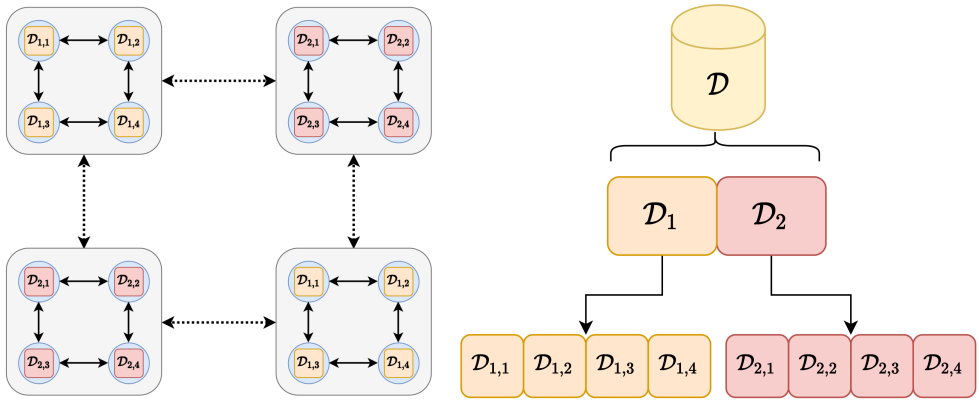


Figure 4.6: Household shards consisting of worker shards



(a) Assignment of household shards

(b) Two household shards

Figure 4.7: Example of assignment of household shards when $S_H < H$. Here we have $M = 16$, $H = 4$ and $S_H = 2$

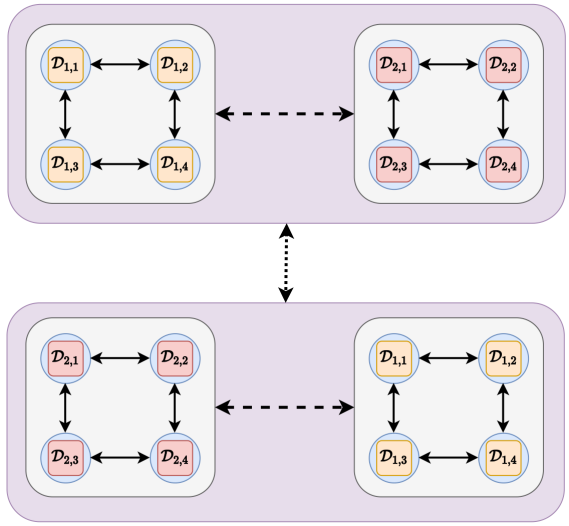


Figure 4.8: Neighbourhoods with $S_H = \frac{H}{NB}$. Each neighbourhood has access to the entire dataset

4.3 Plan of the Experiments

This section will introduce necessary information about how the experiments in Chapter 5 are conducted. This includes a look at the technology used, both software frameworks and the utilized hardware, as well as how we design and run our experiments.

4.3.1 Technology

When selecting a distributed deep learning framework, we value flexibility over speed and scalability for three main reasons:

1. We are not directly concerned with the runtime in terms of seconds, but rather the total number of epochs (that is, the total number of samples processed)

2. The scope of this project will limited to 16 workers, thus limiting the need for high scalability

3. The observation that frameworks that prioritize speed and scalability often comes with a limitation on flexibility, leaving out the possibility to control how the data is assigned

Thus, we will use Ray [66], a flexible framework for designing and running parallel and distributed applications in Python. Ray can express both task-parallel and actor-based computations. Tasks are used to dynamically load balance simulations, process large inputs and state spaces, and recover from failures. Actors enable stateful computations, such as model training. For our experiments, we will utilize the actor abstraction in combination with Tensorflow [67] to represent model replicas.

All experiments conducted in this project are run on the NTNU IDUN computing cluster [68]. The cluster has more than 70 nodes and 90 GPGPUs. Each node contains two Intel Xeon cores, at least 128 GB of main memory, and is connected to an Infiniband network. Half of the nodes are equipped with two or more Nvidia Tesla P100 or V100 GPGPUs. Idun's storage is provided by two storage arrays and a Lustre parallel distributed file system.

4.3.2 Experimental process

As stated in Section 1.3, this project will use an experimental and analytical research method. The process will be a standard scientific experimental process. More specifically, the process will consist of first stating a couple of hypotheses, followed by implementing the required concepts to test these hypotheses, running experiments, and at last analysing the results. The analysis will possibly lead to new hypothesis, and thus, the process is then repeated to address the research goal and research questions of this thesis.[3] It should also be noted that we split dataset into a train/validation split, and run each experiment with 5 different seeds in which we will use the mean top-1 validation accuracy for analysis, in addition to the standard deviation when conducting statistical hypothesis tests.[4] For the results we find most interesting and relevant for the key message of this thesis, we will run the model on the test set to ensure that the analysis performed on the validation set also accounts for the test set.

4.3.3 Project scope

We limit the scope of this project to the use of one DNN architecture and dataset. More specifically, for all experiments, we train ResNet-20 [22] on CIFAR-10 [62]. We use the workload described by He *et al.* [22] (specific details are elaborated on in Experiment E1), and turn the focus to different data assignment schemes in combination with different degrees of synchronization. Thus, most exploration of hyperparameters will be with regards to these two concepts (*i.e.,* data assignment and degrees of synchronization). However, for some experiments, we alter a couple of model hyperparameters to get a broader view and multiple comparisons to enable deeper analyses. More specifically, we will make some alterations to the batch size and the learning rate, while keeping all other model hyperparameters constant. The scope and focus of this project are summarized in Figure 4.9 where red boxes represent what we keep constant throughout all experiments, green boxes represent the main exploration, and blue boxes represent what we explore to a certain degree (mainly in combination with either data assignment or communication reduction).

Even though one of the key motivating factors of parallelizing the training of DNNs is to improve the runtime, we are mainly concerned with the performance

[3] Note that not all steps are necessarily repeated. For instance, a concept is only implemented once, and then experimented in combination with other concepts.

[4] The seed impact the initialization of the neural network weights as well as the shuffling of the dataset.

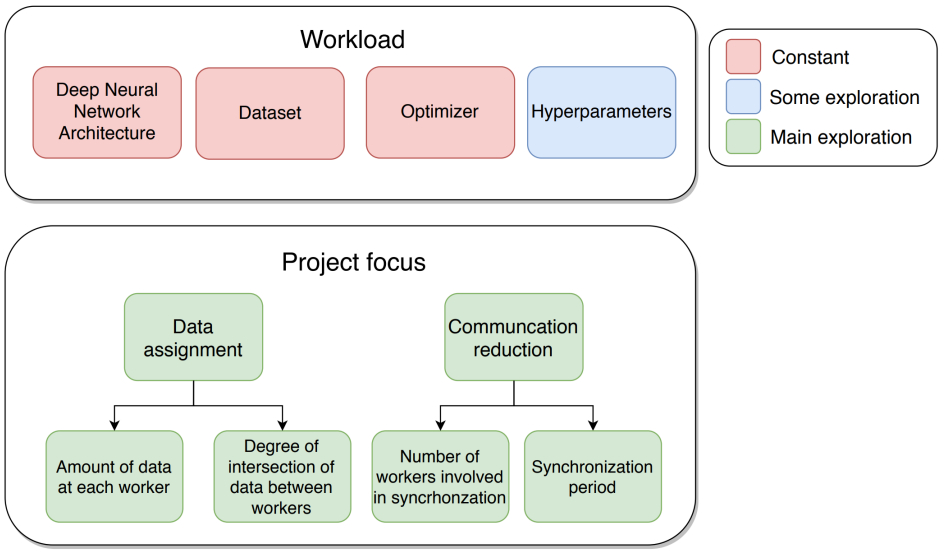


Figure 4.9: Experimental scope

in this project. As discussed in Section 2.2.7, with performance we refer to the final accuracy of the model. More specifically, we are concerned with the mean final top-1 validation accuracy over 5 runs given a constant number of processed data samples. We will, however, reduce the number of communication rounds, and thus also improve the runtime, but are mainly concerned with how different data assignment schemes impact the performance in combination with the reduction of communication rounds.

Chapter 5

Results & Analysis

In this chapter, we will analyse and discuss the experiments that has been conducted to address the research goal of this thesis (see Section 1.2). We will first present the baseline experiment in E1. Here, the training is fully synchronous, and the workers are assigned disjoint data shards. We keep experimenting with fully synchronous data parallelism in E2 where we explore different degrees of overlap between the workers. This experimental section presents three experiments:

- **E2.1**: Compares the difference between assigning full overlap and disjoint data.
- **E2.2**: Compares three data assignment schemes where the amount of data and intersection of data between workers differ.
- **E2.3**: Explores how increasing the batch size with varying degrees of overlap impact the accuracy.

At last, in E3, we will look at different ways of reducing the number of communication rounds with the focus on how the assignment of different amount of data impact the accuracy. The experiments in this section are structured as follows:

- **E3.1**: Compares disjoint and full overlap data assignment when reducing then number of communication rounds by using local SGD.

- **E3.2.1**: Divides the workers into households where each household is given a disjoint household shard. All model hyperparameters are the same as in the baseline experiment.
- **E3.2.2**: Divides the workers into households where each household is given a disjoint household shard. The effective batch size is kept constant.
- **E3.2.3**: Divides the workers into households where each household is given a disjoint household shard. Both the effective batch size and batch size/learning rate relationship are kept constant.
- **E3.2.4**: Compares disjoint and overlapping data between households in combination with three different batch size and learning rate combinations.
- **E3.3**: Compares the assignment of disjoint and full overlapping data between two neighbourhoods.

To conclude this chapter, we will find the test accuracy for the experiments we consider most significant for the research goal of this thesis. The purpose of the test accuracy is to ensure that the analyses performed with respect to the validation accuracy throughout the experiments are also valid for the test set. Also note that for the majority of the experiments in this chapter we show summary plots that contains the mean final validation accuracy. We refer the reader to Appendix B for plots with mean validation accuracy plotted throughout training.

5.1 E1 - Baseline

Goal The goal of this experiment is to establish a baseline that can be used for comparison for all other experimental results. As stated in Section 2.2.6, the common way to assign data is with no overlap. Thus, the baseline experiment will use this data assignment scheme. The DNN architecture and hyperparameters are based on the setup described by He *et al.* [22].

Method & Data We train ResNet-20 with BatchNorm for 64k iterations (182 epochs) on CIFAR-10 with varying number of workers M. The data is split into a 45k/5k train/validation split, and assigned with no overlap between workers (see Figure 4.2a). We use global batch size $B_{global} = 128$. For the runs with $M > 1$, we keep the global batch size constant, and set $B_{local} = \frac{B_{global}}{M}$. We use SGD with momentum [69] as optimizer. The momentum coefficient is set to 0.9. The weights are initialized as described by He *et al.* [70] with a weight decay of 0.0001. The learning rate is set to 0.1 at start of training, and is divided by 10 at 32k and 48k iterations (50% and 75% of the training process, respectively).

In order to reduce overfitting, we use data augmentation; an image is padded with 4 pixels on each side, a 32x32 random crop is sampled from this, and the crop may be horizontally flipped with a probability of 50%. In addition, the pixel values are normalized and the mean is subtracted.

Results & Discussion The results from this experiment are plotted in Figure 5.1. As described in Section 2.2.1, training DNNs using data parallelism is vulnerable to large global batch sizes and, when using BatchNorm, small local batch sizes. We have kept the global batch size constant, and decreased the local batch size when increasing the number of workers. With M=16, we have $B_{local} = \frac{128}{16} = 8$, but as we can see from the results in Figure 5.1 the effects in terms of performance when scaling from 1 to 16 workers is negligible.

Experiments [21] conducted by Wu *et al.* shows that when training ResNet-50 on ImageNet with local batch sizes smaller than 16, the accuracy starts to drop. Even though we have a smaller local batch size than 16 in this experiment, we hypothesize that the simplicity of the CIFAR10 dataset compared to ImageNet enables the use of smaller local batch sizes. We further hypothesize that the accuracy could start dropping if the number of workers are increased beyond 16. However, since exploring the effects of small local batch sizes is not the focus of our work, in combination with budgeting for other experiments, we leave this for

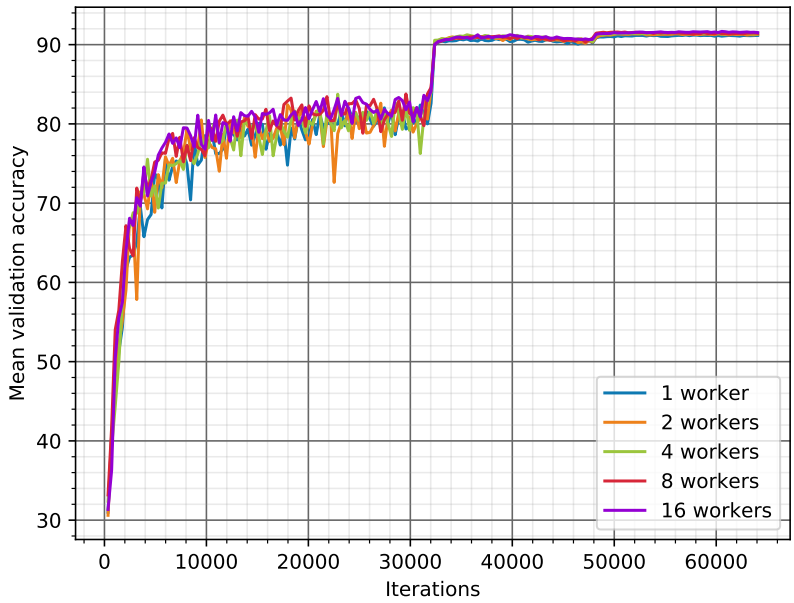


Figure 5.1: Baseline experiment with no overlap in data between workers

future work.

5.2 E2 - Fully synchronous training

In this experimental section we will explore the effects of assigning different degrees of overlapping data when training fully synchronous (see Section 2.2.3). We will also explore the effects of increasing the global batch size when varying the amount of overlap between the workers. The overall goal of these experiments is to investigate **RQ1**. That is, we want to find out the effects in terms of accuracy when assigning the data in different ways between the workers.

5.2.1 E2.1 - Full overlap

Goal The goal of this experiment is to explore whether there is any difference in final accuracy between the two extremes of assigning data, namely disjoint and full overlap (see Figure 4.2 in Section 4.1).

