Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

January 31, 2022

Implementation of a hybrid data parallel algorithm for deep neural network training with reduced communication targeted to GPU-based supercomputers

Stefano Gonçalves Simao

Abstract

Machine Learning and Parallel Programming are a significant part of modern Computer Science. The former with applications such as image recognition, medical diagnosis, self-driving cars and many others. The latter is a consequence of trying to compensate for the fact that Moore's Law is not valid anymore and the constant increase in the usage of multicore processors. The use of increasingly fast hardware can provide an enormous impact on the performance of the training algorithms by adapting the algorithms to exploit parallelism and execute faster. The goal of this project is to investigate existing Parallel Programming strategies to distribute the work of Machine Learning algorithms for training Deep Neural Networks and propose a novel algorithm that reduces communication complexity. This algorithm is a hybrid parallel stochastic gradient descent (SGD) method that significantly reduces the communication between the nodes while improving the generalization performance of the standard parallel SGD method. This hybrid approach exploits both NVIDIA Collective Communication Library (NCCL) and Graphics Processing Units (GPUs), to speed up the evaluation of the loss function and its derivatives.

Advisor
Prof. Rolf Krause
Assistant
Dr. Alena Kopaničáková

Advisor's approval (Prof. Rolf Krause): Date:

Contents

1 Introduction

In today's world, *machine learning* applications are becoming more and more ubiquitous. In this context, Deep Neural Networks are taking the spot as one of the most impactful technologies of the last forty years [6]. Some of the applications that have received the greatest benefit from the use of DNNs include image [23] and speech recognition [2] language translation [54], autonomous driving [14], content recommendation [11], medical diagnosis [8], image upscaling [53] and many others.

Another important aspect of this field is the growing economic impact that it has in the industry and on the world entirely. In our digital society, *data* is becoming increasingly important and has become the world's most valuable resource [48]. Data can be considered the new oil, as you can extract an enormous value from it. To make it valuable, it is necessary to extract all the possible *information* from the available data. Once information is extracted, it is possible to process it in order to get the most important product: *knowledge*. This is what yields the value that can be extracted from data. Machine learning plays a primary role in this *value chain*.

Inspired by the brain, artificial neural networks provide answers to very complex problems by processing huge amounts of data. In the last years, with the rising of multicore processors and subsequently of parallel programming, this field has seen an exponential increase in popularity [6]. This goes hand in hand with the increase in the problem sizes. While hardware manufacturers push the boundary of what graphics processing units (GPUs) can do, there is a need for new models, algorithms, and frameworks to take advantage of these new hardware innovations.

When we are training very large and complex networks with huge datasets, as in the case of Tesla in the context of self-driving cars [47], it is crucial to take into consideration the computational work and memory needed to perform certain tasks and achieve the desired performance. There are multiple ways to exploit the available hardware, for example by splitting the work or the data between the computing nodes, consisting of hardware elements with computational power (e.g. CPUs, GPUs, etc). This is where the concepts of parallel programming help us to design and develop concurrent algorithms.

When designing concurrent algorithms, we have to take into account the communication between the computing nodes. The goal of a distributed algorithm is to be able to exploit the parallelization possibilities given by the hardware of a given node or cluster of nodes while trying to reduce the communication overhead. This is important as the time spent communicating is usually time that the nodes are idle. As a consequence, the performance of parallel algorithms can be dramatically affected, if nodes spend too much time communicating, especially if there are tens or hundreds of nodes involved. Therefore, it is of crucial importance to design algorithms with a reduced communication cost.

1.1 Scope

The goal of this project is to develop a novel algorithm for the distributed training of neural networks that performs well in terms of computational time and scalability, and at the same time, it produces networks with good generalization properties.

For this project, we develop a distributed variant of stochastic gradient descent method with Nesterov momentum. We investigate multiple communication and synchronization strategies, to obtain a convergent and scalable algorithm that reduces the communication overhead. Moreover, we analyze the effect of the mini-batch size, the choice of learning rate and the number of computing nodes, on the scalability of the algorithm.

