High Performance Computing

Speeding up deep neural network training

Habib Slim
habib.slim@grenoble-inp.org

Supervisor: **Christophe Picard**

Abstract

For this project, we have implemented and tested various schemes to speed up the training of deep neural networks (DNNs), using parallel versions of Minibatch Stochastic Gradient Descent (minibatch SGD).

We first conduct a brief review of the main strategies used in the literature to speed up minibatch SGD in section 1.1.

In section 1.2, we then present our proposition for a minimal framework that we use to quickly prototype neural networks and facilitate weight and gradients sharing among processing nodes.

Finally, we introduce and implement parallel algorithms and conduct a (hopefully complete) benchmark of the different methods.

We obtain parallel versions of minibatch SGD that converge significantly faster than their sequential counterpart.

Outline

1 Methods and design **1**
1.1 Parallel and distributed DNNs: a brief review 1
1.1.1 Pipeline parallelism 3
1.1.2 Model parallelism 3
1.1.3 Data parallelism 4
1.2 Base features 6
1.2.1 General design 6
1.2.2 Implemented components 7
1.3 Algorithms 8

2 Experiments **10**
2.1 Evaluation 10
2.2 Results 11
2.2.1 Gradient sharing 11
2.2.2 Parameter averaging 12
2.2.3 Weighted parameter averaging 14
2.3 Further analysis 15

3 Conclusion **16**

4 Appendix **17**
4.1 Hardware and software used 17
4.2 Code snippets 17
4.3 Additional experiments 18

1 Methods and design

1.1 Parallel and distributed DNNs: a brief review

Various ways of speeding up the training of Deep Neural Networks (DNNs) by making use of multiple processors have been explored in the literature [1], and implemented in frameworks like PyTorch [2] or Tensorflow [3]. After providing a general context for the training of deep neural networks, we will first give a brief summary of some of the main strategies used.

Fully Connected Network. In what follows, $(\boldsymbol{x}, y) \in \mathcal{X} \times \mathcal{Y}$ denotes a labelled sample, where x is the data-point and y the associated label, and $\mathcal{X} \subseteq \mathbb{R}^d$.
We suppose that we have $(\boldsymbol{x}, y) \overset{i.i.d}{\sim} \mathcal{D}$, where $\mathcal{D}$ is an unknown probability distribution, and we note $\mathcal{S} = (\boldsymbol{x}_i, y_i)_{i \in [\![1,N]\!]}$ the training set.

For all of the three major partitioning strategies that can be used in order to speedup forward and backward passes in a neural network, we will use as illustration the following generalizable example of a fully connected neural network given in figure 1 below.

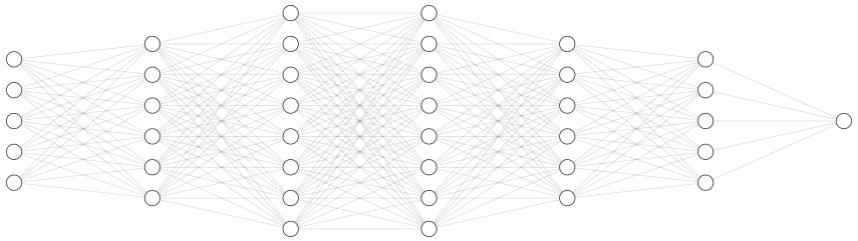

Figure 1: Example of a fully connected network with five hidden layers

In a fully connected layer, a forward pass at the level of each layer is equivalent to the simple following operation:

$$z_k = \sigma\left(\langle \boldsymbol{w}_k\,,\ \boldsymbol{x}\rangle_{\mathbb{R}^n} + b_k\right) \tag{1}$$

Where z_k is the output of neuron k in the layer, $\boldsymbol{w}_k$ and b_k are respectively the weight vector and the bias value associated to neuron k, $\boldsymbol{x}$ is the input vector and $\sigma : \mathbb{R} \mapsto \mathbb{R}$ is an activation function.

By grouping samples from the training set $\mathcal{S}$, we can form a training matrix $\boldsymbol{X} \in \mathbb{R}^{N \times d}$. Defining for each layer weight matrices $\boldsymbol{W} \in \mathbb{R}^{d \times o}$ and bias vectors $\boldsymbol{b} \in \mathbb{R}^o$ with o the number of neurons in the following layer, we can then rewrite the definition (1) in the following simple matrix form:

$$f_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{X}) = \boldsymbol{Z} = \mathfrak{S}(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}) \tag{2}$$

Where $\boldsymbol{Z} \in \mathbb{R}^{N \times o}$ is the output activation matrix, with the new activation function $\mathfrak{S}$ simply defined by applying σ on each matrix coefficient:

$$\mathfrak{S}_{i,j} : \mathbb{R} \mapsto \mathbb{R} \tag{3}$$

$$x \mapsto \sigma(x) \tag{4}$$

With $\boldsymbol{\theta}$ denoting the set of all weights and biases of the neural network, the forward function $F_{\boldsymbol{\theta}}$ is then defined as the composition of all layer functions:

$$F_{\boldsymbol{\theta}}(\boldsymbol{X}) = f_{\boldsymbol{W}_L, b_L} \circ \cdots \circ f_{\boldsymbol{W}_1, b_1}(\boldsymbol{X}) \tag{5}$$

Where L denotes the number of linear layers.

SGD and Minibatch SGD. In a supervised learning setup, the general objective is to learn optimal parameters $\boldsymbol{\theta}^*$ such that:

$$\begin{aligned}
\boldsymbol{\theta}^* &= \arg\min_{\boldsymbol{\theta} \in \Theta} \ \mathcal{L}_{\mathcal{D}}(\boldsymbol{\theta}) && (6)\\
&= \arg\min_{\boldsymbol{\theta} \in \Theta} \ \mathbb{E}_{(\boldsymbol{x}, y) \sim \mathcal{D}}[\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), y)] && (7)\\
&= \arg\min_{\boldsymbol{\theta} \in \Theta} \int_{\mathcal{X} \times \mathcal{Y}} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), y) d\mathcal{D}(x, y) && (8)
\end{aligned}$$

Where $\ell \colon \mathcal{Y} \times \mathcal{Y} \mapsto \mathbb{R}$ and $f_{\boldsymbol{\theta}}$ are respectively an arbitrary loss function and the forward function, both defined for an individual sample.
Since $\mathcal{D}$ is generally unknown, minimizing the generalization error $\mathcal{L}_{\mathcal{D}}$ can be done by estimating its gradient as follows. By linearity of the gradient operator:

$$\nabla \mathcal{L}_{\mathcal{D}}(\boldsymbol{\theta}) = \nabla \mathbb{E}_{(\boldsymbol{x}, y) \sim \mathcal{D}}[\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), y)] = \mathbb{E}_{(\boldsymbol{x}, y) \sim \mathcal{D}}[\nabla \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), y)] \tag{9}$$

Where the right-hand side can be estimated by randomly sampling from the training set $\mathcal{S}$, computing the gradient at the sampled point and updating the model parameters $\boldsymbol{\theta}$ accordingly: this is commonly known as the Stochastic Gradient Descent (SGD) algorithm (alg. 1).