Method & Data We assign all data samples to all workers. If we used same number of epochs as when assigning disjoint shards, it would result in processing C times as many data samples (see Section 4.1 for an explanation of how C quantifies the amount of overlap). Also note that the term epoch is hard to interpret when assigning overlapping data. Thus, we use the term *overlap epoch*. An overlap epoch is completed when a worker has processed all its $\frac{N \cdot C}{M}$ local data samples. At that point, the entire system has processed a total of $N \cdot C$ samples. To process approximately the same total number of samples as with disjoint data assignment, we set a target epoch count E_{target} and divide by the number of copies C to get the total number of overlap epochs $E_{overlap} = \frac{E_{target}}{C}$. Note that the number of target epochs will not always be divisible by the number of copies, that is, $E_{target} \mod C \neq 0$. We could run the last overlap epoch until we have processed a total of $N \cdot E_{target}$ samples, but this can result in some samples being processed more than others. Therefore we ceil the number of overlap epochs, as it ensures that every sample is processed equally many times. When we do this we get $\left\lceil \frac{E_{target}}{C} \right\rceil \cdot C$ *effective epochs*. The number of effective epochs represents how many times each data sample is processed.

When we train using disjoint data assignment we validate the model once after every epoch. However overlapping data causes a higher number of iterations per epoch, which would result in validating less frequently. So instead we validate every $\lceil \frac{N}{B_{global}} \rceil$ iteration regardless of the number of iterations per epoch.

Results & Discussion The mean validation accuracy throughout training when assigning full overlap data assignment is shown in Figure 5.2. Figure 5.3 shows the difference between disjoint data assignment and full overlap, where above zero means that the disjoint is better, and below zero means that full overlap is better. Also note that this comparison only applies up until the last iteration of the baseline experiment (*i.e.,* at iteration $\lceil \frac{N}{B_{global}} \rceil \cdot E_{target}$). We are more concerned with the final accuracy, and as the plot shows, there is no significant difference between full overlap and disjoint data assignment at the end of training.

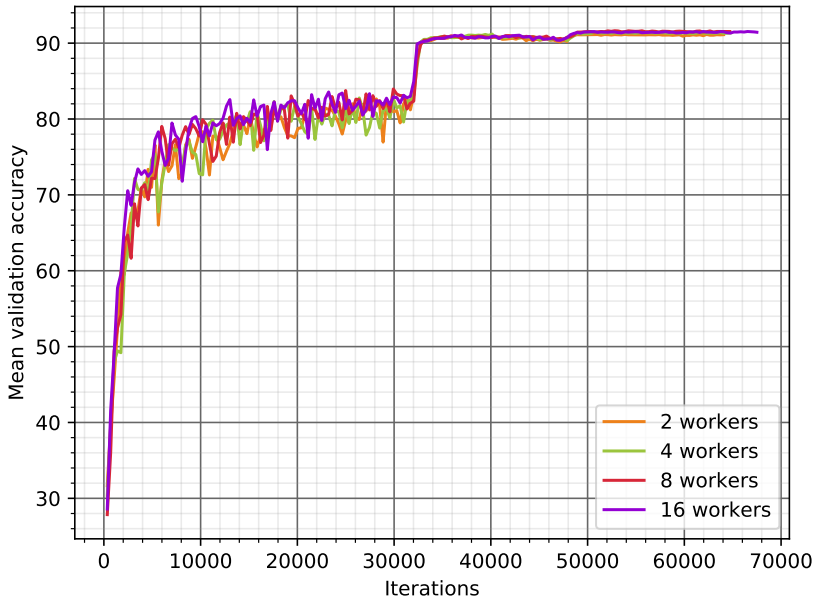


Figure 5.2: Mean validation accuracy throughout training for varying number of workers when assigning full overlapping data

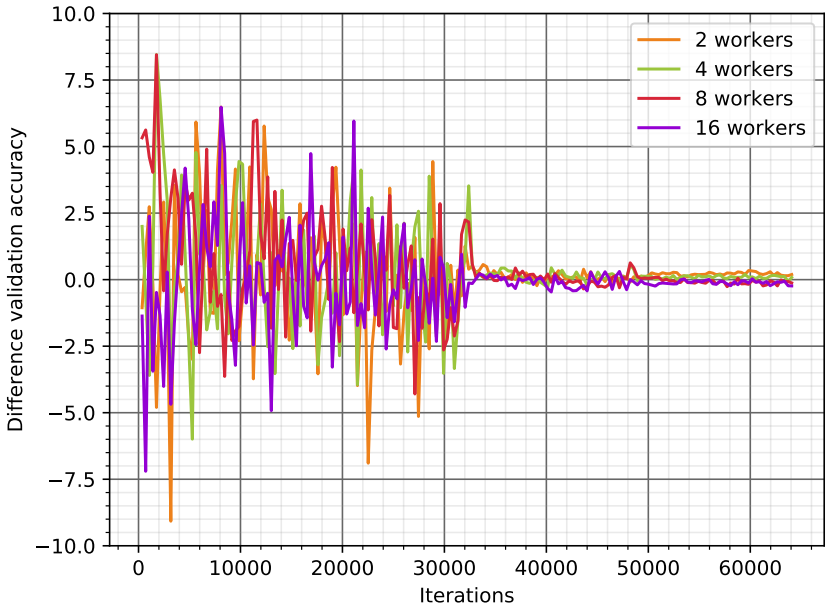


Figure 5.3: Difference in mean validation accuracy between disjoint and full overlap data assignment. Above zero means disjoint is better, below zero means full overlap is better.

5.2.2 E2.2 - Varying degrees of overlap

Goal The goal of this experiment is to explore how varying degrees of overlap between multiple workers affect the accuracy when training fully synchronous. We want to find out whether there is a trade-off between the degree of overlap and the accuracy in which there are two main points of interest (see Section 4.1): (1) the amount of data available to each worker and (2) the degree of intersecting data between the workers.

Method & Data We experiment with three different data assignment schemes (see Section 4.1 for a detailed explanation of the schemes):

1. *Coarse sharding*: We divide the dataset into $S < M$ shards and assign overlapping shards (see Figure 4.3a for a specific example with $M = 4$ and $S = 2$). We run with $M = 16$, and experiment with $S \in \{2, 4, 8\}$. The number of copies C are given by $C = \frac{M}{S}$
2. *Medium-coarse sharding*: We give each worker C shards (see Figure 4.3b for a specific example with $M = 4$ and $C = 2$). We run with $M = 16$, and experiment with $C \in \{2, 4, 8\}$.
3. *Fine-grained sample assignment*: We define a number of copies C and assign each worker $\frac{N \cdot C}{M}$ samples in such a way that all workers have intersecting data (see Figure 4.3c for a specific example with $M = 4$ and $C = 2$). For this experiment, we have assigned the samples to get approximately equal intersection between all pairs of workers (*i.e.,* variance of intersection approximately equal to 0). We run with $M = 16$, and experiment with $C \in \{2, 4, 6, 8, 10, 12, 14\}$.

When assigning overlapping data, we run $E_{overlap} = \left\lceil \frac{E_{target}}{C} \right\rceil$ overlap epochs, as described in Experiment E2.1.

Results & Discussion The plot in Figure 5.4 shows a summary of the three different data assignment strategies where the x-axis quantifies the amount of overlap with $\frac{C}{M}$. This quantification means that a value of 1 means full overlap, and a value of $\frac{1}{M}$ means disjoint data assignment. Instead of plotting the validation accuracy throughout training, we only plot the final validation accuracy. The plot shows that there is no significant difference between the different strategies, nor between the amount of overlap. Individual plots for the different data

assignment schemes with the validation accuracy plotted every 352 iteration (the number of iterations in an epoch when assigning disjoint data) can be seen in Figure B.1 in Appendix B.2.1.

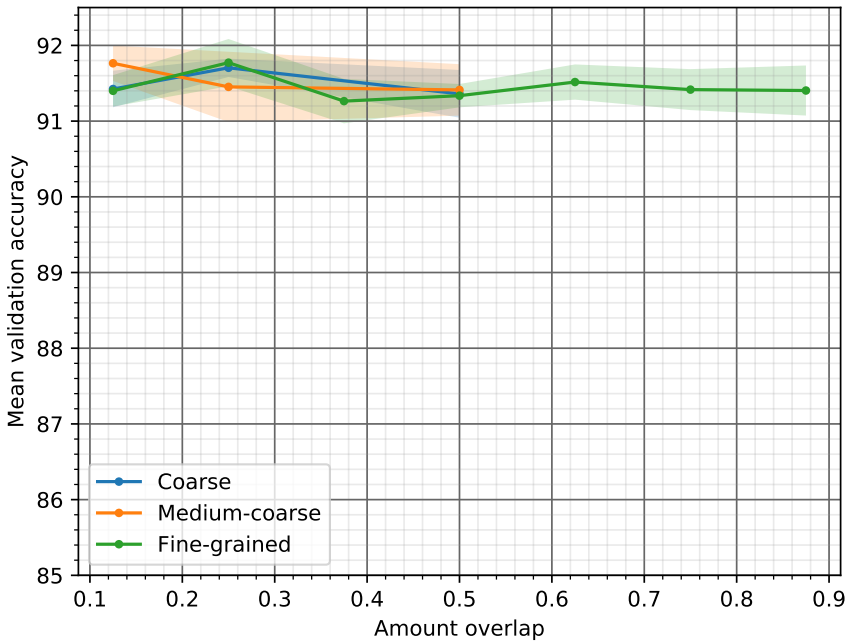


Figure 5.4: Mean final top-1 validation accuracy for the different data assignment schemes. Amount of overlap is quantified by $\frac{C}{M}$. The filled areas represents one standard deviation

5.2.3 E2.3 - Increase global batch size

Goal As described in Section 2.2.1, DNNs when trained using data parallelism are vulnerable to large global batches, in general, and small local batches when using BatchNorm. The goal of this experiment is to explore whether increasing the data available to each worker makes the model more resilient to an increase in global batch size independently of the learning rate (and thus, altering the batch size/learning rate relationship) when training fully synchronous.

Method & Data We experiment with global batch sizes $B_{global} \in \{128, 256, 512, 1024, 2048, 4096\}$. All experiments are run with 16 workers ($M = 16$), and for each global batch size, we have three separate variations of data assignment:

1. *Disjoint data*: we assign disjoint data shards.
2. *Full overlap*: we assign all data to all workers, resulting in $C = 16$ and $\frac{C}{M} = 1$.
3. *Coarse sharding*: we assign 2 shards using the coarse sharding scheme, resulting in $C = 8$ and $\frac{C}{M} = 0.5$.

For the runs with overlapping data, we ceil the number of overlap epochs to ensure that each data sample is processed equally many times, as discussed in Experiment E2.1.

Results & Discussion The results from this experiment are plotted in Figure 5.5 where the x-axis shows the global batch size. As expected, when increasing the global batch size, the accuracy degrades. In the figure, we see no observable difference for the different data assignment schemes up until global batch size 2048. At global batch size 4096, however, we see some difference: full overlap data assignment reaches a mean final validation accuracy 0.43 p.p. better than 2 shards (*i.e.,* $C = 8$, with coarse sharding), and 0.83 p.p. better than disjoint data assignment. We do, however, observe rather large standard deviations (see Table B.3 in Appendix B.2.2). To further study the difference of 0.83 p.p. between disjoint and full overlap, we conduct a two-sample one-tailed t-test with significance level $\alpha < 0.05$, and hypotheses:

- $\boldsymbol{H_0}$: There is no difference in final validation accuracy with $B_{global} = 4096$ when assigning overlapping data compared to full overlap data assignment.

- $\boldsymbol{H_1}$: Assigning full overlapping data is better than assigning disjoint data when $B_{global} = 4096$.

We find a p-value of $0.059 > 0.05$, and thus, we fail to reject the null hypothesis. We do, however, notice that the p-value is close to the significance level, and argue that doing more than 5 runs could change the outcome of this significance test. We leave this for future work.

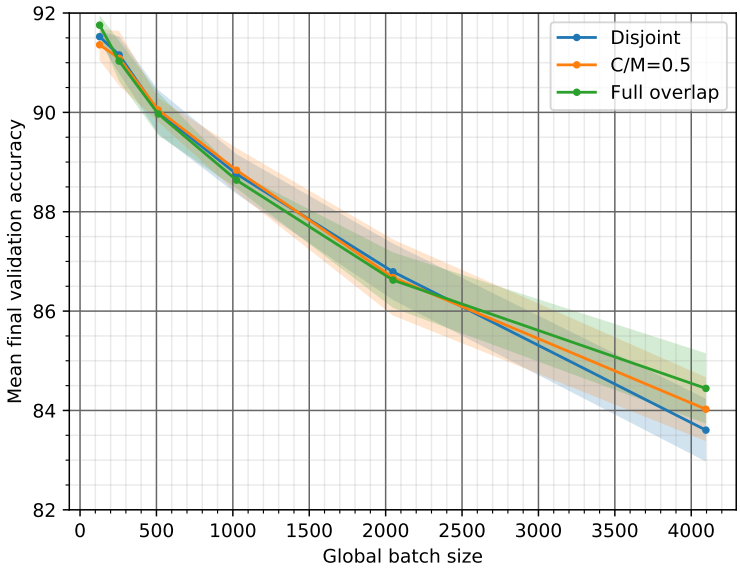


Figure 5.5: Increasing the global batch size with varying amount of data at each worker. Every experiment are run with 16 workers. The filled areas represents one standard deviation.

5.3 E3 - Communication reduction

In this section we experiment with communication reduction methods. More specifically, we experiment with local SGD, households and neighbourhoods (see Section 3.2 and Section 4.2). For the experiments in this section, we still study the general question of how different data assignment schemes impact the accuracy, namely **RQ1**. Since we reduce the number of communication rounds, we are also concerned with **RQ2**: we want to find out whether overlapping data can improve the accuracy compared to disjoint data assignment when reducing the number of communication rounds.

5.3.1 E3.1 - Local SGD

Goal As mentioned in Section 2.2.3, the model accuracy is negatively impacted when reducing the synchronization frequency. The goal of this experiment is to investigate the effects of assigning disjoint and full overlapping data when training when reducing the synchronization frequency by training with local SGD. As a part of this investigation, we want to find out whether increasing the amount of data available to each worker reduces the degradation in accuracy when reducing the number of communication rounds.