The project is developed using Python. The deep learning model, as well as the training algorithms, are implemented using the library Pytorch with NCCL (NVIDIA Collective Communication Library) backend for the communication. The developed algorithm is tested using the Piz Daint supercomputer at CSCS (Swiss National Supercomputing Centre).

This report has the following structure. In Chapter 2 we present an overview of machine learning algorithms and parallel programming, needed for the project. Chapter 3 states the project requirements, software as well as hardware. In Chapter 4 the novel, hybrid synchronous parallel stochastic gradient descent algorithm is presented and explained. In Chapter 5, we discuss the numerical experiments and tests that have been performed. In the end, Chapter 6 concludes and summarizes the project.

2 Background

2.1 Types of Machine Learning algorithms

> A computer program is said to learn from **experience E** with respect to some class of **tasks T** and **performance measure P**, if its performance at tasks in T, as measured by P, *improves* with experience E.
>
> *Tom Mitchell*

In this chapter, we introduce the concepts and terminology behind this project. Machine Learning is the part of Artificial Intelligence that has the purpose of developing algorithms that can learn and improve autonomously through the use of data. Machine learning algorithms have a vast range of uses and it has become one of the primary focuses of computer science.

There are essentially three types of machine learning algorithms:

- *Supervised Learning*
- *Unsupervised Learning*
- *Reinforcement Learning*

In the following section, we discuss these three types of algorithms. The focus is placed on *Supervised Learning*, as it is the main focus of this thesis.

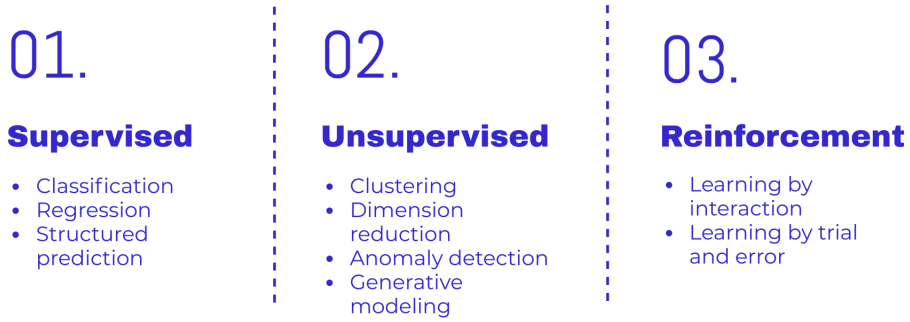


Figure 1. Types of machine learning algorithms.

2.1.1 Supervised Learning

For a given a dataset $T = \{(x_i, y_i)\}_{i=1}^{n}$, consisting of a pair of input feature $x_i \in X$ and label $y_i \in Y$, the goal of *supervised learning* is to learn a mapping function $g : X \to Y$, which is able to predict the correct label: $g(x_i; \theta) = y_i$, where θ represents the parameters of the model. Classical examples of supervised learning are *classification* and *regression*. In the latter, the function aims to map input data to a continuous output, in the former the goal is to map data to a discrete set of categories.

The basic supervised learning pipeline is shown in Figure 2. The learning methods expect a standardized representation of the data. After that, the dataset is split into *training and test sets*. The training set is used to find a *decision rule*, which means that the *model* (e.g. a neural network) is trained to predict the label given the data sample and correct label. This phase can be called *model fitting* and the model is the output of the learning algorithm. After this stage, the test data is used to validate the model by comparing the true labels with the predicted ones. The goal is to find a good balance between complexity and how good the model is at predicting labels. Thus, a good model should be statistically and computationally efficient. In the end, we have a model that we can use to predict the labels of unknown samples.