Algorithm 1 Stochastic Gradient Descent (SGD)

1: **for** $t = 0$ to T **do**
2: $\quad (\boldsymbol{x}, y) \overset{\$}{\leftarrow} \mathcal{S}$ ▷ Sampling from $\mathcal{S}$ with replacement
3: $\quad \hat{y} = f_{\boldsymbol{\theta}^{(t)}}(\boldsymbol{x})$ ▷ Evaluating f with the current parameters
4: $\quad \boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta \nabla \ell(\hat{y}, y)$ ▷ Updating the parameters
5: **end for**

Where η is the learning rate. When using neural networks, the computation of the gradient in line (3) is performed at each layer using the backpropagation [4] algorithm.

Alternatively, the samples from the training set $\mathcal{S}$ can be grouped together in batches $\boldsymbol{X} \in \mathbb{R}^{B \times d}$, where B is the size of each batch, such that $B \leq N$.
The parameters are then updated after each batch instead of after each sample in SGD: this is known as the Minibatch SGD algorithm (alg. 2).

Algorithm 2 Minibatch SGD (one epoch)

1: **for** $t = 0$ to $\lfloor \frac{N}{B} \rfloor$ **do**
2: $(\boldsymbol{X}, \boldsymbol{y}) \overset{\$}{\leftarrow} \mathcal{S}$ ▷ Sampling B samples from $\mathcal{S}$ without replacement
3: $\hat{\boldsymbol{y}} = F_{\boldsymbol{\theta}^{(t)}}(\boldsymbol{X})$ ▷ Evaluating F with the current parameters
4: $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta \frac{1}{B} \sum_{i=1}^{B} \nabla \ell(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i)$ ▷ Updating the parameters
5: **end for**

Using larger minibatch sizes has the advantage of facilitating parallelism, as parameter updates occur a single time for each minibatch and minibatches can be split and distributed to nodes as subbatches.

However, some trade-offs in the generalization capacity of models occur when the batch size becomes too large, which hinders the scaling capacity of parallel models relying on minibatch partitioning [5].
Typical values for B range from 32 to 512 samples.

1.1.1 Pipeline parallelism

Pipeline parallelism is a first partitioning strategy commonly used to speedup minibatch SGD. In this method, when computing forward and backward passes each layer (or groups of adjacent layers) is assigned to different processors, as illustrated in figure 2.

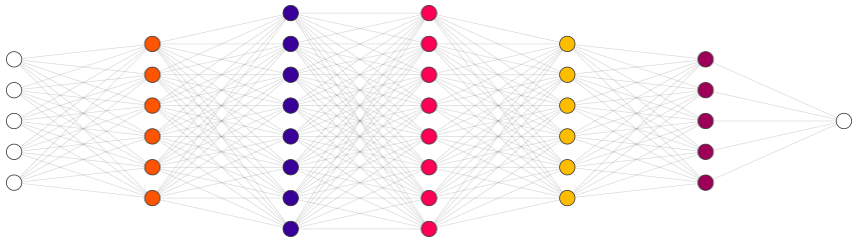

Figure 2: Pipelining a dense network (different colors are different processors)

This strategy is particularly useful when the memory footprint of all of the parameters is too big for a single processor. In cases where the number of parameters is not an issue, this strategy (although less relevant) will still give speedups over sequential training. Processors in this method only communicate to a single other processor between each layer (or partition), which is also advantageous.

1.1.2 Model parallelism

An other way to partition computation in the forward and backward passes of a DNN is to use model parallelism: the work is divided horizontally in each layer among the processors, as illustrated in figure 3.

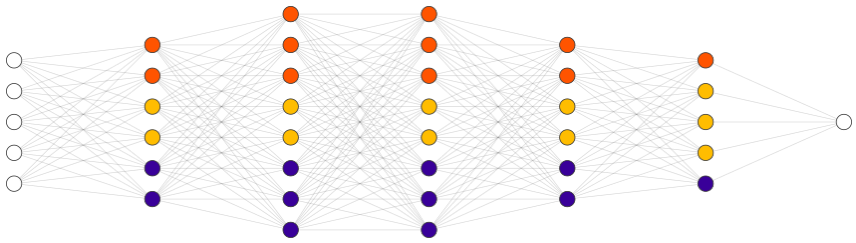

Figure 3: Model parallelism for a dense network

This strategy requires low latency between the processing nodes to be efficient, as back-propagation will require all-to-all communications at every layer. We can also note that all mini-batches have to be copied to all processors, which overall doesn't make it suitable when training on large amounts of data.
However, like for pipeline parallelism, it enables the distribution of parameters across multiple processors in cases where the number of parameters is too high. A typical example of model parallelism for DNNs could be the implementation of Cannon's distributed matrix algorithm to efficiently compute forward and backward passes [6].

1.1.3 Data parallelism

Data parallelism is the most prevalent way to obtain speedups when training deep neural networks, both in the literature and in framework implementations.

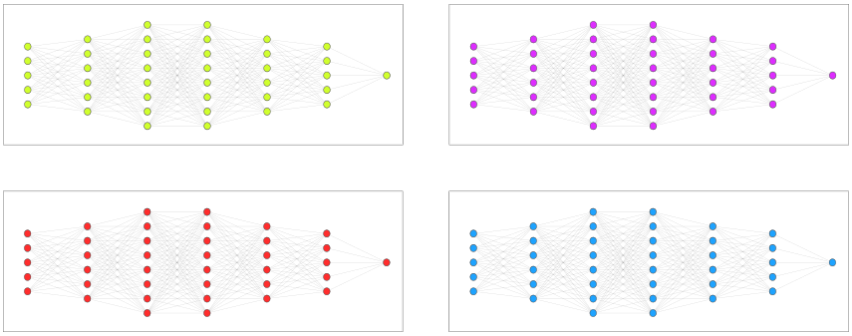

Figure 4: Data parallelism for a dense network

Up until this point, all of the approaches considered (in 1.1.1, 1.1.2) are exact parallel versions of minibatch SGD: that is, with the same initial random seed, they will converge in the same number of iterations and learn the exact same parameter values. Furthermore, only a single copy of the learned parameters $\boldsymbol{\theta}$ exist at any timepoint.