Method & Data We train with 2, 4, 8 and 16 workers, and use world synchronization periods $L_W \in \{2, 4, 8, \ldots, 128\}$. For each combination of L_W and M, we run two different experiments with the only difference being how the data is assigned: disjoint and full overlap. For the experiment were we assign overlapping data, we must find the number of overlap epochs. With full overlap, we have $C = M$, and thus, we do $E_{overlap} = \frac{E_{target}}{M}$ overlap epochs (with $E_{target} = 182$). When $E_{target} \mod M \neq 0$, we ceil the result to ensure that every data sample is processed equally many times, as discussed in Experiment E2.1.

Results & Discussion The results for this experiment are plotted in Figure 5.6. The results from assigning disjoint data are plotted as solid lines, and the result from assigning full overlapping data between the workers are plotted as dashed lines. As expected (see Section 2.2.3), the accuracy drops when we increase the world synchronization period. We also observe that this drop in accuracy gets larger as we increase the number of workers. We are, however, more concerned with the difference between the two data assignment schemes. As the

plot shows, there is no significant difference between the two data assignment schemes.

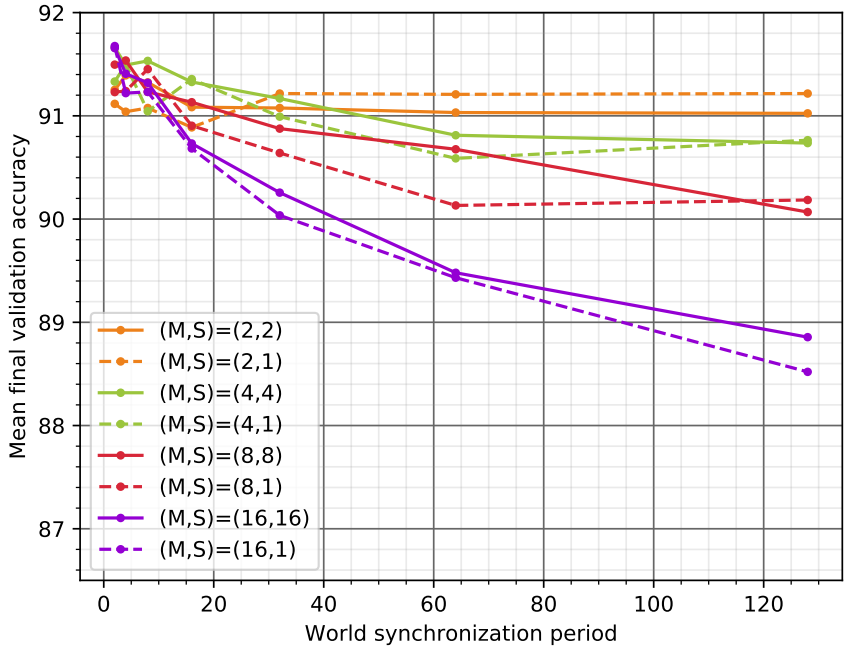


Figure 5.6: Mean final validation accuracy with different world synchronization period training with local SGD. Solid lines (S=M) represent disjoint data assignment, dashed lines (S=1) represent full overlap data assignment

5.3.2 E3.2 - Households

This section will present multiple experiments in which we explore the effects of dividing workers into households (see Section 4.2). In the first three experiments we assign disjoint household shards and explore three different ways of comparing households by altering some model hyperparameters. In the last experiment, we do the same alteration to the model hyperparameters, while also increasing the amount of data available to each household.

E3.2.1 - Constant local batch size

Goal The goal of this experiment is to explore the use of households with disjoint household shards (*i.e.,* $S_H = H$, see Section 4.2.1), and how different world synchronization periods impact the accuracy while keeping all model hyperparameters from the baseline experiment fixed.

Method & Data We use the same parameters for the model as in the baseline experiment. Three different number of households are tried, $H \in \{2, 4, 8\}$, and the number of workers are kept fixed at 16 ($M = 16$). We set $S_H{=}H$, which means that each household will get a unique household shard. For each value of H, we run with world synchronization periods $L_W \in \{2, 4, 8, \ldots, 128\}$. All experiments are run with household synchronization period of 1. To ensure synchronized parameters between all households, we do not validate at the end of an epoch if the number of iterations per epoch is not divisible by the world synchronization period. Instead, we run some extra iterations into the next epoch and validate at the first world synchronization in the next epoch.

Results & Discussion The results in Figure 5.7 show the mean final validation for different number of households and world synchronization periods. For 2 households, we can see that the validation accuracy does not suffer from increasing the world synchronization period up until 128. In fact, the validation accuracy for $L_W = 128$ at the end of training is better than the baseline. However, this difference (around 0.25 p.p.) is too small to be of any significance, and we could expect a more similar accuracy with more than 5 runs. When we increase the number of households, we see a more clear drop in the final validation accuracy. More specifically, we see that the decrease in final validation accuracy with 4 and 8 households compared to the baseline is 0.75 and 1.5 p.p., respectively.

One aspect of this comparison between different number of households is that we keep the number of iterations constant independent of the number of households and the world synchronization period. This means that as we increase the number of households while keeping the world synchronization period fixed, we do the same total number of world synchronizations. On the other hand the effective batch size is not the same for different values of H and L_W (see Appendix A and Equation (A.2)).

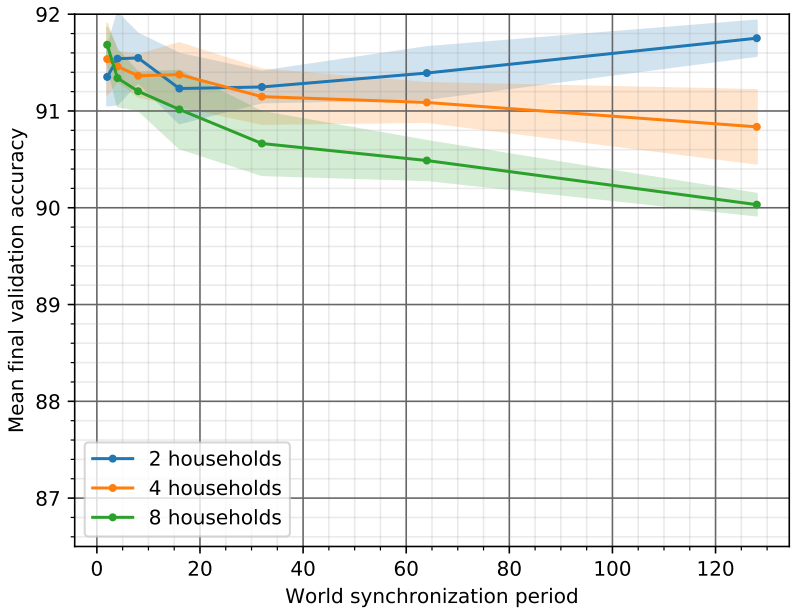


Figure 5.7: Final validation accuracy for different number of households with different world synchronization period with constant local batch size independent of H and L_W. Filled area represents one standard deviation.

E3.2.2 - Constant effective batch size

Goal As stated in Experiment E3.2.1, the effective batch size is not constant for different number of households and world synchronization periods when we keep the local batch size constant. The goal of this experiment is to expand upon Experiment E3.2.1 to enable further analysis. The expansion is done by comparing the validation accuracy for different number of households when we keep the effective batch size constant.

Method & Data We use the same parameters as in Experiment E3.2.1, but instead of dividing the global batch size by the number of workers to find the local batch sizes (*i.e.,* $B_{local} = \frac{B_{global}}{M}$), we set the target effective batch size to 128, and find new local batch sizes by using Equation (A.3). As seen in Figure A.1 in the Appendix, this leads to an increase in the local batch size which results in fewer iterations per epoch. Since we keep the total number of epochs fixed, the total number of iterations is not the same as in the baseline experiment. We keep the same LR schedule, but since the total number of iterations is changed, we do not decrease the LR at 32k and 48k iterations, but instead at 50% and 75% of total iterations, which will vary depending on H and L_W. We also see that for $H = 4$ with $L_W = 128$, and $H = 8$ with $L_W = 64$ and $L_W = 128$, we get larger world synchronization periods than the number of iterations per epoch. Thus, we do not validate every epoch, but instead only validate at every world synchronization to ensure synchronized parameters at validation.[1]

Results & Discussion The mean final validation accuracies are plotted in Figure 5.8, where the filled area around each line represents one standard deviation. We can see that there are a larger drop in accuracy between the different number of households compared to what we saw in Experiment E3.2.1 where we kept the local batch size constant. There are, however, a couple of factors that make the comparison between different number of households and world synchronization periods suboptimal: (1) since we have increased the local batch and kept the number of epochs fixed, we have fewer total iterations, which (2) also results in fewer total world synchronizations.[2] We also see a similar pattern

[1] There are ways to still validate at the same interval as with the baseline experiment. However, this would mean validating with different model parameters. One could for instance shard the validation data disjointly between the households, and validate with different model parameters at the end of every epoch.

[2] For example, with $H = 8$ and $L_W = 128$, we have $B_{local} = 61$ which leads to a total of $\lceil \frac{N}{M \cdot B_{local}} \rceil \cdot E = \lceil \frac{45000}{16 \cdot 61} \rceil \cdot 182 = 8554$ total iterations. This results in a total of $\lceil \frac{8554}{128} \rceil = 67$

in how the local batch increases with the world synchronization period (Figure A.1a in Appendix A) and the final validation accuracy decreases with the world synchronization period (Figure B.8). This similar pattern could imply that the increased local batch with its side-effects (fewer total iterations and world synchronizations) is a factor in the decreasing validation accuracy. It should also be noted that increasing the local batch size as a function of H and L_W results in a different batch size/learning rate relationship, something we hypothesize can impact the accuracy. This altered batch size/learning rate relationship motivates Experiment E3.2.3.

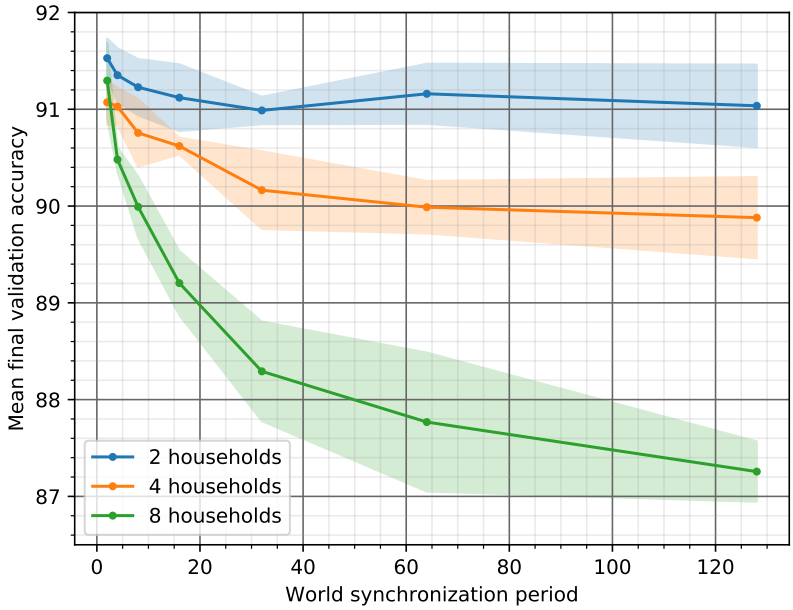


Figure 5.8: Mean final validation accuracy for different number of households with different world synchronization periods. Filled area represents one standard deviation.

world synchronizations, which is approximately 7.5 times less than in the experiment with constant local batch size with same H and L_W.

E3.2.3 - Constant effective batch size with linearly scaled learning rate

Goal As we saw in Experiment E3.2.2, when using Equation (A.3) to increase the local batch size and at the same time keeping the initial learning rate constant, the batch size/learning rate relationship is altered. We want this relationship to be constant and equal to what was used in the baseline experiment. The goal of this experiment is thus to further explore the effect of dividing workers into households with disjoint data where both the effective batch size and the batch size/learning rate relationship is constant. This is achieved by using Equation (A.3) to find local batch sizes in addition to applying the linear scaling rule [52].

Method & Data We use the same parameters as in Experiment E3.2.2, except for the initial learning rate: Based on the new local batch size from using Equation (A.3) with $B_{effective} = 128$, we scale the learning rate linearly with the increase in batch size. In the baseline the local batch size was

$$B_{local} = \frac{B_{global}}{M}$$

were B_{global} was set to 128. In this experiment we set the local batch size to

$$B_{local} = \frac{B_{effective} \cdot L_W \cdot H}{M(H + L_W - 1)},$$

and we see an increase in local batch size

$$\frac{B_{effective} \cdot L_W \cdot H}{128 \cdot (H + L_W - 1)}.$$

For this experiment we keep the effective batch size constant at $B_{effective} = 128$. Thus we end up with scaling the initial learning rate by

$$\frac{L_W \cdot H}{H + L_W - 1}$$

As explained by Goyal *et al.* [52], applying the linear scaling rule without any other techniques typically results in early optimization difficulties. Thus, we also apply their proposed learning rate warmup period of 5 epochs where we gradually increase the learning rate from 0.1 to the target scaled learning rate to overcome this issue. Even though we change the initial learning rate, we keep the same learning rate schedule, which means that we divide the learning rate by 10 at 50% and 75% of training.

Results & Discussion The mean final validation accuracies for this experiment are plotted in Figure 5.9. Comparing this plot to the result plotted in Figure 5.8 from Experiment E3.2.2, we can see a general improvement in accuracy. This could indicate that the altered relationship between batch size and learning rate in Experiment E3.2.2 resulted in some degradation in accuracy. With the assumption that scaling the learning rate fixes this degradation in accuracy, we can argue that the remaining degradation (compared to Experiment E3.2.1 where we keep the local batch size constant independent of H and L_W) is a result of fewer total number of iterations and world synchronizations, as well as the decreased iteration-to-sync ratio $\frac{T_E}{L_W}$, when increasing the local batch size.