To get into the details, there is a probabilistic assumption: the learning algorithm assumes that the data is sampled from the same probability distribution D and the goal is to find g that minimizes $\mathbb{E}[l(g(x; \theta), y)]$. That means that it minimizes the expected value of a continuous and differentiable *loss function* l: $Y \times Y \to \mathbb{R}$, which measures the difference between the true label and the predicted label. The smaller the expected loss, the better the mapping

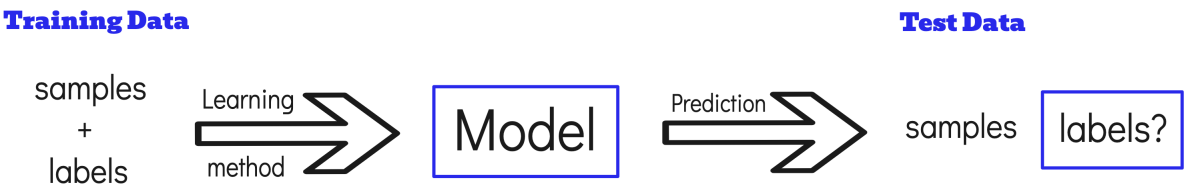


Figure 2. Supervised Learning Pipeline.

function.

In regression tasks, we want to predict labels $y \sim Y$ from new and unseen samples $x \sim X$ by finding the relation between X and Y. Some examples of per-sample loss functions for regression are the *squared difference* ($l(y_{predicted}, y_{true}) = (y_{predicted} - y_{true})^2$) and the *absolute difference* ($l(y_{predicted}, y_{true}) = |y_{predicted} - y_{true}|$).
In classification tasks, we want to assign a sample $x \sim X$ to a discrete class $y \sim Y$. One loss function that can be used is the *cross entropy* ($l(y_{predicted}, y_{true}) = -\sum_i y_{true_i} log(\phi(y_{predicted_i}))$), where the output of the model is passed to the *softmax function* $\phi(y_i) = \frac{e^{y_i}}{\sum_k e^{y_k}}$ in order to get the normalized probabilities over the prediction categories.

The *model fitting* phase works as follows:

1. Feed the sample to the model and get the prediction
2. Compute the difference of the prediction with respect to the label
3. Use this information to update the parameters of the model

Finding the best parameters of the model is an optimization problem and the most widely used solution for minimizing the loss function is the Stochastic Gradient Descent method [39]. This method will be covered in detail later in chapter 2.3.

One of the possible applications for supervised learning are image captioning [52], spam detection [17], price prediction [21] and fraud detection [49].

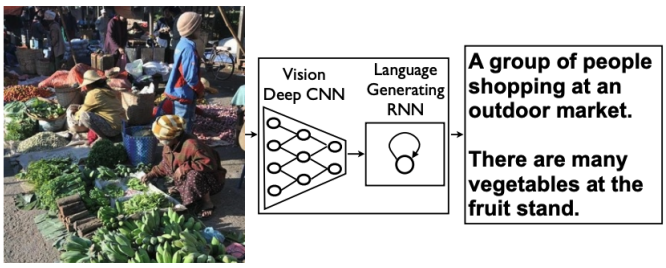


Figure 3. Example of an image captioning network: CNN followed by a language generating recurrent neural network.

2.1.2 Unsupervised Learning

In *unsupervised learning*, as shown in Figure 4, the learning happens without the labels. Some examples are *clustering* (unsupervised classification) [35], *dimension reduction* (unsupervised regression) [13] and *generative modeling* (autoencoders [5] and GANs [29]). Possible applications of these methods include the identification of latent variables [19], information compression [51], feature learning [15] and anomaly detection [1].

Clustering works by dividing a dataset without labels into clusters such that the samples in one cluster are more similar than those in other clusters.

Dimension reduction is very useful, for example for visualization purposes, as it is a method for lowering the dimensions of high-dimensional data. One of these techniques is *principal component analysis*, where we exploit the largest eigenvalues (the principal components) to reconstruct the data in a lower dimension without losing too much information about the dataset. In the context of Generative modeling, two very interesting applications are *autoencoders* [5] and *generative adversarial networks (GANs)* [29]. The former is a neural network that is trained to output exactly the same value that is fed as input to the network resulting, for example, in denoising filters applications. The latter is a technique to generate realistic data. This method works by having one neural network trained to generate "real" data samples, and another trained to distinguish real from generated samples. In Figure 5 there are some examples of GAN generated images [7]. When GANs are used for solving visual tasks, the second network is usually a convolutional neural network (CNN).