We now consider various parallelizations of minibatch SGD that are either approximations or alternatives of their sequential counterpart. This implies that additionally to speedup, performance metrics like model accuracy will have to be considered to fairly compare algorithms.

In data parallelism, each processor possesses its own instance of the full deep network, where parameters are either instance-specific or shared across all nodes. A consistent scheme is one in which every updated instance of the model parameters $\boldsymbol{\theta}$ is identical

(which can be in practice enforced using a parameter server), and a model is considered more or less consistent depending on the frequency of the parameter updates.

Below are some directions that can be taken to implement data-parallel DNN-training algorithms:

- **Gradient-sharing.** Building on the minibatch SGD algorithm, and in particular on its minibatched variant, an intuitive way to parallelize the training of DNNs is to split minibatches into subbatches. Forward/backward passes can then be computed in parallel through instances of the DNN, and output gradients can be averaged by aggregation.

- **Weight-sharing.** Instead of sharing gradients, weights can be shared and averaged among model instances, which can approximate the effect of multiple sequential updates. The frequency at which weights are averaged and shared has a direct positive impact on accuracy, but a potentially negative one on performance if communication is a constraint.

- **Asynchronous/synchronous.** In the case of a centralized parameter server, updates can be performed in a synchronous manner using the equivalent of lock-/unlock primitives on the central weights. However, this may greatly hinder the speed of the algorithm as the number of nodes grows: to solve this issue, various algorithms relying on asynchronous update schemes have been proposed (e.g. HOGWILD [7]).

- **Ensemble learning.** Instead of learning a single final set of parameters, multiple instances of the model can be trained completely independently. Aggregating weights would then have negative effects on accuracy due to model inconsistency[1]. However, if storing multiple parameter instances is viable at evaluation time, predictions can be pooled together to boost accuracy (in the case of a classifier) into a network ensemble.

- **Knowledge distillation.** If holding an ensemble of parameters is not a viable solution, knowledge distillation can be employed to transfer knowledge from the learned network ensemble (in general, called the teacher network) to a student network. In practice, this can be done by training the student network on an auxiliary prediction task which consists in predicting the softmax outputs of the teacher network.

[1] As seen later in section 2.2.2

1.2 Base features

In order to easily benchmark various parallelized algorithms of DNN training, we first implemented a minimal C++ deep learning framework. We used OpenMPI [8] to implement our distributed algorithms, and Eigen [9] for efficient matrix operations.
We provide alongside our code a README file with instructions on how to compile and re-run experiments.
We will give here a brief summary of the features implemented and of the structure of the proposed software.

1.2.1 General design

In figure 5 below is a diagram for the simple modular architecture that we propose for this project.

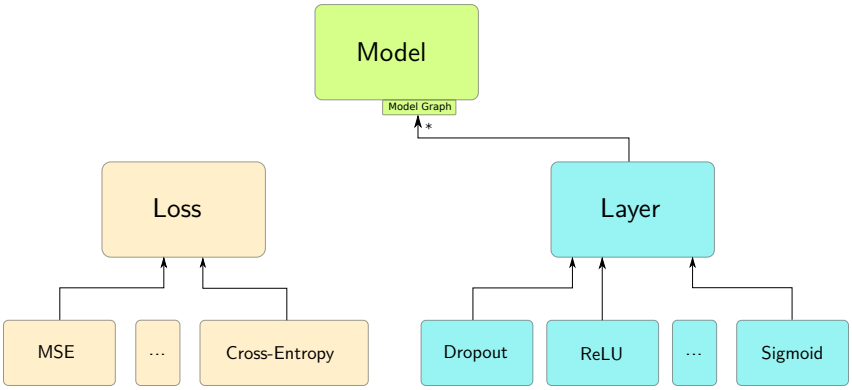

Figure 5: Diagram of the software solution proposed

Classes inheriting from the Layer class must specify their own `forward`, `backward` and `serialize` functions. When creating a new model, the component's directed acyclic graph (DAG) is updated automatically. The DAG will then be used directly when computing forward and backward passes: there is thus no need for explicitly defining the forward path for each new model created.

Models can then be created by simply compositing layers, as illustrated in the working code example given in figure 16 of appendix 4.2.
This design enabled the quick prototyping of models used to build experiments, and more generally to write readable and drastically less redundant code.

Models can be serialized using the `serialize` primitive, and loaded back with the `load` primitive. When serializing, the DAG of the model is searched for layers which contain shareable parameters[2], and the raw Eigen buffers are packed together in a vector. These buffers can then be directly exchanged via MPI primitives.

[2]Examples of non-parametric layers include activation layers like ReLU and TanH. Dropout layers are also ignored since the random regularization masks are not relevant.

To facilitate the implementation of algorithms involving the sharing of gradients, we also add gradient serializing functions to layers, which enable both to share gradients across nodes and to update layer parameters from received serialized gradients.

1.2.2 Implemented components

We briefly present here the main neural network blocks implemented to build our models.

Layers. We implement forward and backward functions for dense (fully connected) layers, and use Xavier initialization [10] to initialize our weights.
We implement Dropout [11] regularization layers. In particular, we implement Gaussian dropout, which adds a perturbative Gaussian noise to output activations.

We also implement the following activation layers:

- ReLu and Leaky ReLu
- Sigmoid
- Hyperbolic tangent
- Softmax

Losses. We implement the following loss functions:

- Mean squared error loss (MSE)
- Cross-entropy
- Cross-entropy with logits losses

Cross-entropy with logits is a computational trick combining cross-entropy loss with the softmax activation in a single layer, which allows it to directly operate on the unscaled outputs of a neural network. This allows to significantly speed up the training of networks using both the cross-entropy loss and softmax output activations.

1.3 Algorithms

We present here various approaches tested to implement parallel minibatch SGD algorithms. In our experimental setup (4.1), communication is not a big constraint. We have thus focused mainly on facilitating computation when selecting and comparing parallel solutions to DNN training.