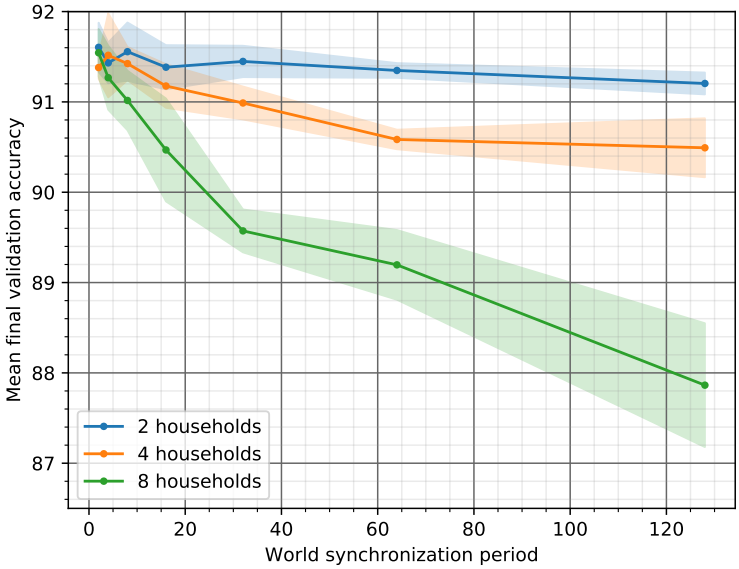


Figure 5.9: Mean final validation accuracy for different number of households with different world synchronization periods where we scale the learning rate linearly. Filled area represents one standard deviation.

E3.2.4 - Overlapping data between households

Goal The goal of this experiment is to explore the effects of overlapping data between households (*i.e.,* $S_H < H$, see Section 4.2.1) in combination with different batch/size learning rate relationships. We want to find out whether assigning overlapping data when using any of these batch size/learning rate combinations can improve the accuracy compared to disjoint data assignment.

Method & Data We divide 16 workers into 2, 4 and 8 households. We further set the number of household shards $S_H < H$ (*i.e.,* overlapping household shards, see Figure 4.7 for an example where H=4 and S_H=2). For H=2 we have S_H=1, for H=4 and H=8 we have $S_H \in \{1, 2\}$. For each of the previous combinations of H and S_H, we run with world synchronization periods $L_W \in \{2, 4, 8, \ldots, 128\}$. All of these combinations of H, S_H and L_W are run with three combinations of local batch size and learning rate:

1. We keep the local batch size constant with $B_{local} = 8$ with an initial learning rate of 0.1 (as in Experiment E3.2.1)
2. We find the local batch size using Equation (A.3) with $B_{effective} = 128$, with an initial learning rate of 0.1 (as in Experiment E3.2.2)
3. We find the local batch size using Equation (A.3) with $B_{effective} = 128$, and scale the initial learning rate linearly (as in Experiment E3.2.3)

Since we have overlapping data between the households, we must find the number of overlap epochs. We first find the number of copies of each data sample with $C = \frac{H}{S_H}$. The number of overlap epochs is then given by $E_{overlap} = \lceil \frac{E_{target}}{C} \rceil$ where E_{target} is the target number of epochs. The reason it is ceiled is to ensure that each data sample is processed equally many times.

Results & Discussion The results of the three combinations of local batch size and learning rate listed above are plotted in Figure 5.10, Figure 5.11 and Figure 5.12. For easier comparison to the results where we assign disjoint data (Experiment E3.2.1, Experiment E3.2.2 and E3.2.3), we plot these results with solid lines, while plotting the results for overlapping data with striped and dashed lines.

From Figure 5.10 we can see that there is practically no difference between overlapping household shards and disjoint household shards when we keep the local

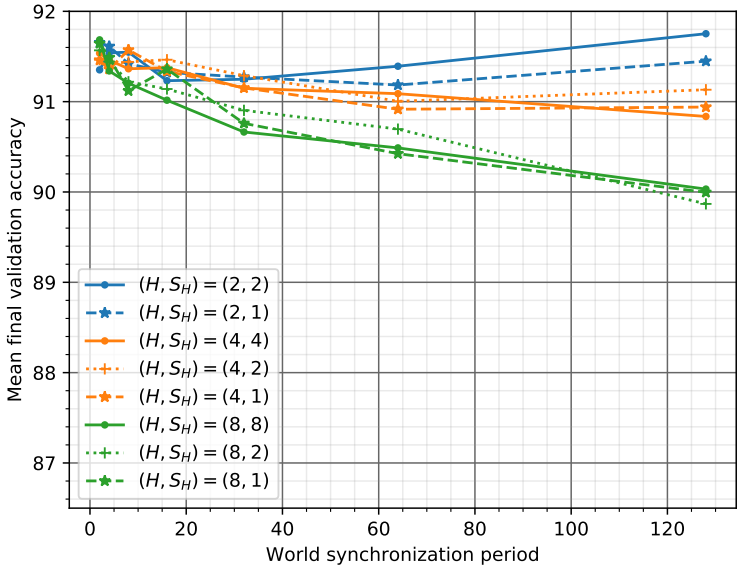


Figure 5.10: Mean final validation accuracy for different number of households and household shards. Local batch size is kept constant at $B_{local} = \frac{B_{global}}{M}$, with an initial learning rate of 0.1

batch size constant. When we increase the local batch to reach an effective batch size of 128 while keeping the initial learning rate at 0.1, we can see that overlapping household shards can improve the accuracy when the world synchronization period gets large. From Figure 5.11, we can see that this improvement is most significant for $H = 8$ and $L_W > 16$. To further study this difference, we list the numbers for these results in Table 5.1 on the format "(mean ± std)". As the table shows, the biggest difference in mean validation accuracy is obtained for L_W=64, between S_H=8 and S_H=2. We do, however, observe some difference in standard deviation between the different combinations of L_W and S_H. To investigate the significance of the differences, we run a two-sample one-tailed t-test with significance level $\alpha < 0.05$, and hypotheses:

- $\boldsymbol{H_0}$: Overlapping data between households has no effect compared to disjoint data assignment.
- $\boldsymbol{H_1}$: Overlapping data between households is better, with regards to final validation accuracy, than disjoint data assignment.

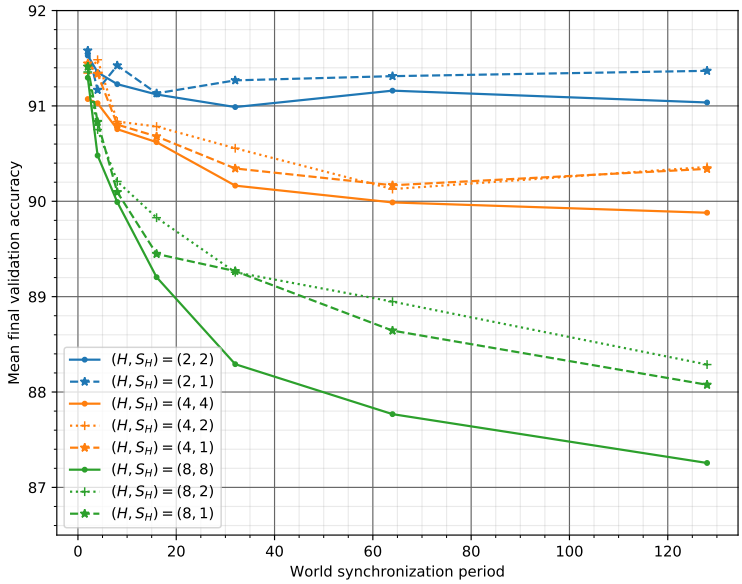


Figure 5.11: Mean final validation accuracy for different number of households and household shards. Local batch size is given by Equation (A.3), with an initial learning rate of 0.1

We find that all comparisons between disjoint (*i.e.,* S_H=8) and overlapping data (*i.e.,* $S_H \in \{1, 2\}$) where $H = 8$ and $L_W > 16$ results in a p-value < 0.05. Thus, we reject the null hypothesis for these world synchronization periods (see Table 5.2 for the specific p-values for these world synchronization periods). For H=8, we further find a p-value < 0.05 when $L_W = 16$ and S_H=2 (but not for S_H=1). We also find a p-value < 0.05 for $H = 4$ with $L_W = 128$, both with S_H=1 and S_H=2.

Looking at the results plotted in Figure 5.12 in which we scale the learning rate linearly with the increase in local batch size when using Equation (A.3), we see that there is no longer any significant difference in mean final validation accuracy when assigning overlapping household shards, compared to disjoint data assignment.[3] The biggest difference in validation accuracy between disjoint and

[3]However, as discussed in Experiment 5.3.2, there is a general improvement in valida-

L_W \ S_H	8	2	1
32	(88.29 ± 0.51)	(89.26 ± 0.42)	(89.27 ± 0.15)
64	(**87.77** ± 0.72)	(**88.95** ± 0.21)	(88.64 ± 0.38)
128	(87.26 ± 0.31)	(88.29 ± 0.13)	(88.08 ± 0.30)

Table 5.1: Final validation accuracy for 8 households with mean and standard deviation (std) over 5 runs on the format "(mean ± std)". Bold number represent the biggest difference in mean between data assignment schemes

L_W \ S_H	2	1
32	0.014	0.007
64	0.0113	0.0375
128	**0.001**	0.0065

Table 5.2: Resulting p-values from running a two-sample t-test comparing disjoint (S_H=8) to overlapping data assignment ($S_H \in \{1, 2\}$) with numbers from Table 5.1. Bold number represents smallest p-value

overlapping data is observed for H=8 with L_W=128, where we see a difference of 0.57 p.p. between S_H=8 and S_H=2. Running a two-sample one-tailed t-test with the same significance level and hypotheses described earlier in this section, we get a p-value of $0.09 > 0.05$, failing to reject the null hypothesis.

To reiterate our results, we find that the experiment where the effective batch size is kept constant while the learning rate is not scaled based on the increase in batch size results in an improvement in final validation accuracy when assigning overlapping data between households compared to disjoint data assignment. For the two remaining combinations of batch size and learning rate, we see no significant difference between overlap and disjoint data assignment.

Finding conclusive answers to some of the results from these experiments is hard. The act of arranging workers into households and reducing the communication during training is not alone the cause of the difference in accuracy. In addition, our results also show that using households and increasing the batch size when the initial learning rate is scaled with this increase does not lead to this difference either. Further, the difference is not solely caused by the altered relationship between batch size and learning rate, as it did not appear in Experiment E2.3. All

tion accuracy for both disjoint and overlapping data compared to Experiment E3.2.2 with $B_{effective}$=128 where we keep the initial learning rate at 0.1.

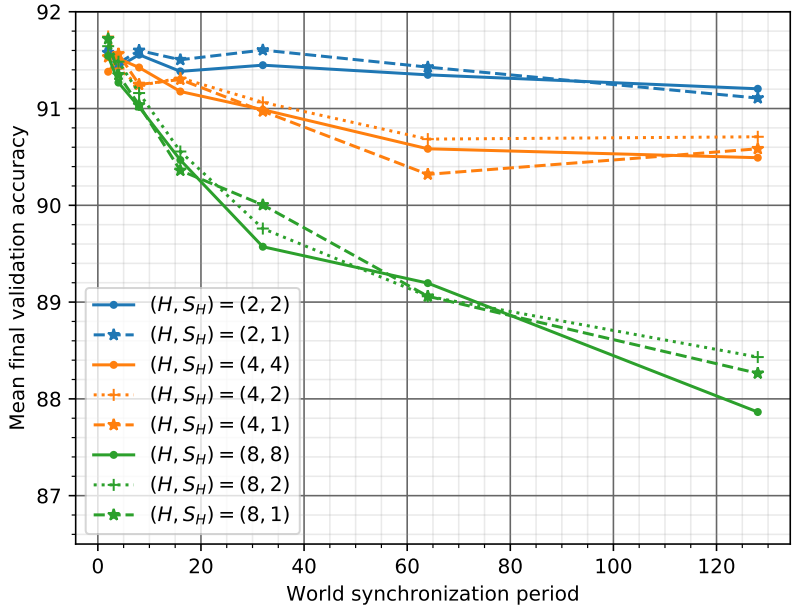


Figure 5.12: Mean final validation accuracy for different number of households and household shards. Local batch is given by Equation (A.3), and the initial learning rate is scaled linearly with the increase in local batch size

these experiments that were conducted that did not see this difference in accuracy between disjoint and overlapping data greatly limits the number of possible explanations for the observed difference. Since we only observe the difference at large H and L_W, which results in larger local batch when using Equation (A.3), we argue that the batch size/learning rate relationship must be altered to a certain degree before overlapping data has an effect. Even though a specific combination of hyperparameters resulted in a difference in accuracy between disjoint and overlapping data, there may not exist a general rule as to what kind of hyperparameter combinations will result in such a difference. As a side note, the value of these results is also somewhat limited by the fact that the experiment that had this difference showed the largest degradation of accuracy out of all our experiments, at worst falling below 88%.

5.3.3 E3.3 - Neighbourhoods

Goal The goal of this experiment is to explore whether there is any difference of assigning overlapping data between neighbourhood compared to assigning fully disjoint data.