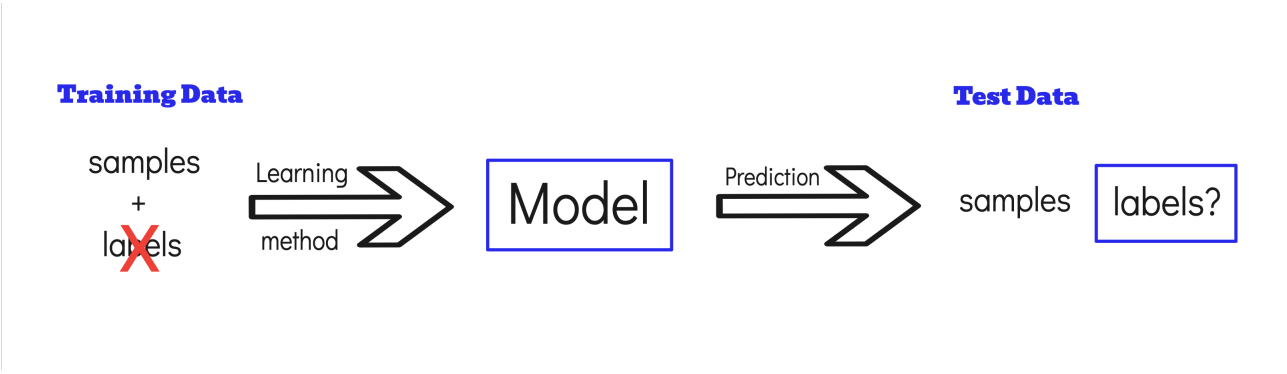


Figure 4. Unsupervised Learning Pipeline.

Figure 5. Examples of samples generated by a GAN.

2.1.3 Reinforcement Learning

In *reinforcement learning*, we want to learn a *policy* by trial and error. A policy can be seen as a mapping from an observed environment state to an action to be executed. This action represents the best one across a wide range of possible behaviours, which will maximize a specific reward, or minimize some time-dependent cost. The main goal is to learn a series of actions that represent the best long-term strategy.
Also in this type of machine learning problem, convolutional neural networks are used for visual tasks, for example for robotic manipulation or for playing video games [31].

In order to be able to achieve the best long time behaviour, these algorithms need to have some sort of *planning* built-in, but given the sequential nature of the decision-making process, this is not straightforward. In recent years there have been some improvements for example by using *value iteration networks* [46]. These neural networks have a *planning module* inside that enables planning-based reasoning for predicting the best long-term outcome. The network is able to learn to plan the best strategy, for example in route planning, by taking the shortest path from point A to point B. This type of network achieve better generalization to new and unseen tasks, Figure 6 illustrates some examples of generated maps and images of the surface of Mars.

2.2 Models

There are multiple models that can be used in a learning algorithm. Some of them are *linear regression*, *decision trees*, *support-vector machines* and *artificial neural networks*. In the following section, the focus will be on neural networks.

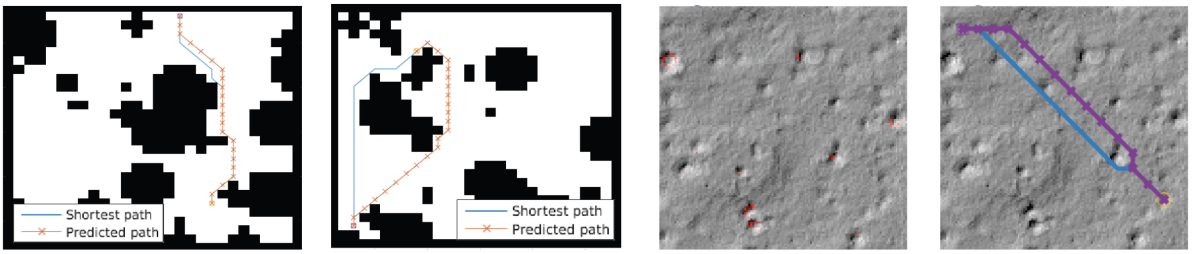


Figure 6. Examples of shortest path computation using *value iteration networks*.