Gradient sharing. In figure 6 below, we give a diagram describing the gradient sharing algorithm proposed.

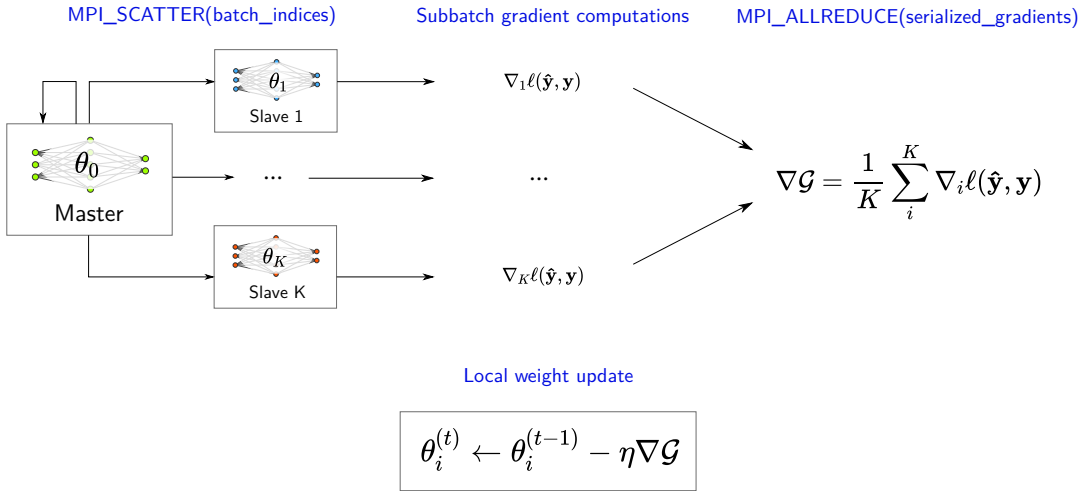

Figure 6: Distributed gradient averaging (for a single minibatch). The master node scatters each subbatch across all processors, all nodes (including the master node) compute forward passes on their subbatches and evaluate gradients without updating their parameters. The gradient is then all-reduced, and the parameters are updated locally for every model instance.

Following on the idea of splitting batches into minibatches, we compute gradients in parallel through (local) backpropagation without directly updating the layer weights. Gradients are then aggregated and averaged, and each model locally performs a weight update using the averaged gradient $\nabla \mathcal{G}$.

This method thus implies synchronization mechanisms at the end of each minibatch. In order to best take advantage of using multiple processing nodes, the batch size will have to be increased so that the work share for each node is large enough.

Parameter averaging. Instead of synchronizing processors every minibatch, we consider a first parallel variant of minibatch SGD which allows us to compute multiple parallel forward passes on different minibatches. In order to ensure some level of consistency, model weights are averaged across nodes at most a single time at the end of each epoch. A diagram for this algorithm is given in figure 7 below.

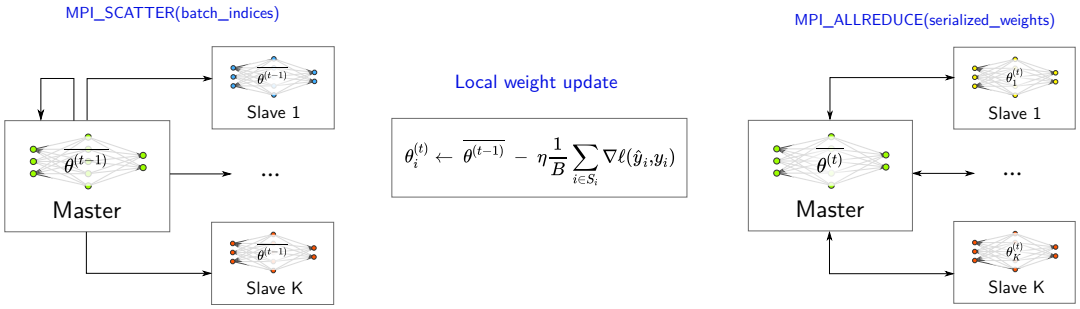

Figure 7: Distributed parameter averaging (for epochs of one batch). The master node scatters batch indices across all processors, all nodes (including the master node) compute forward/backward passes on their minibatches and locally update their parameters, then parameters are all-reduced at the end of the epoch. In the last epoch, the `ALL_REDUCE` operation is replaced by a `REDUCE`.

In our implementation, the master node[3] also trains its own instance of the model on partitions of the full dataset. We consider variants of this algorithm, in which we:

- Reduce $\boldsymbol{\theta}_i^{(t)}$ only a single time at the end of all training epochs
- All-reduce $\boldsymbol{\theta}_i^{(t)}$ every k epochs, where k is a tunable hyperparameter

In 2, we measure the differences between the two approaches and the effects of varying the update frequency k.

Weighted parameter averaging. One of the disadvantages of the proposed parameter averaging algorithm is the frequency of all-reduce operations required to maintain model consistency.
A possible solution to this problem is to only all-reduce weights if individual $\boldsymbol{\theta}_i$ values are diverging too much from the previously averaged $\bar{\boldsymbol{\theta}}$.
We can also all-reduce $\boldsymbol{\theta}_i^{(t)}$ at a fixed rate k, where each parameter vector is weighted by a parameter γ_i, defined as:

$$\gamma_i = \exp\left(-\lambda \cdot \frac{\left\|\overline{\boldsymbol{\theta}^{(t-1)}} - \boldsymbol{\theta}_i^{(t)}\right\|_2}{\left\|\overline{\boldsymbol{\theta}^{(t-1)}}\right\|_2 + \left\|\boldsymbol{\theta}_i^{(t)}\right\|_2}\right) \tag{10}$$

With λ being an adjustable hyperparameter. The master node then performs a weighted average of parameter values instead of a flat average as in the previous method, using an additional `ALL_REDUCE` operation on the computed weights.
We thus aim to penalize model instances in which weights drift too much from the previous average, in an attempt to improve model consistency[4].

[3] In this context, it is also referred to as the "parameter server" in the literature.
[4] This is similar to the intuition behind the Elastic Averaging SGD [12] algorithm.

2 Experiments

2.1 Evaluation

Datasets. We experiment with training classifiers on the MNIST and F-MNIST [13] image datasets. Fashion-MNIST was introduced as a more challenging alternative for MNIST, and we use it in section 2 as a secondary benchmark for the convergence speed of implemented algorithms. In both datasets, the number of classes is equal to ten.