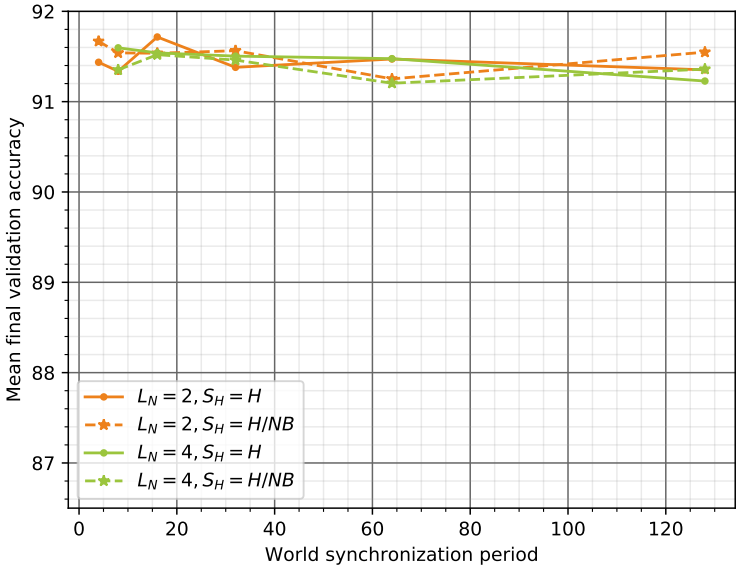
Method & Data We divide 16 workers ($M = 16$) into 4 and 8 households ($H = 4$ and $H = 8$). We further divide these households into 2 neighbourhoods ($NB = 2$) and set the household synchronization period $L_H = 1$. We experiment with two different neighbourhood synchronization periods, $L_N = 2$ and $L_N = 4$. For $L_N = 2$, we run with world synchronization periods $L_W \in \{4, 8, 16, \ldots, 128\}$, and for $L_N = 4$ we run with $L_W \in \{8, 16, 32, \ldots, 128\}$. The data is assigned in two different ways:

- *Disjoint*: All workers are given a disjoint shard, and thus, there are no overlap between neither households nor neighbourhoods
- *Full overlap between neighbourhoods*: We assign overlapping household shards in such a way that each neighbourhood will have one copy of each household shard (an example where H=4, NB=2 and $S_H = 2$ is shown in Figure 4.8). Specifically, we set $S_H = \frac{H}{NB}$ and assign the household shards in such way that there are no overlap *within* a neighbourhood, but full overlap *between* neighbourhoods

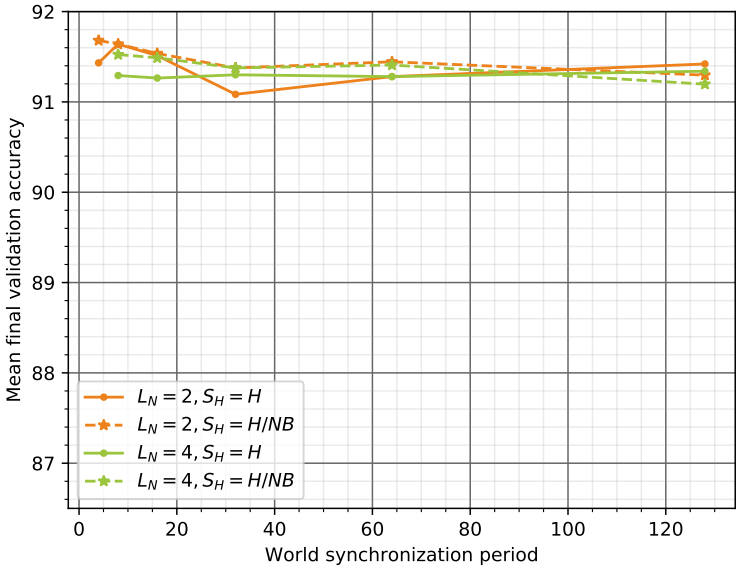
For the experiment with overlapping data, we set the number of overlap epochs $E_{overlap} = \lceil \frac{E_{target}}{C} \rceil$. Since we have full overlap between neighbourhoods, but disjoint data within each neighbourhood we have $C = NB$. Thus, we do $E_{overlap} = \lceil \frac{182}{2} \rceil = 91$ overlap epochs.

Results & Discussion The mean final validation accuracy for different combinations of H, L_N and L_W for this experiment are plotted in Figure 5.13. We plot neighbourhoods with disjoint data as solid lines and neighborhoods with overlapping data as dashed lines. The main observation is that there is no significant difference between disjoint and overlapping data between neighbourhoods. We further observe that the results for both H=4 (Figure 5.13a) and H=8 (Figure 5.13b) are very similar, with no significant difference when comparing pairs of L_N and L_W. Next, we will look at a couple of points that we observe for both H=4 and H=8.

- *There is no significant difference between the two neighbourhood synchronization periods*: We observe that this is especially the case for larger world synchronization periods.
- *The model is quite resilient against an increase in world synchronization period*: That is, we observe no significant difference between the smallest and largest world synchronization periods.



(a) 4 households in 2 neighbourhoods



(b) 8 households in 2 neighbourhoods

Figure 5.13: Households arranged into 2 neighbourhoods. Solid lines show results where each worker is given a disjoint data shard, and dashed lines show results where there are full overlap between neighbourhoods.

5.4 Summary

In this section, for the results that we consider most significant for the key message of this thesis, we will run the model on the test set to either confirm or reject that the validation set is representative for the test set.

5.4.1 Varying degrees of overlap

The results from running Experiment E2.2 are plotted in Figure 5.14 with the y-axis showing the test accuracy at end of training. We observe that there is no significant difference between the test accuracy and the validation accuracy (Figure 5.4).

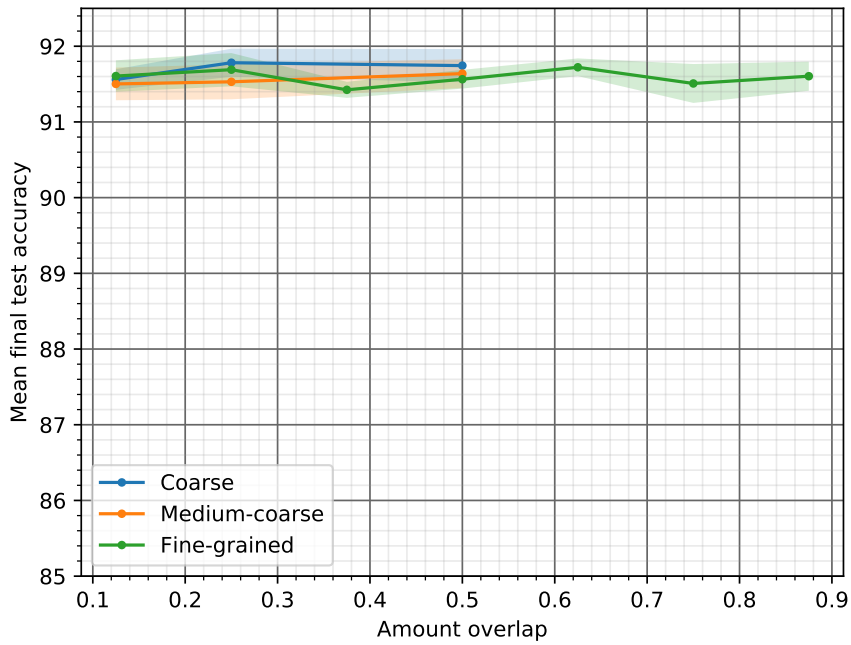


Figure 5.14: Mean test accuracy for different data assignment schemes. Filled area represents one standard deviation

5.4.2 Households with overlapping data

The results from running the batch size/learning rate combination (2) described in Experiment E3.2.4 are plotted training in Figure 5.15 with test accuracy at end of training. Here, we also find that there is no significant difference between the test accuracy and the validation accuracy (Figure 5.11).

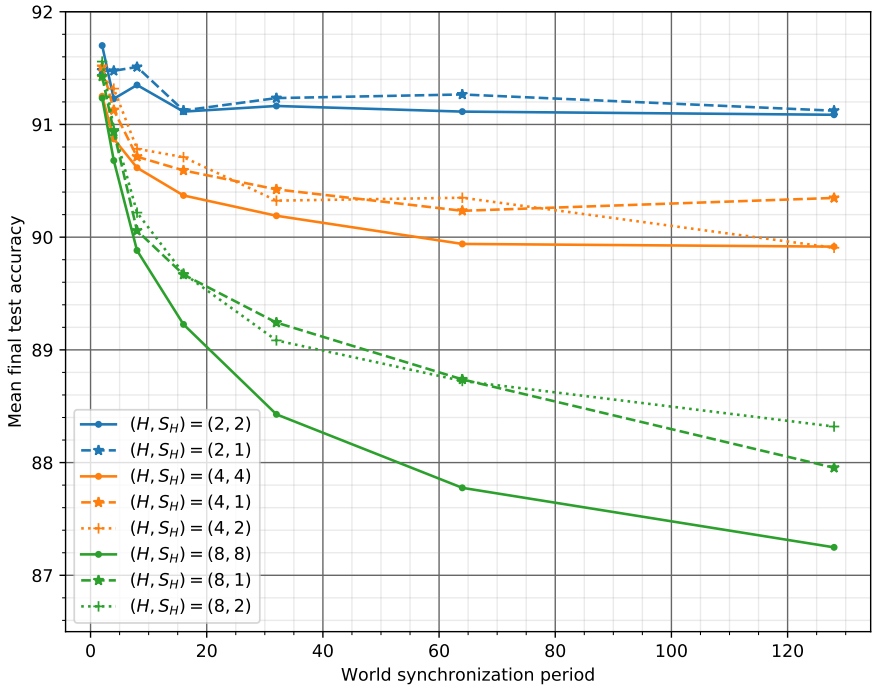


Figure 5.15: Mean test accuracy for households with disjoint and overlapping data where the effective batch size is kept constant at 128

Chapter 6

Evaluation and Conclusion

In this thesis we have conducted multiple experiments were we trained a convolutional neural network using data parallelism. The overall goal of the work has been to explore the effects of different data assignment schemes in terms of final accuracy. We have experimented with both fully synchronous systems, as well as methods that reduce the number of communicating rounds. In this chapter, we will conclude the thesis by evaluating and discussing the main findings of the work presented in the previous chapters. This includes a discussion where we will address the research questions stated in Chapter 1. We will also discuss the limitations of this work which will motivate possible directions for future work.

6.1 Evaluation

With fully synchronous data parallel training we find that there is no difference in terms of final accuracy between giving every worker access to all data and giving each worker a unique part of the dataset. When assigning data somewhere in between these two extremes, we also find that there is no difference in final accuracy regardless of the amount of data available to each worker. This finding applies for three different data assignment schemes where the intersection of data between there workers differ. Further, we have explored the effects of assigning varying amount of data to each worker when we reduce the number of communicating rounds. Here, we find that in certain circumstances, overlapping data can improve the final accuracy compared to disjoint data. Specifically, this

improvement is found when dividing workers into households and the local batch size is increased without changing the learning rate, leading to an altered batch size/learning rate relationship. However, since we only see this improvement for larger local batch sizes, we assert that the relationship must be altered to a certain degree for there to be any improvement with overlapping data. In a different experiment, we show that assigning overlapping data has no significant effect when we train synchronously with an altered batch size/learning rate relationship. We therefore argue that the improvement stems from the combination of reduced communication rounds and an altered batch size/learning rate relationship.

To sum up, we will answer the research questions stated in Section 1.2. For convenience, we will repeat the questions in this section, starting with **RQ1**:

RQ1 *In terms of performance, what are the effects of assigning the data in different ways between the workers?*

Most of the results generated in this project support the fact that there is no difference in performance between different data assignment schemes. As mentioned in the previous paragraph we find one exception in which this is not the case. This exception is found when reducing the number communication rounds while altering the batch size/learning rate relationship to a certain degree. This specific scenario also helps us answer **RQ2**:

RQ2 *Can overlapping data make the system more resilient to communication reduction? If so, in what circumstances?*

We find that overlapping data in many cases does not make the model more resilient to communication reduction. However we did find one scenario where it had a significant improvement over disjoint assignment. The specifics of this scenario are briefly mentioned in the previous paragraph, and are fully elaborated on in Experiment 3.2.4]. Finally, we will address **RQ3**:

RQ3 *What is the optimal way to assign data between workers?*

When it comes to fully synchronous data parallel training, we observe no difference in final accuracy between any of the strategies that assign overlapping data. For these reasons, we think it is reasonable to view the optimal strategy from a practical standpoint, but finding the most practical strategy is outside the scope

of this project. When it comes to evaluating an optimal way of assigning data when reducing the number of communication rounds, we also find no significant difference between disjoint and overlapping data assignment in most scenarios, therefore we again make the claim that the most optimal way to assign data should be seen from a practical standpoint. We do, however, see that assigning overlapping data is better than disjoint data whenever the batch size/learning rate relationship is altered to a certain degree in combination with significant reduction in communication. Thus, we argue that overlapping data can be the optimal way of assigning data in this specific scenario. We finally note that this is based on the findings in this work, and may not apply for other workloads and communication methods.

6.2 Contributions

The work done in this thesis has given three main findings:

- When reducing the number of communication rounds, assigning overlapping data can improve accuracy compared to disjoint data assignment when the batch size/learning rate relationship is altered to a certain degree.
- There is no difference in final accuracy with regards to neither the amount of data, nor the degree intersection of data between workers when training is fully synchronous.
- In many cases assigning overlapping data has no impact on final accuracy, even when reducing the amount of communication between workers during training.

6.3 Discussion

In this section we will discuss what we consider as the most notable limitations of this work. The discussion will serve as potential directions for future work, which will be pointed out in Section 6.4.

6.3.1 Workload

For the experiments conducted in this thesis, we have set the scope at a certain workload, *i.e.,* neural network architecture, dataset, hyperparameters and optimizer, and we have conducted multiple experiments with focus on data assignment schemes with this specific workload. We have, however, altered some of the hyperparameters to get a broader view and a wider range of comparisons for some of the methods. It should be noted that there might be other alterations to the workload that could lead to new and interesting results. Below, we list some possible alterations to the workload:

- **Architecture and data**: We have experimented with a specific convolutional neural network architecture, trained on spatial data (more specifically, images with three channels). An interesting direction of study would be to explore other deep learning architectures and types of data. For instance, recurrent models trained on time series data, with focus on different data assignment schemes, could be an interesting research direction.
- **Optimizer**: All experiments conducted in this project used SGD with a momentum coefficient of 0.9. Since we found a significant difference between disjoint and overlapping data in the specific scenario in which the batch size/learning rate relationship was altered when reducing the number of communication rounds, experiments conducted with an adaptive learning rate optimizer (*e.g.,* RMSProp or Adam) could provide more results to further investigate this scenario.
- **Hyperparameters**: For the majority of our experiments, we have kept all hyperparameters constant with the only alteration being data assignment scheme. In the experiments in which did alter some hyperparameters, we only altered the batch size and learning rate. We note that there are other hyperparameters that could be changed, *e.g.,* weight decay, momentum coefficient and learning rate schedule.[1]

6.3.2 Communication reduction methods

Most of the experiments on communication reduction methods conducted in this project used the concept of households, where we conducted multiple experiments

[1] If the optimizer is changed, there are potentially other hyperparameters than the momentum coefficient.

with alterations to batch size and learning rate to get more comparisons and a broader view. For local SGD and neighbourhoods, we have not conducted the same extensive amount of experiments. We hypothesize that some of the findings for households could also hold for local SGD and neighbourhoods due to the similar nature of these communication reduction methods. We also note that for the communication reduction methods experimented with in this project, we have only experimented with varying degrees of overlap, and have not been concerned with the intersection of data between workers/households/neighbourhoods. Even though the different data assignment schemes described in Section 4.1 showed no difference in terms of final accuracy when training fully synchronous, a possible extension for future research would thus be to apply these schemes to achieve varying degrees of intersection between workers/households/neighbourhoods when reducing communication. At last, we note that communication reduction when training deep neural networks with data parallelism is a large field and we have have focused on a specific method, namely reducing the number of workers involved in a synchronization as well as reducing the total number of synchronizations. Other types of communication reduction methods might yield other interesting results when exploring different kinds of data assignment schemes.[2]

6.4 Future Work

Based on the discussed limitations of this work in Section 6.3, we consider the following as the most notable future work:

- Apply different data assignment schemes to other neural network architectures and data types.
- Further explore the altered relationship between batch size and learning rate by for instance using an adaptive learning rate optimizer, as well as exploring the effects of other hyperparameters in combination of assigning overlapping data.