2.2.1 Neural Networks

The human brain is made of more than 10 billion neurons, each one connected to other neurons. Together they give us the power to succeed at very hard cognitive and intellectual tasks. The *neuron* is the building block of *our neural network* and can communicate with others by sending *impulses*. An impulse (action potential) is sent when the electrical potential inside reaches a threshold (the neuron has then "fired") via the *synapse*, a structure that enables the electrical pulse to travel to another neuron. With this chemical mechanism, the neurons can receive, process and send pieces of information.

The *artificial neural network* (ANN) has been thought as a generalization of the theoretical models of these biological structures and chemical reactions. The *artificial neuron*, or *node*, is the simplified model of the real counterpart. The *weights* of an ANN represent the modulation of the connection between neurons, the electrical potential threshold is represented by an *activation function* and this adds nonlinearity to the network. Popular activations are the *sigmoid* ($\phi(x) = \frac{1}{1+e^{-x}}$), the *Rectified Linear Units* ($\phi(x) = \max(0, x)$) and *softmax* ($\phi(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$ with K possible classes), which is often used as last activation function. The impulse then becomes the sum of input signals plus a bias (to shift the activation function) that get transformed by the activation function. Putting all this operations together, we get the function g that represents these transformations.

When training an ANN (that from now on it will be simply called neural network), the weights are changed (updated) in order to let the network *learn* something. The same process happens in the brain, where some connections between neurons grow stronger, giving us the ability to learn. In Figure 7 it is possible to see one neuron with two inputs $x1$ and $x2$, weights $w1$ and $w2$, bias $b1$ and activation function ϕ. The output of the neuron is then given by $y = \phi(\sum_j w_j x_j + b)$ where j is the index of the input.

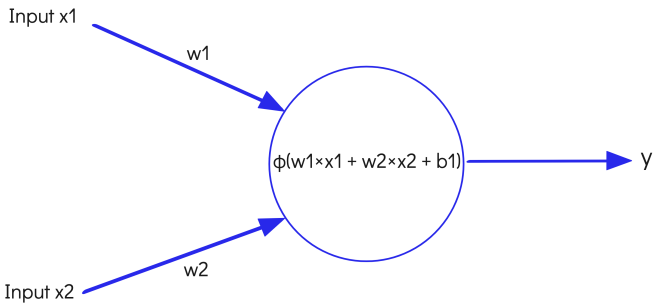


Figure 7. Example of an artificial neuron.

The neurons are grouped into *layers*, the *input layer* and *output layer*. There can also be *hidden layers*, as shown in Figure 8, and they are strictly connected with the layers before and after. In *feed-forward neural networks*, information flow unidirectionally from the input to the output. In contrast, recurrent neural networks [43] allow previous outputs to be used as inputs in the same layer (feedback connection).

The *model fitting phase* that was introduced in the supervised learning section applies to neural networks as follows. Before starting the learning process the weights and biases are randomly initialized (other strategies as *pre-training*