Models. In figure 8 below, we give the architecture of the MLP we train on both datasets.

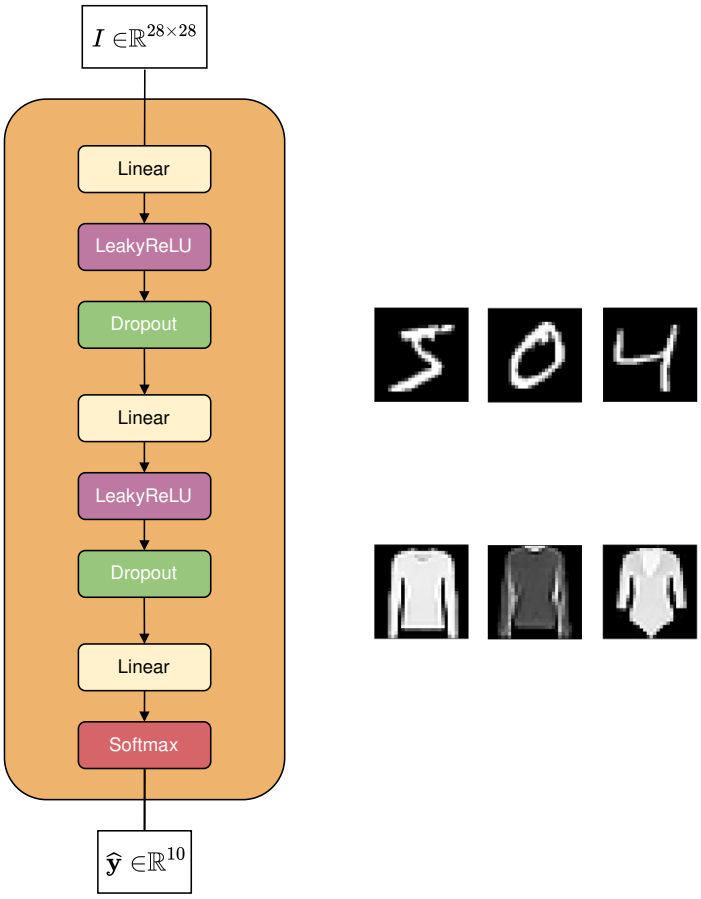

Figure 8: MLP proposed as a benchmark model, MNIST (top) and F-MNIST (bottom) samples.

We use dropout layers with $p = 0.2$, LeakyReLU activations with $\alpha = 0.3$, $\eta = 0.01$ as a fixed learning rate, and the MSE loss. Regarding layer sizes, we experiment with dimensions 256×64 for the hidden layer (referred to as "model A" later in the report), the outer layers being conditioned on input and output dimensions.

Hardware. All of the hardware and software details of our testing environment are given in appendix 4.1.

Baseline. Accuracy and speed measurements for the sequential baseline (evaluated using the same testing environment) are given in appendix 4.3.

2.2 Results

2.2.1 Gradient sharing

Accuracy and B. We plot validation accuracies of the master node (see fig. 6) for various number of cores and batch sizes and compare it to the sequential baseline, in figure 9 below.

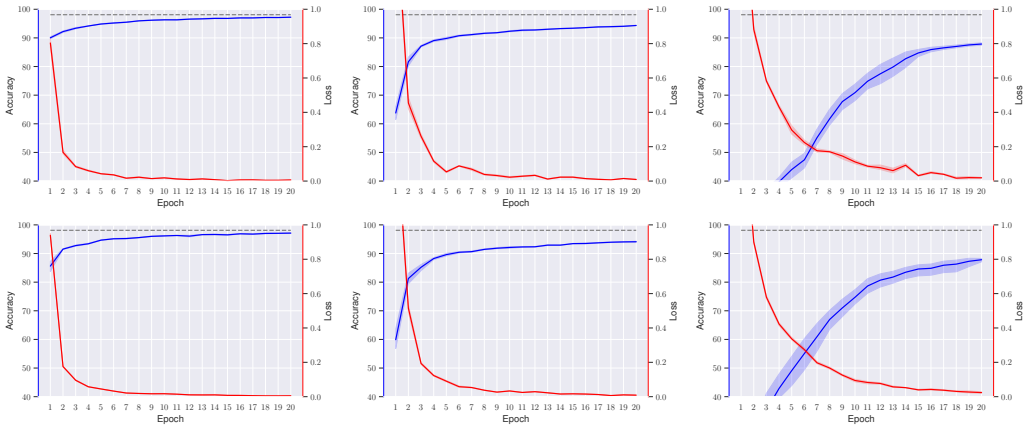

Figure 9: Accuracy and average loss values of model A trained with the gradient averaging algorithm on MNIST. Using varying batch sizes (top: $B = 64$, bottom: $B = 512$), and number of processors ($N = 2, 4, 8$ from left to right). In dashed grey is the sequential baseline for the same batch size. Margins of error evaluated across $T = 5$ runs, for training sessions of 20 epochs.

Besides runs with $N = 2$ which gave final accuracies remarkably close to the sequential baseline after 20 epochs[5], we can see that further increasing the number of processors leads to a deteriorated final validation accuracy.

Speedups. As seen in table 1, the speedup figures computed are highly misleading considering the significant losses in accuracy compared to the sequential version - although expected since the size of each batch matrix is effectively divided by the number of processors which speeds up matrix products considerably. The comparison with the sequential baseline thus loses of its relevance.

In section 2.3, we provide a more thoughtful comparison between this algorithm and sequential minibatch SGD.

	N=2	**N=4**	**N=8**
Speedup	3.43	9.87	13.66

Table 1: Average speedups per epoch and efficiencies as a function of the number of processors for $B = 512$ (bottom).
Evaluated across at least $T = 5$ runs, for training sessions of 20 epochs.

[5]Please refer to appendix 4.3 for the sequential baseline measurements.

2.2.2 Parameter averaging

Speedups. In table 2 below, we give the average speedup per epoch and efficiencies using the parameter averaging algorithm, where weights are aggregated every epoch.

	N=2	**N=3**	**N=4**	**N=6**	**N=8**
Speedup	1.85	2.77	3.66	3.2	**4.20**
Efficiency	0.92	**0.92**	0.91	0.53	0.52

Table 2: Average speedups per epoch and efficiencies as a function of the number of processors. Evaluated across at least $T = 5$ runs, for training sessions of 20 epochs ($B = 64$, $k = 1$).