[2]One example would be to do gradient quantization and sparsification (see Section 2.2.4).

Bibliography

[1] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups", *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, Nov. 2012, ISSN: 1558-0792. DOI: 10.1109/MSP.2012.2205597.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks", in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[3] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch", *J. Mach. Learn. Res.*, vol. 12, pp. 2493–2537, Nov. 2011, ISSN: 1532-4435. [Online]. Available: http://dl.acm.org/citation.cfm?id=1953048.2078186.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[5] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis", *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, p. 65, 2019.

[6] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution", *IEEE Micro*, vol. 38, no. 2, pp. 21–29, 2018.

[7] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey and benchmarking of machine learning accelerators", *arXiv preprint arXiv:1908.11348*, 2019.

[8] D. Amodei and D. Hernandez, *Ai and compute*, OpenAI, Ed., Accessed 12-November-2019, May 2018. [Online]. Available: https://openai.com/blog/ai-and-compute/.

[9] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, *et al.*, "Large scale distributed deep networks", in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[10] B. Birkeland and A. Håland, "Machine learning at the exaflop scale and beyond", Department of Computer Science, NTNU – Norwegian University of Science and Technology, Project report in TDT4501, Dec. 2019, The report is available from one of the authors.

[11] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[12] D. E. Rumelhart, G. E. Hinton, R. J. Williams, *et al.*, "Learning representations by back-propagating errors", *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[13] H. Robbins and S. Monro, "A stochastic approximation method", *The annals of mathematical statistics*, pp. 400–407, 1951.

[14] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude", *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[15] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, cite arxiv:1412.6980 Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015, 2014. [Online]. Available: `http://arxiv.org/abs/1412.6980`.

[16] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh, *Large batch optimization for deep learning: Training bert in 76 minutes*, 2019. arXiv: `1904.00962 [cs.LG]`.

[17] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network", in *Advances in neural information processing systems*, 1990, pp. 396–404.

[18] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks", *arXiv preprint arXiv:1404.5997*, 2014.

[19] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima", 2016, cite arxiv:1609.04836 Comment: Accepted as a conference paper at ICLR 2017. [Online]. Available: `http://arxiv.org/abs/1609.04836`.

[20] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15, Lille, France: JMLR.org, 2015, pp. 448–456. [Online]. Available: `http://dl.acm.org/citation.cfm?id=3045118.3045167`.

[21] Y. Wu and K. He, "Group normalization", in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 3–19.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[23] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "ImageNet large scale visual recognition challenge", *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[24] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server", in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 583–598.

[25] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters", *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[26] A. Agarwal, O. Chapelle, M. Dudık, and J. Langford, "A reliable effective terascale linear learning system", *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1111–1133, 2014.

[27] *The Message Passing Interface (MPI) standard*, Accessed: 11-11-2019. [Online]. Available: `https://www.mcs.anl.gov/research/projects/mpi/`.

[28] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations", *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[29] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-SGD for distributed deep learning", in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16, New York, New York, USA: AAAI Press, 2016, pp. 2350–2356, ISBN: 978-1-57735-770-4. [Online]. Available: `http://dl.acm.org/citation.cfm?id=3060832.3060950`.

[30] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing, "Solving the straggler problem with bounded staleness", in *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[31] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD", in *International Conference on Learning Representations Workshop Track*, 2016. [Online]. Available: `https://arxiv.org/abs/1604.00981`.

[32] D. Povey, X. Zhang, and S. Khudanpur, "Parallel training of deep neural networks with natural gradient and parameter averaging", *arXiv preprint arXiv:1410.7455*, 2014.

[33] X. Zhang, J. Trmal, D. Povey, and S. Khudanpur, "Improving deep neural network acoustic models using generalized maxout networks", in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2014, pp. 215–219.

[34] J. Zhang, C. De Sa, I. Mitliagkas, and C. Ré, "Parallel SGD: When does averaging help?", *arXiv preprint arXiv:1606.07365*, 2016.

[35] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent", in *Advances in neural information processing systems*, 2010, pp. 2595–2603.

[36] Y. Zhang, M. J. Wainwright, and J. C. Duchi, "Communication-efficient algorithms for statistical optimization", in *Advances in Neural Information Processing Systems*, 2012, pp. 1502–1510.

[37] S. U. Stich, "Local SGD converges fast and communicates little", *arXiv preprint arXiv:1805.09767*, 2018.

[38] T. Lin, S. U. Stich, K. K. Patel, and M. Jaggi, "Don't use large mini-batches, use local SGD", *arXiv preprint arXiv:1808.07217*, 2018.

[39] S. Zhang, A. E. Choromanska, and Y. LeCun, "Deep learning with elastic averaging SGD", in *Advances in Neural Information Processing Systems*, 2015, pp. 685–693.

[40] M. Blot, D. Picard, M. Cord, and N. Thome, "Gossip training for deep learning", *arXiv preprint arXiv:1611.09726*, 2016.

[41] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns", in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[42] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient SGD via gradient quantization and encoding", *arXiv preprint arXiv:1610.02132*, 2017.

[43] N. Strom, "Scalable distributed DNN training using commodity GPU cloud computing", in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[44] A. F. Aji and K. Heafield, "Sparse communication for distributed gradient descent", *arXiv preprint arXiv:1704.05021*, 2017.

[45] L. Bottou, "Curiously fast convergence of some stochastic gradient descent algorithms", in *Proceedings of the symposium on learning and data science, Paris*, 2009.

[46] Q. Meng, W. Chen, Y. Wang, Z.-M. Ma, and T.-Y. Liu, "Convergence analysis of distributed stochastic gradient descent with shuffling", *arXiv preprint arXiv:1709.10432*, 2017.

[47] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ML via a stale synchronous parallel parameter server", in *Advances in neural information processing systems*, 2013, pp. 1223–1231.

[48] R. Mayer and H.-A. Jacobsen, "Scalable deep learning on distributed infrastructures: Challenges, techniques and tools", *arXiv preprint arXiv:1903.11314*, 2019.

[49] S. Gupta, W. Zhang, and F. Wang, "Model accuracy and runtime tradeoff in distributed deep learning: A systematic study", in *2016 IEEE 16th International Conference on Data Mining (ICDM)*, IEEE, 2016, pp. 171–180.

[50] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network", in *NIPS Deep Learning and Representation Learning Workshop*, 2015. [Online]. Available: `http://arxiv.org/abs/1503.02531`.

[51] R. Anil, G. Pereyra, A. Passos, R. Ormandi, G. E. Dahl, and G. E. Hinton, "Large scale distributed neural network training through online distillation", *arXiv preprint arXiv:1804.03235*, 2018.

[52] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour", *arXiv preprint arXiv:1706.02677*, 2017.

[53] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le, "Don't decay the learning rate, increase the batch size", *arXiv preprint arXiv:1711.00489*, 2017.

[54] V. Codreanu, D. Podareanu, and V. Saletore, "Scale out for large minibatch sgd: Residual network training on ImageNet-1K with improved accuracy and reduced time to train", *arXiv preprint arXiv:1711.04291*, 2017.

[55] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely large minibatch SGD: Training ResNet-50 on ImageNet in 15 minutes", *arXiv preprint arXiv:1711.04325*, 2017.

[56] Y. You, I. Gitman, and B. Ginsburg, "Large batch training of convolutional networks", *arXiv preprint arXiv:1708.03888*, 2017.

[57] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "ImageNet training in minutes", in *Proceedings of the 47th International Conference on Parallel Processing*, ACM, 2018, p. 1.

[58] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, *et al.*, "Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes", *arXiv preprint arXiv:1807.11205*, 2018.

[59] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng, "Image classification at supercomputer scale", *arXiv preprint arXiv:1811.06992*, 2018.

[60] H. Mikami, H. Suganuma, P. U-chupala, Y. Tanaka, and Y. Kageyama, "Massively distributed SGD: ImageNet/ResNet-50 training in a flash", *arXiv preprint arXiv:1811.05233*, 2018.

[61] S. Zagoruyko and N. Komodakis, "Wide residual networks", in *Proceedings of the British Machine Vision Conference (BMVC)*, E. R. H. Richard C. Wilson and W. A. P. Smith, Eds., BMVA Press, Sep. 2016, pp. 87.1–87.12, ISBN: 1-901725-59-6. DOI: 10.5244/C.30.87. [Online]. Available: https://dx.doi.org/10.5244/C.30.87.

[62] A. Krizhevsky, "Learning multiple layers of features from tiny images", 2009.

[63] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[64] J. Wang and G. Joshi, "Adaptive communication strategies to achieve the best error-runtime trade-off in local-update SGD", *arXiv preprint arXiv:1810.08313*, 2018.

[65] T. G. Dietterich, "Ensemble methods in machine learning", in *Proceedings of the First International Workshop on Multiple Classifier Systems*, ser. MCS '00, London, UK, UK: Springer-Verlag, 2000, pp. 1–15, ISBN: 3-540-67704-6. [Online]. Available: http://dl.acm.org/citation.cfm?id=648054.743935.

[66] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, *et al.*, "Ray: A distributed framework for emerging AI applications", in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 561–577.

[67] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: `http://tensorflow.org/`.

[68] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, *EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure*, 2019. arXiv: `1912.05848 [cs.DC]`.

[69] N. Qian, "On the momentum term in gradient descent learning algorithms", *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

[70] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

Appendices

Appendix A

Household effective batch size

The standard way of calculating local batch sizes in distributed environments is by dividing the global batch size B_g by the number of workers M, given by

$$B_l = \frac{B_g}{M}. \tag{A.1}$$

When decreasing the synchronization frequency, we observe that the effective batch size is not equal to the global batch size. By effective batch size, we mean the average number of samples that are synchronized at one iteration from the viewpoint of one worker. When using households with household synchronization period of 1, we define the effective batch size B_e as

$$B_e = \frac{B_h \cdot (L_W - 1) + B_h \cdot H}{L_W}, \tag{A.2}$$

where B_h is the household batch size (that is, the number of samples involved in a household synchronization), H is the number of households, and L_W is the world synchronization period. The first term in the numerator, $B_h \cdot (L_W - 1)$, is the number of samples involved in all household synchronizations before a world synchronization. The second term in the numerator, $B_h \cdot H$, is the number of samples involved in a world synchronization. This is divided by the total number of iterations involved in a world synchronization to get the effective batch size. The equation can further be simplified to

$$B_e = \frac{B_h(H + L_W - 1)}{L_W},$$

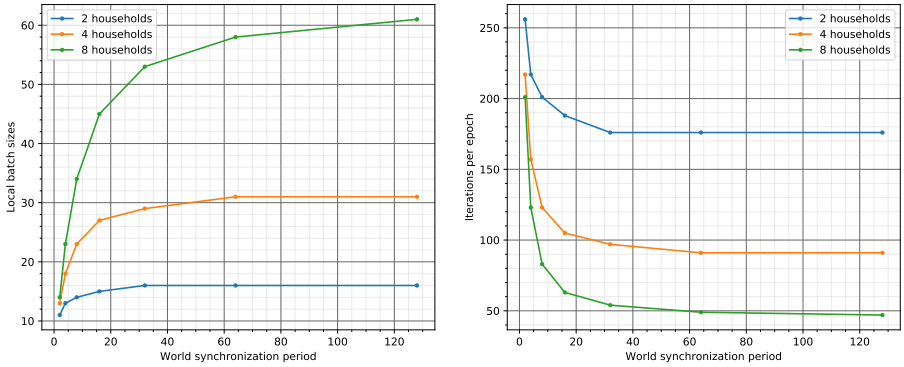
and if we set $B_h = \frac{B_l \cdot M}{H}$, where B_l is the local batch size and M is the number of workers, we have

$$B_e = \frac{B_l \cdot M(H + L_W - 1)}{H \cdot L_W}.$$

We can then solve for B_l

$$B_l = \frac{B_e \cdot L_W \cdot H}{M(H + L_W - 1)}. \tag{A.3}$$

With this formula, we can insert a target effective batch size and get local batch sizes to reach this target effective batch size. As seen in Figure A.1a, the local batch sizes are larger than when using the default local batch size calculation given by Equation (A.1), which results in $B_l = 8$ when $B_g = 128$ and $M = 16$. Since the local batch size is bigger when using households, we have fewer iterations per epoch, as shown by the plot in Figure A.1b.



(a) Local batch sizes

(b) Iterations per epoch

Figure A.1: Household parameters with $B_e = 128$ and $M = 16$, using Equation (A.3) to find local batch sizes

Appendix B

Additional results

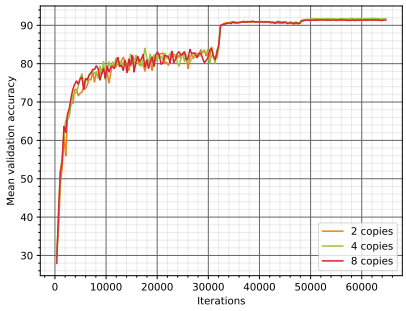
B.1 Baseline

M	Mean ± std
1	91.16 ± 0.27
2	91.31 ± 0.22
4	91.36 ± 0.23
8	91.43 ± 0.28
16	91.52 ± 0.11

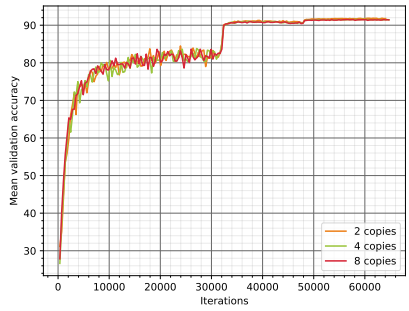
Table B.1: Top-1 validation accuracy at end of training for different number of workers

B.2 Fully synchronous

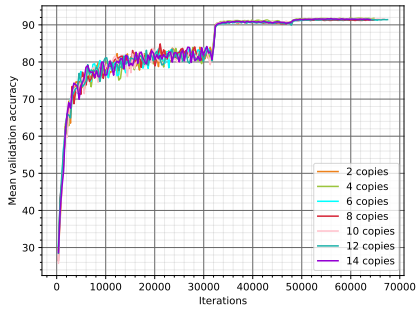
B.2.1 Varying degrees of overlap



(a) Coarse sharding



(b) Medium-coarse sharding



(c) Fine-grained sample assignment

Figure B.1: Results for different data assignment schemes. All results are run with $M = 16$

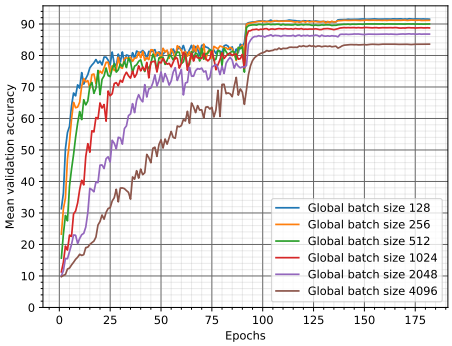
C	**Coarse**	**Medium-coarse**	**Fine-grained**
2	(91.42 ± 0.24)	(91.76 ± 0.23)	(91.40 ± 0.21)
4	(91.70 ± 0.12)	(91.45 ± 0.46)	(91.77 ± 0.31)
6			(91.26 ± 0.20)
8	(91.36 ± 0.32)	(91.41 ± 0.34)	(91.34 ± 0.15)
10			(91.52 ± 0.23)
12			(91.42 ± 0.23)
14			(91.40 ± 0.33)

Table B.2: Top-1 validation accuracy at end of training for different data assignment schemes. For each value of C we have run 5 experiments with different seeds, and report the results on format "(mean ± std)".