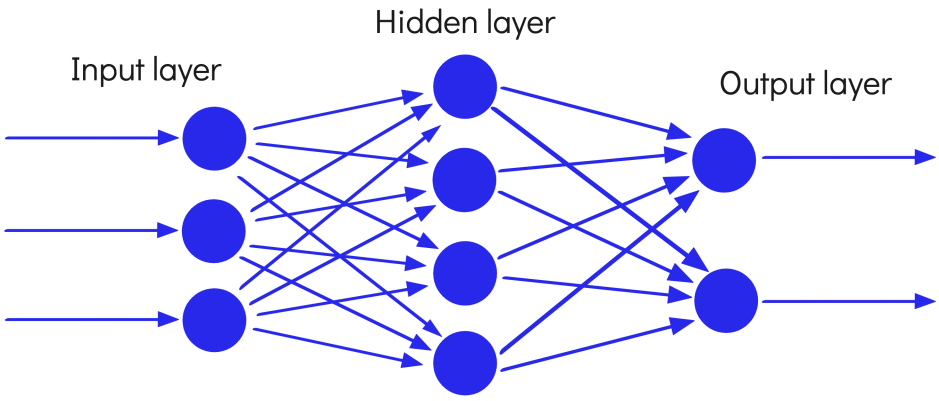


Figure 8. Example of an artificial neural network, information flow from left to right.

can be used). Then an input vector x is selected from the training set together with the desired output y.

1. The learning algorithm does a *forward pass*: the input vector is passed to the network which computes an output (prediction) by passing the values through the network (Figure 8), where they are computed and elaborated inside the neurons (Figure 7). The forward propagation through the neural network can be seen as

$$g(x;\theta) = \phi^L(W^L\phi^{L-1}(W^{L-1}\phi^{L-2}(W^{L-2}\cdots\phi^1(W^1x+b^1)\cdots+b^{L-2})+b^{L-1})+b^L),$$

   where L is the number of layers, θ represents the parameters of the function (weights and biases here), and W^L is a matrix containing all the weights at layer L.

2. The error (difference between predicted value and given label) is computed by using a loss function $l(g(x_i;\theta), y_i)$ (for example with the cross entropy loss function, see 2.1.1).

3. The goal is to minimize the loss function for all samples in the dataset. To this aim, the optimal parameters θ are found by solving the following optimization problem:

$$\hat{\theta} = \arg\min_{\theta} L := \sum_{i=1}^{n} l(g(x_i;\theta), y_i),$$

   where $\hat{\theta}$ provides the minimal loss. Note, the minimization is performed by utilizing all n samples contained in dataset T. As mentioned above, a common way to solve this optimization problem is by employing the stochastic gradient descent algorithm, which requires the gradient of the loss function L.

4. The gradient of the loss function is computed with respect to each weight and bias by the chain rule. For each layer, starting from the last one, the gradients are computed iterating backwards until the first layer. This is called the *backpropagation* phase.

5. The weights and biases are then updated by taking a step in the negative gradient direction. Here is the update if we do it for one sample

$$\theta \leftarrow \theta - \eta\nabla l(g(x_i;\theta), y_i),$$

   where η is the learning rate that determines the step size and $\nabla l(g(x_i;\theta), y_i)$ is the gradient of the loss function evaluated for the data sample (x_i, y_i). After this step, another sample x is passed to the network and we repeat the whole process.

The process described above is called *learning*, or *training*, and its objective is to obtain a model which generalizes well to unseen data. If the network is trained for too long or with too many parameters, *overfitting* can occur. This happens when the network learns really well to predict the training data but it cannot be generalized to other data (this results in *high variance* when the model is too sensitive to small fluctuations in the training set). *Underfitting* can occur when the network can't correctly predict train data and can't generalize to other data (this results in *high bias* when the model is not able to find the relevant relations in the dataset). Ensuring the *bias-variance tradeoff*, we can obtain a model which fits the training data well and is able to generalize well.

Successfully training a neural network is not straightforward. There are a number of *parameters* that can be tuned

in order to have a fast and successful learning process. Some of them like *learning rate*, *momentum* and *batch size*, will be discussed in the next section. The number of parameters affects the ability of the network to understand the relationships between the data. Too few and the network may be unable to predict the data. Too many and it may overfit, thus losing the ability to generalize. Moreover, it would also increase the computational complexity of the learning algorithm. Another important aspect is the *parameter initialization*, as the initial state of the network is the starting point for the minimization problem. This starting point can be in a place near a local minimum and for this reason, the gradient descent algorithm may get stuck and the network may never improve. Alternatively, the starting point might be too far from a minimizer and it would take too much time to converge.