Overall, we obtain slightly sub-linear speedups before $N = 6$, and reach a maximum speedup of $4\times$ with $N = 8$ processors. We observe a significant loss of efficiency after $N = 4$ processors[6].

Accuracy and k. We plot validation accuracies of the master node model (see 7) per epoch and vary the averaging frequency k. Results are given in figure 10 below.

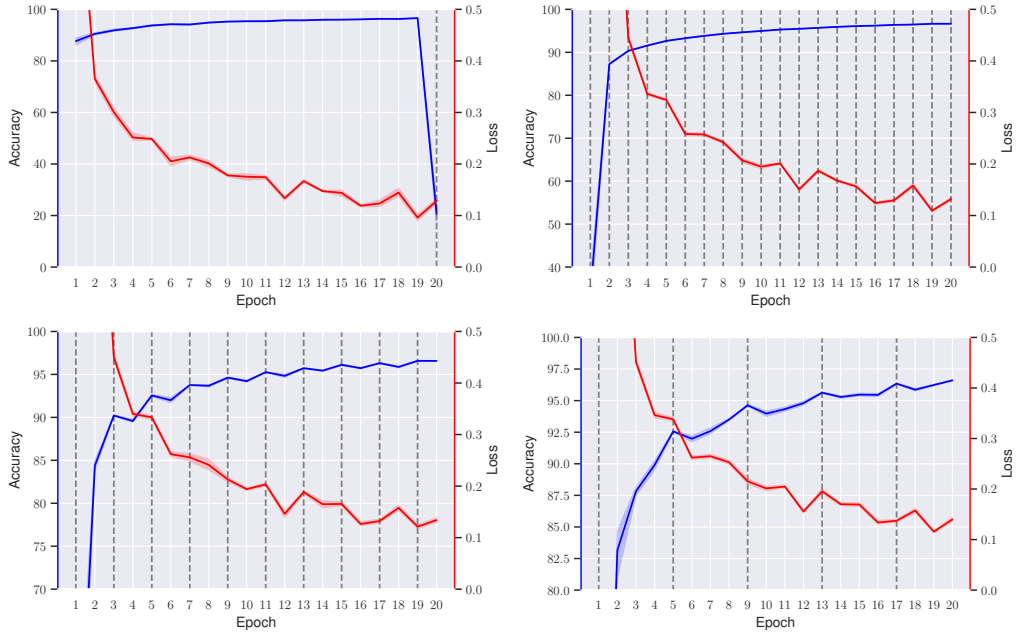

Figure 10: Accuracy and average loss values of model A trained with parameter averaging on MNIST. In dashed grey are weight averaging epochs. Margins of error evaluated across $T = 5$ runs, for training sessions of 20 epochs ($N = 8, B = 64$).

In the case for which we aggregate weights only at the end of training (top left), we see that models clearly exhibit inconsistency which explains the sharp drop in validation accuracy of the final model after averaging. In other cases, weight averaging steps clearly

[6] After additional measurements, we find that this efficiency drop is not due to a communication bottleneck (which take $\approx 0.1\%$ of the epoch time with $B = 64$). This may be due to additional constraints of our compute environment (please refer to 4.1 for additional details).

correlate with increases in validation accuracy throughout training.

Accuracy and N. In figure 11 below, we plot the validation accuracy of the final averaged models as a function of the number of cores, compared to the sequential baseline.

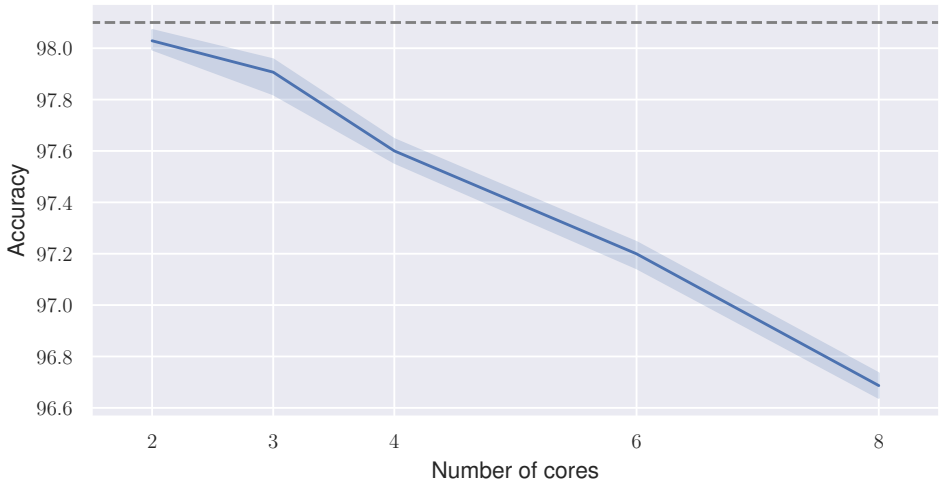

Figure 11: Accuracy after 20 epochs as a function of the number of cores. In dashed grey is the maximum validation accuracy obtained with the sequential baseline with the same parameters. Margins of error evaluated across $T = 5$ runs ($B = 64$, $k = 1$).

We can see that while using two processors gives accuracy figures comparable to those obtained with the sequential baseline, further increasing N leads to increased inconsistency and hinders convergence speed.

Furthermore, as seen in table 3 below, modifying the batch size does not yield to a more stable training for high numbers of processors.

B=64	**B=128**	**B=256**
96.68 ± 0.05	96.67 ± 0.04	96.29 ± 0.08

Table 3: Average validation accuracy after 20 epochs as a function of batch size. Margins of error evaluated across $T = 5$ runs ($N = 8$, $k = 1$).

The final accuracy metric decreases consistently as a function of the number of cores, with a maximum of $\sim 1.5\%$ additional error with $N = 8$ processors.