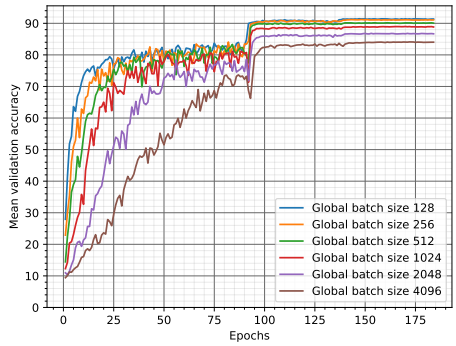
B.2.2 Fully synchronous training with large batches

B_{global}	Disjoint	C/M=0.5	Full overlap
128	(91.52 ± 0.11)	(91.36 ± 0.32)	(91.44 ± 0.20)
256	(91.16 ± 0.37)	(91.09 ± 0.55)	(91.03 ± 0.41)
512	(90.01 ± 0.44)	(90.05 ± 0.24)	(89.97 ± 0.43)
1024	(88.76 ± 0.40)	(88.84 ± 0.45)	(88.64 ± 0.16)
2048	(86.79 ± 0.57)	(86.68 ± 0.77)	(86.62 ± 0.56)
4096	(83.60 ± 0.63)	(84.03 ± 0.64)	(84.44 ± 0.71)

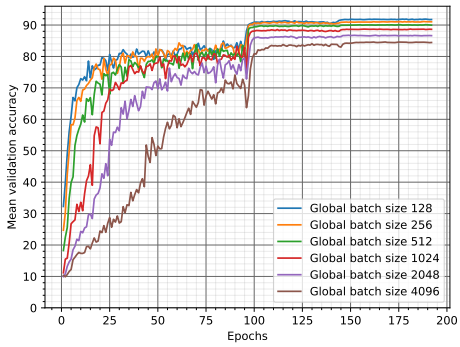
Table B.3: Top-1 validation accuracy at *end of training* for different data assignment schemes with varying global batch size. The results are presented on the format "(mean ± std)" over 5 runs



(a) Disjoint data assignment



(b) Coarse sharding with $S = 2$, which leads to $\frac{C}{M} = 0.5$

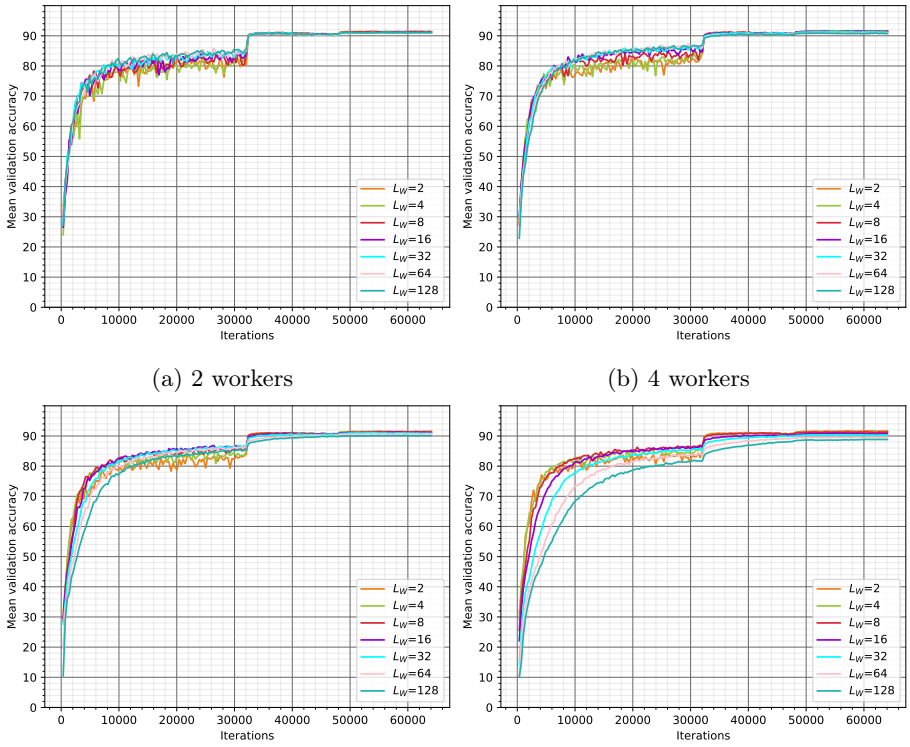


(c) Full overlap data assignment

Figure B.2: Mean validation accuracy *throughout training* for different data assignment schemes with varying global batch size

B.3 Communication reduction

B.3.1 Local SGD



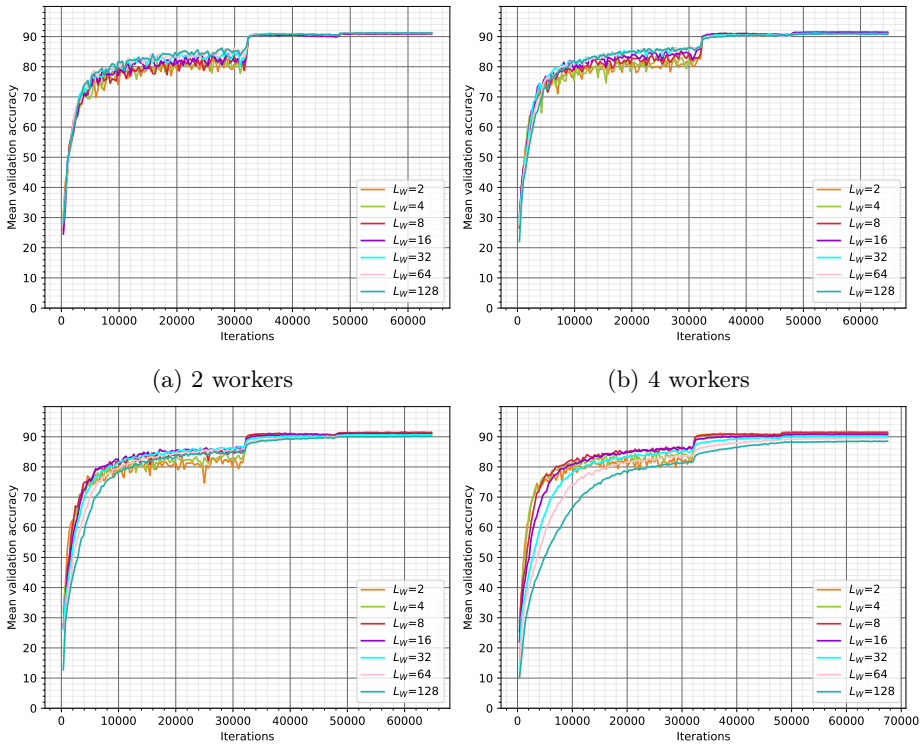
(a) 2 workers

(b) 4 workers

(c) 8 workers

(d) 16 workers

Figure B.3: Mean validation accuracy for varying number of workers when training with local SGD. Each worker is assigned a disjoint data shard.



(a) 2 workers

(b) 4 workers

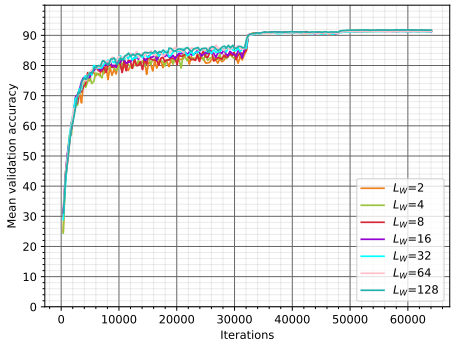
(c) 8 workers

(d) 16 workers

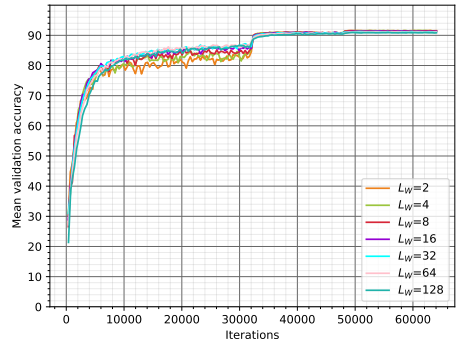
Figure B.4: Mean validation accuracy for varying number of workers training with local SGD. Every worker is assigned the entire dataset, *i.e.*, full overlap

B.3.2 Households

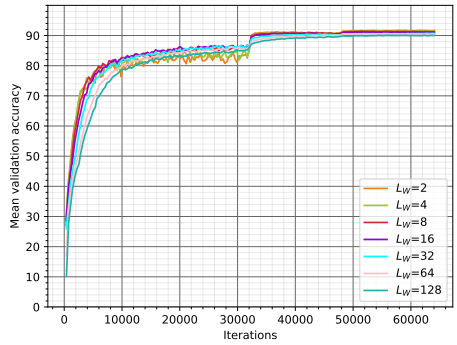
Constant local batch size



(a) 2 households

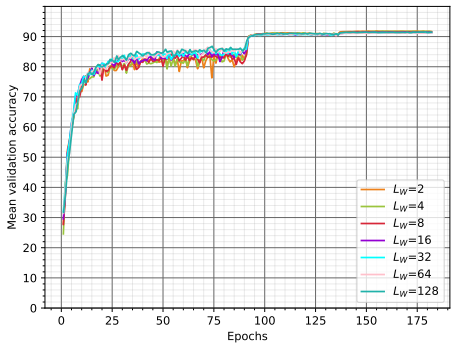


(b) 4 households

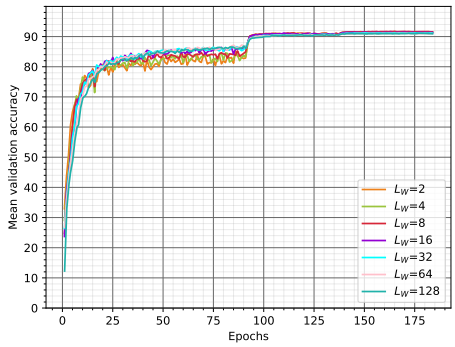


(c) 8 households

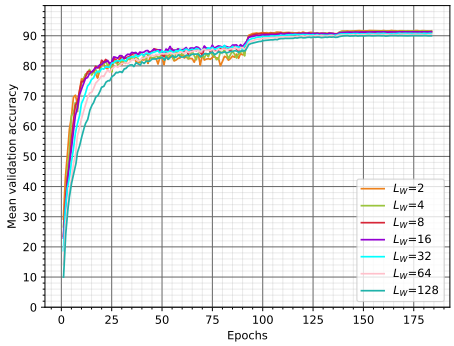
Figure B.5: Mean validation accuracy for varying number of households with different world synchronization periods. Each household has a unique household shard.



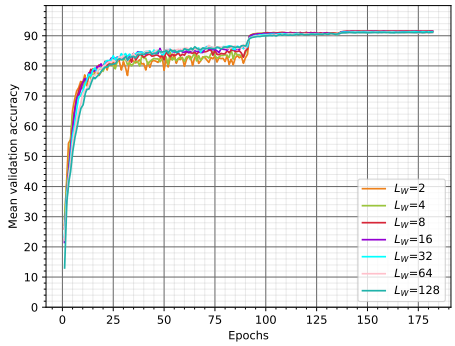
(a) 2 households with 1 household shard



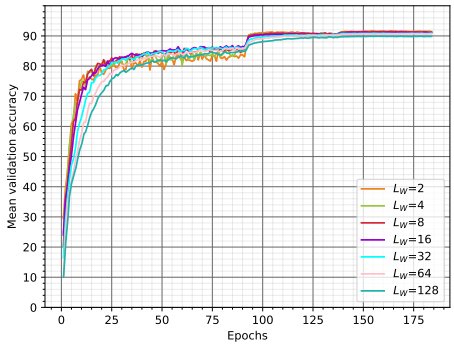
(b) 4 households with 1 household shard



(c) 8 households with 1 household shard



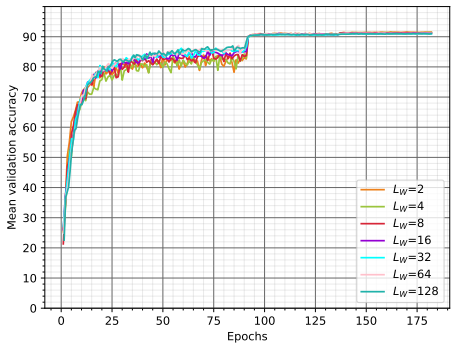
(d) 4 households with 2 household shard



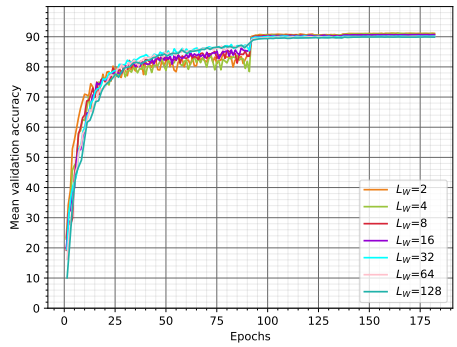
(e) 8 households with 2 household shard

Figure B.6: Mean validation accuracy for varying number of households with different world synchronization periods. Each household has a unique household shard.