The complexity of the functions that a network can approximate increases when adding hidden layers or increasing the depth of a given layer. These *deep neural networks* can be useful thanks to their ability to model more complex non-linear relationships but, as seen above, increasing the network size exacerbates the possibility of overfitting. Popular countermeasures are

- *early stopping* when the validation error (prediction error on some validation set) starts to increase,
- adding a *regularization term* to keep the weights small, for example

$$\hat{\theta} = \arg\min_{\theta} \sum_{i=1}^{n} l(g(x_i;\theta), y_i) + \lambda \, ||\theta||_F^2 ,$$

- *dropout*, by randomly dropping out hidden units during each iteration of the training with probability p.

One of the many applications of DNNs is *image recognition* and, in general, image-driven recognition tasks. For solving these types of problems one type of DNNs is usually employed, *convolutional neural networks* [6].

2.2.2 Convolutional Neural Networks

One of the drawbacks of ANNs is that they struggle with the complexity of image data. If we consider a coloured image of dimensions 72×72, it is clear that as we have one input neuron for each pixel, we would have $72 \times 72 \times 3 = 15,552$ weights for each neuron of the first hidden layer. Normal neural networks can quickly become really big, and this can cause performance and memory issues and also, as seen above, overfitting. Here is where *convolutional neural networks (CNNs)* come into play, as these networks have an architecture that is better suited for images. The predecessor of CNN was the *neocognitron*, proposed in 1980 [16]. After 10 years the modern framework for the CNNs was presented [25]. For the next 20 years, the research continued and in 2012 the network *AlexNet* was introduced [23]. This network won the ImageNet Challenge and sparked a new interest in the field.

CNNs are good at transforming images into a smaller form while not losing the original *features*. In order to extract these features from an image, *convolutional layers* are used. The structure of typical CNN can be seen in Figure 9. The main parts of this architecture are:

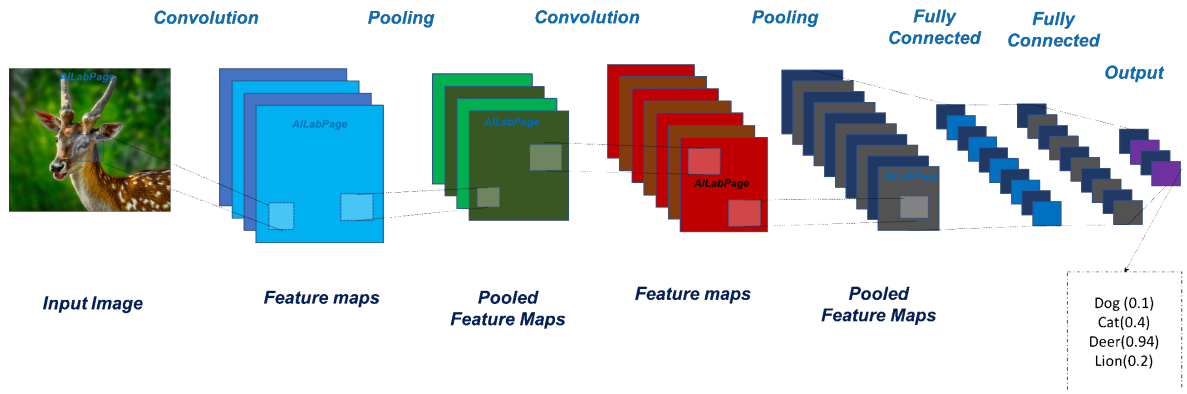


Figure 9. Example of a convolutional neural network [18].

- *convolutional layers*, which extract the *features* of the images,
- *pooling layers*, which downsample the feature maps by reducing the number of parameters. The two common pooling types are *max* and *average*,

- *ReLU*, which performs a nonlinear operation that replaces all negative pixel values in the feature map by zero,
- *fully connected layers*, which have the same functionality as a standard ANN and output the K-vector of probabilities for classification (K number of possible classes).

To get into more detail, when an image is passed through the network, it usually goes through multiple layers of convolutional and pooling layers.