2.2.3 Weighted parameter averaging

Effects of γ. In figure 12 below, we compute the values of the γ_i coefficient in 40 epochs of training for the master node:

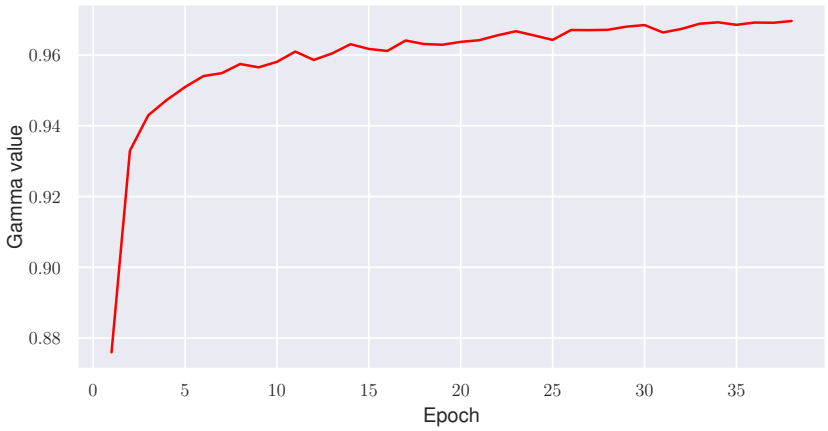

Figure 12: Gamma value in a single node, for 40 epochs of training ($B = 64$, $k = 1$, $\lambda = 1$).

As expected, the first epochs yield the lowest values for γ_i because of high divergences from the random initial weight values, but asymptotically approaches 1 in the subsequent epochs indicating a lower change rate.

Accuracy. In tables 4 below, we give accuracy figures obtained with $\lambda = 1$, $\lambda = 10$:

	N=2	**N=3**	**N=4**	**N=5**	**N=6**	**N=7**	**N=8**
Accuracy	97.97 ± 0.04	97.92 ± 0.04	97.62 ± 0.07	97.39 ± 0.05	97.19 ± 0.03	96.82 ± 0.07	96.70 ± 0.06
	N=2	**N=3**	**N=4**	**N=5**	**N=6**	**N=7**	**N=8**
Accuracy	98.05 ± 0.08	97.89 ± 0.05	97.63 ± 0.03	97.40 ± 0.06	97.20 ± 0.06	96.84 ± 0.06	96.58 ± 0.03

Table 4: Average validation accuracy after 20 epochs as a function of number of cores. Margins of error evaluated across $T = 5$ runs ($B = 64$, top: $\lambda = 1$, bottom: $\lambda = 10$).

Overall, this variant does not seem to solve the accuracy drop problem observed in parameter averaging with a high number of cores. We also experiment with modifying the learning rate instead of weighing the weights differently, which yields similar results.

Speedup. We observe no significant change in speedup with this method compared to parameter averaging, which is an expected result considering that the computational difference between the two methods is a single additional `ALL_REDUCE` on a scalar value.

2.3 Further analysis

Accuracy-time plot. In order to properly evaluate the parallel algorithms implemented, we plot the accuracy on MNIST, training model A as a function of time with all algorithms implemented and compare it to the sequential baseline.

The result is given in figure 13 below.

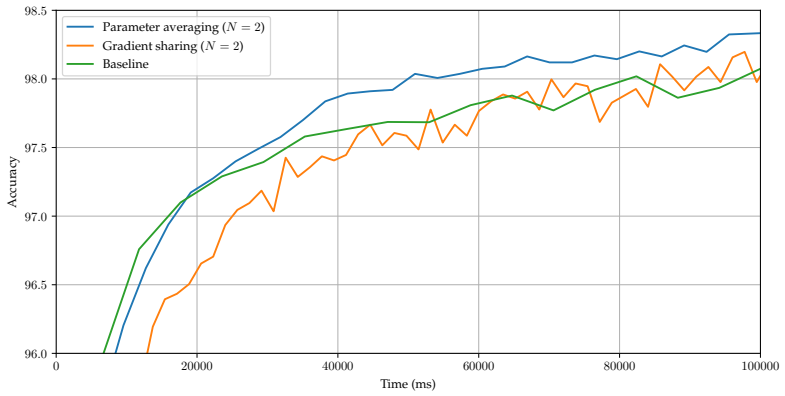

Figure 13: Validation accuracy as a function of time for model A trained with parameter averaging and gradient sharing on MNIST. Evaluated across $T = 5$ runs, for training sessions of up to 40 epochs ($N = 2$, $B = 64$).

We can see that while gradient sharing does not lead to a stable learning curve in practice, parameter averaging converges significantly faster than the baseline in the first minutes of training, although the model is already very close to its maximum accuracy on the dataset[7].

In figure 14 below, we plot the same graph when training model A on F-MNIST which highlights even more clearly the faster convergence rate of parameter averaging.

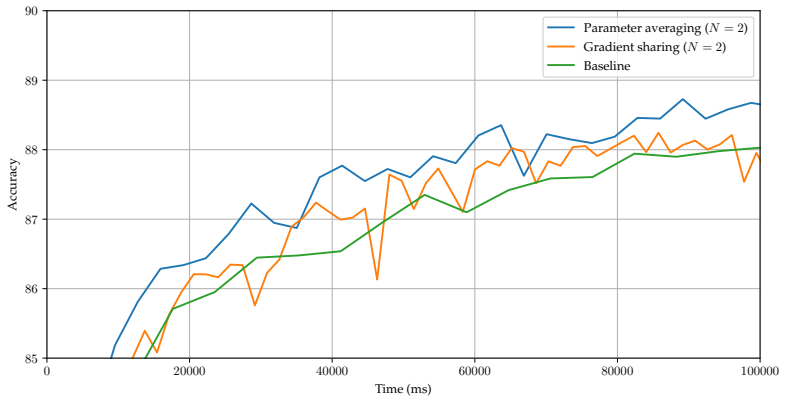

Figure 14: Validation accuracy as a function of time for model A trained with parameter averaging on F-MNIST. Evaluated across $T = 5$ runs, for training sessions of up to 40 epochs ($N = 2$, $B = 64$).

[7] Evaluated at around $\sim 98.5\%$ for model A on MNIST.

Finally, we increase the dimensions of the hidden layer of model A so that it now has 1024×512 coefficients, and increase the number of cores to 4. Results of this experiment are provided in figure 15.