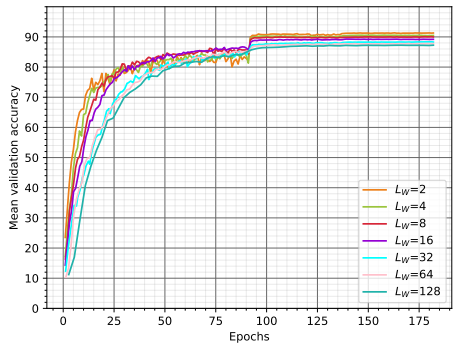
Constant effective batch size



(a) 2 households

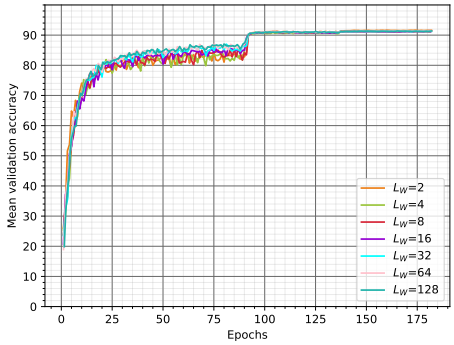


(b) 4 households

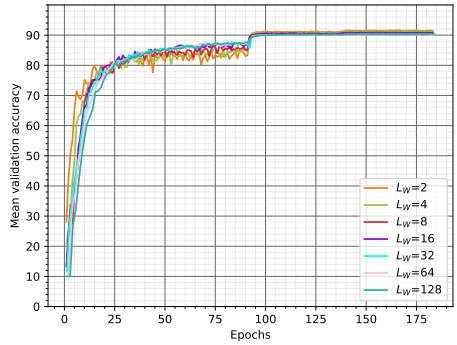


(c) 8 households

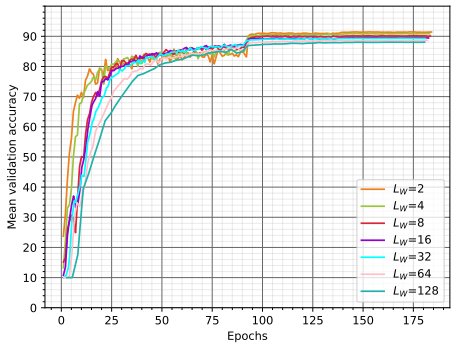
Figure B.7: Mean validation accuracy for varying number of households with different world synchronization periods. The experiments are run with target effective batch $B_{effective} = 128$ and an initial learning rate of 0.1. Each household has a unique household shard.



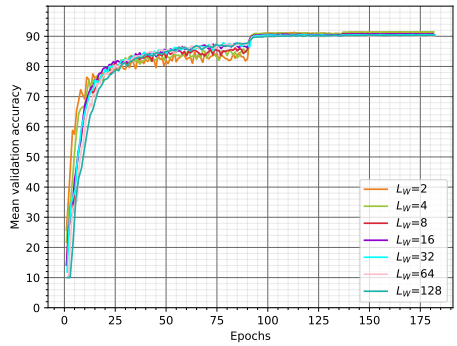
(a) 2 households with 1 household shard



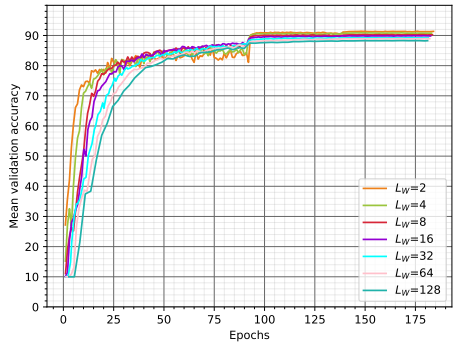
(b) 4 households with 1 household shard



(c) 8 households with 1 household shard



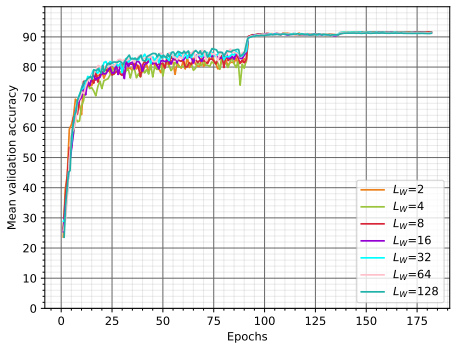
(d) 4 households with 2 household shards



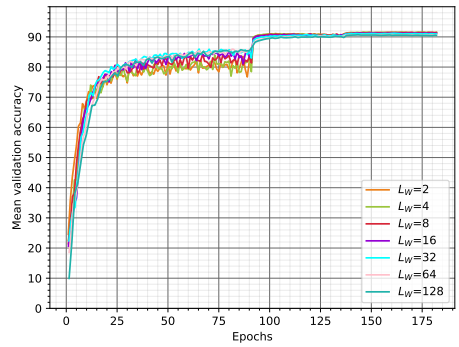
(e) 8 households with 2 household shards

Figure B.8: Mean validation accuracy for varying number of households with different world synchronization periods. The experiments are run with target effective batch $B_{effective} = 128$ and an initial learning rate of 0.1. The data is assigned with overlap between the households.

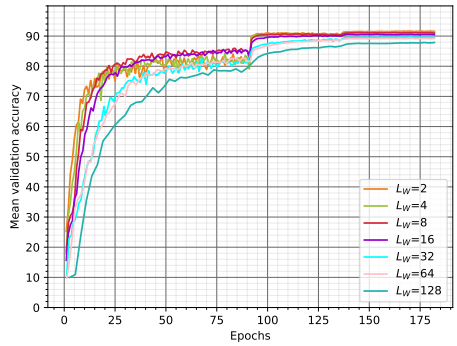
Constant effective batch size with linearly scaled learning rate



(a) 2 households

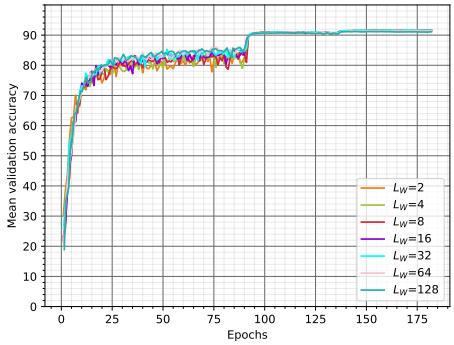


(b) 4 households

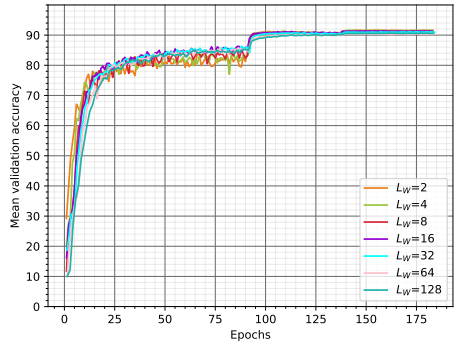


(c) 8 households

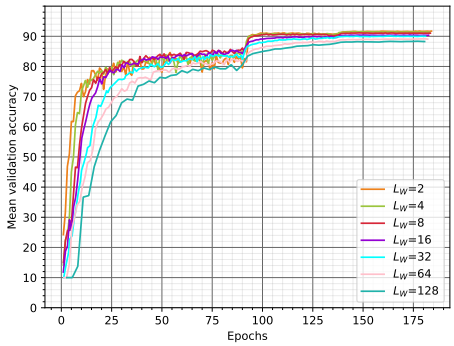
Figure B.9: Mean validation accuracy for varying number of households with different world synchronization periods where we keep a constant effective batch size of 128 and scale the learning rate linearly with the increase in local batch size. Each household has a unique household shard.



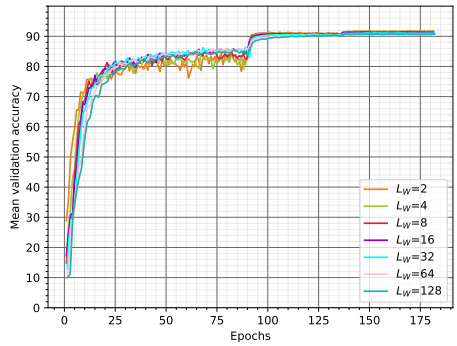
(a) 2 households with 1 household shard



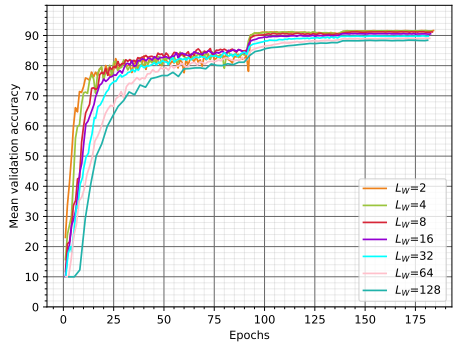
(b) 4 households with 1 household shard



(c) 8 households with 1 household shard



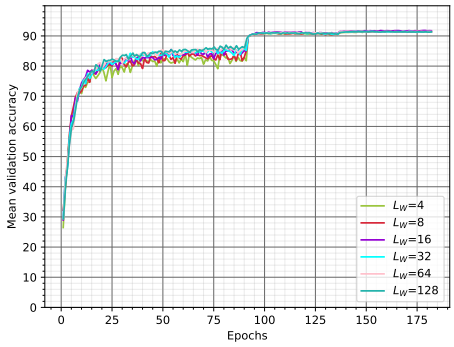
(d) 4 households with 2 household shard



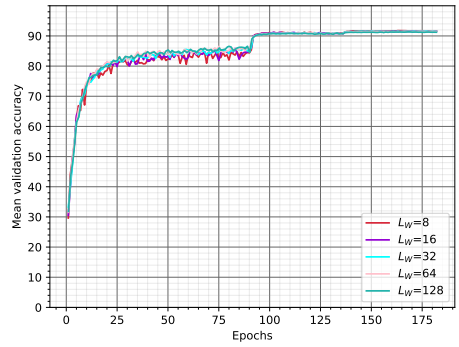
(e) 8 households with 2 household shard

Figure B.10: Mean validation accuracy for varying number of households with different world synchronization periods where we keep a constant effective batch size of 128 and scale the learning rate linearly with the increase in local batch size. The data is assigned with overlap between households.

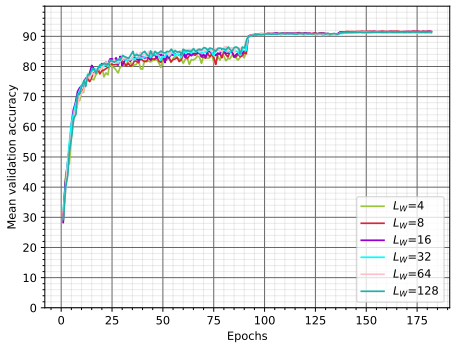
B.3.3 Neighbourhoods



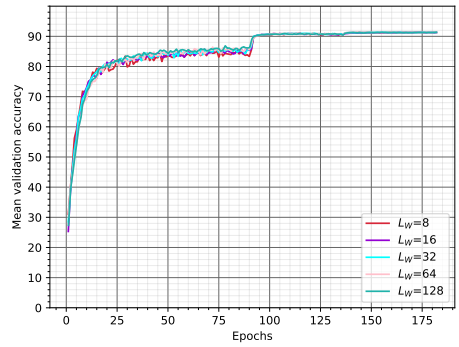
(a) 4 households with neighbourhood synchronization period 2



(b) 4 households with neighbourhood synchronization period 4

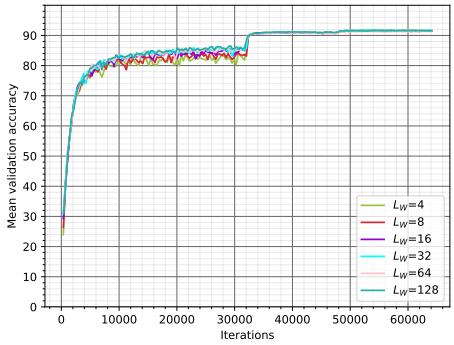


(c) 8 households with neighbourhood synchronization period 2

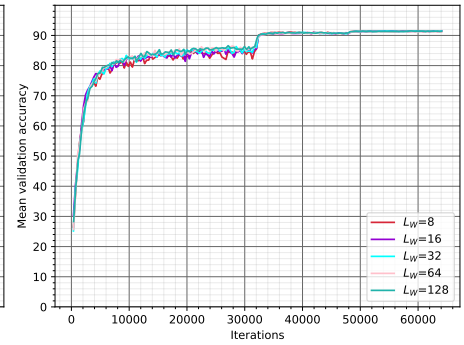


(d) 8 households with neighbourhood synchronization period 4

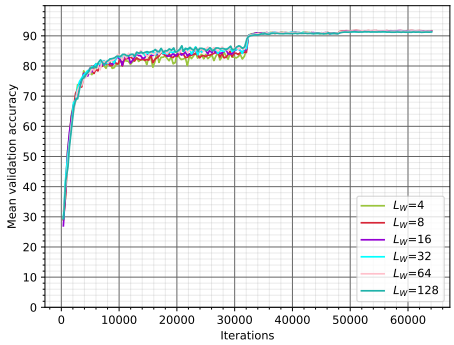
Figure B.11: Mean validation accuracy throughout training for 2 neighbourhoods where each household is given a unique household shard, and thus, there are no overlap between the neighbourhoods



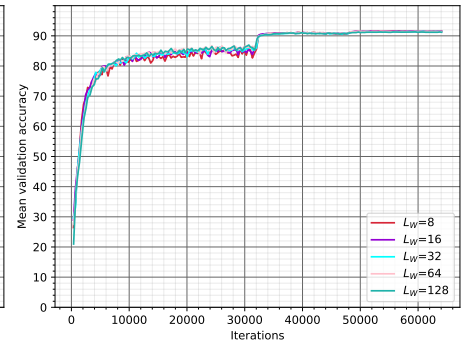
(a) 4 households with neighbourhood synchronization period 2



(b) 4 households with neighbourhood synchronization period 4



(c) 8 households with neighbourhood synchronization period 2



(d) 8 households with neighbourhood synchronization period 4

Figure B.12: Mean validation accuracy throughout training for 2 neighbourhoods with disjoint data *within* the neighbourhoods and full overlap *between* the neighbourhoods