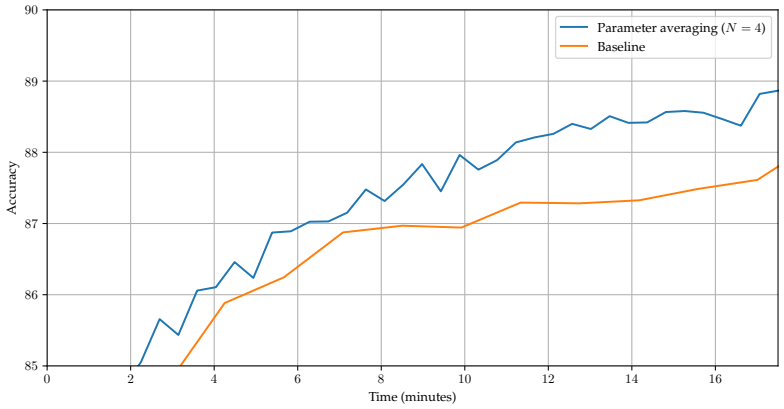

Figure 15: Validation accuracy as a function of time for model A with larger hidden layer, trained with parameter averaging on F-MNIST. Evaluated across $T = 5$ runs, for training sessions of up to 40 epochs ($N = 4$, $B = 64$).

After less than 20 minutes of training, the model trained with parameter averaging is up a full percent in validation accuracy over the sequential baseline.

3 Conclusion

In this project, we have described and implemented multiple algorithms enabling the training of deep neural networks in the context of a distributed CPU environment. We proposed a simple framework to implement and test neural networks, and conducted an experimental analysis of the performance of each method.

We have shown that parameter averaging is a robust method to train deep neural networks in a parallel fashion, and that it converges significantly faster than its sequential counterpart.

Possible improvements to the implementations could include more subtle update rules and learning rate schedules, regularization methods improving model consistency and fine-tuning of the learning rates and batch sizes, so that the increase in the number of processors does not result in a systematic drop in validation accuracy.

4 Appendix

4.1 Hardware and software used

All of our experiments have been computed on a Google Cloud cluster of virtual CPUs, with the following specification: Intel Xeon CPU @2.20 Ghz, with 1 GB per vCPU.

Google Cloud vCPUs are single hardware hyper-threads.

4.2 Code snippets

```
/* Defining a network as an anonymous class */
class : public Model {
    void define() {
        add(new Linear(256, 128, 0.01));
        add(new ReLU());
        add(new Dropout(0.2));

        add(new Linear(64, 10, 0.01));
        add(new Softmax());
    }
} net;

MSELoss loss;

/* Forward pass */
y_pred = net(data);

/* Computing loss */
loss.error(y_pred, y_real);

/* Backpropagation */
net.backward(loss.grad());
```

Figure 16: Creating a perceptron with two layers and a softmax output, and computing one forward/backward pass

4.3 Additional experiments

Sequential baseline. Baseline accuracies for the sequential minibatch-SGD algorithm are given below in figure 17 for various batch sizes.

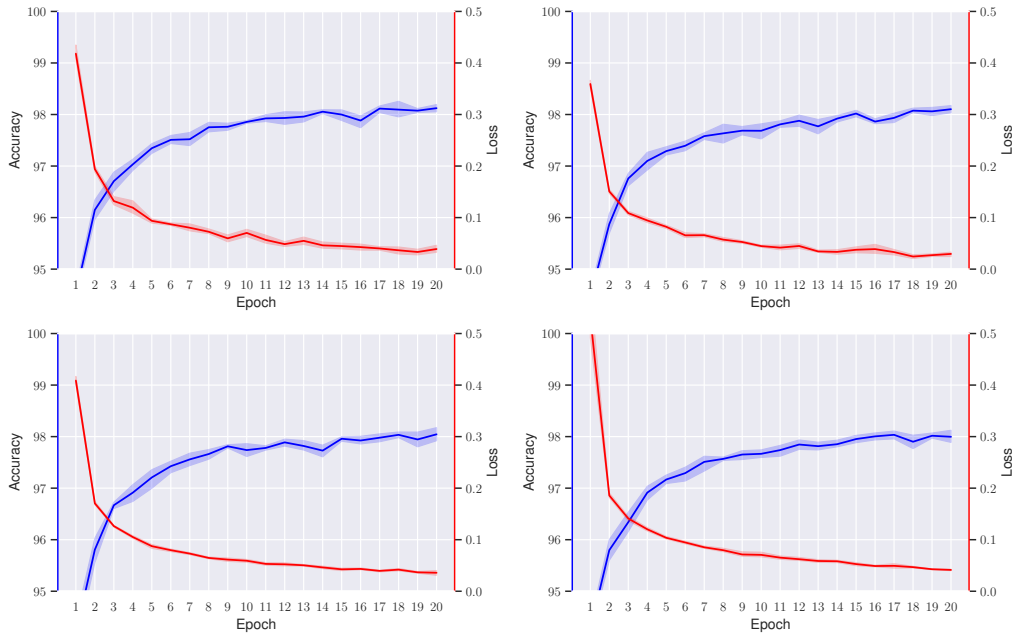

Figure 17: Baseline accuracy and average loss values of model A trained with sequential minibatch-SGD algorithm on MNIST, with $B = (32, 64, 128, 256)$ from left to right, top to bottom. Margins of error evaluated across $T = 5$ runs, for training sessions of 20 epochs.

We do not observe significant variations in the final validation accuracies obtained when modifying the batch size. In table 5 below, the average training time per epoch is given.

B=32	B=64	B=128	B=256
6281.3 ± 12.3	5884.8 ± 9.4	5674.0 ± 8.1	5540.0 ± 21.4

Table 5: Average training time per epoch as a function of batch size (ms). Margins of error evaluated across $T = 5$ runs.

As expected, we observe a significant reduction of the average training time per epoch as the batch size increases, which no significant trade-off in accuracy in the first 20 epochs of training. Gradient averaging overall does not converge to an acceptable solution fast enough.

References

[1] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *arXiv*, 2018.

[2] Adam Paszke et al. Pytorch: An imperative style, high-performance deep learning library, 2019.

[3] Martin Abadi et al. Tensorflow: A system for large-scale machine learning, 2016.

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

[5] Nitish Shirish Keskar et al. On large-batch training for deep learning: Generalization gap and sharp minima, 2017.

[6] Ludvig Ericson and Rendani Mbuvha. On the performance of network parallel training in artificial neural networks. *arXiv*, 01 2017.

[7] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *NeurIPS*, 2011.

[8] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: A flexible high performance MPI, 2006.

[9] Gaël Guennebaud, Benoıt Jacob, Philip Avery, Abraham Bachrach, Sebastien Barthelemy, et al. Eigen v3, 2010.

[10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research*, 2010.

[11] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 2014.

[12] Sixin Zhang, Anna Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. *NeurIPS*, 2015.

[13] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, 2017.