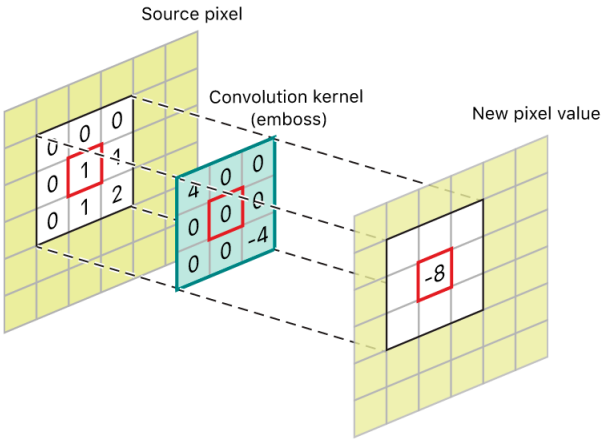


Figure 10. A kernel used to convolve over an image [4].

The convolutional layer takes the image as input (height x width x channels) and uses *kernels* to convolve over the image to produce a *feature map*. The kernel is used to compute the scalar product with the pixels of the image and it creates a new "image" (Figure 10). The result is a set of feature maps, images that show some specific characteristics, or *features*, of the originals (Figure 11). These can be edges, corners, shapes or even textures. For each image, a convolutional layer extracts an arbitrary number of feature maps and this process can be done multiple times in order to refine the extracted features. This step is essential for extracting the correct and most useful characteristics of the images. It also enables the learning algorithm to focus only on the details relevant to the task. Another benefit is that with this process the features of the images don't need to be always in the same place and with the same orientation. The size of the feature maps can be changed by tuning the *stride* (with stride 1 the kernel moves by one pixel), the *depth* (the number of kernels used) and the *zero padding* (the process of padding the border of the image).

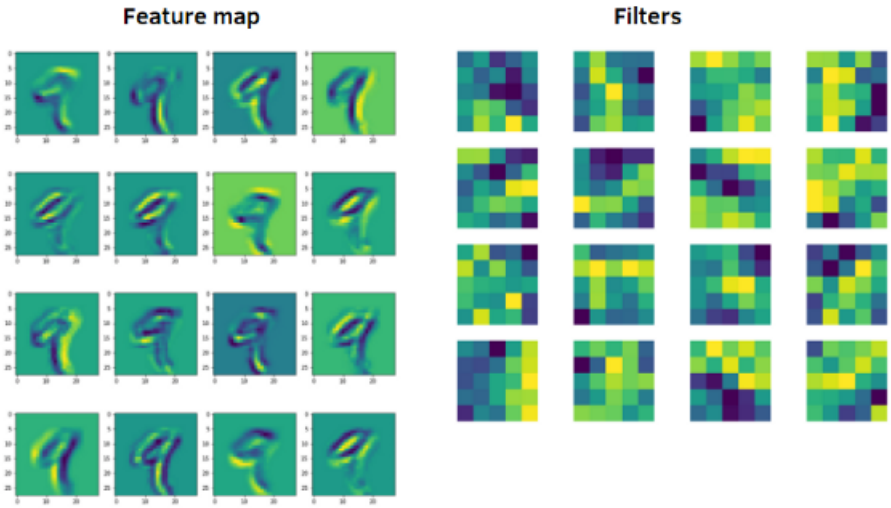


Figure 11. On the left, the feature maps after the convolutional step, the contours of the number were clearly picked up by the convolution. This means that the network learned these features. On the right, the filters that were used to produce the feature maps, these are learned by the network [3].

The pooling layer is used for downsampling. It is common practice to let the feature maps go through the pooling layers to reduce the dimensions while retaining the most important characteristics. This is helpful to reduce the parameters of the network and hence the computational effort and memory complexity. This process works by going over each map with kernels of dimension 2×2 or 3×3 (it is good practice to use small kernels as bigger kernels for pooling can degrade the quality of the feature maps) and using the *max* or *average* function. With a kernel of dimension 2×2 and stride 2, the dimension of the maps can be reduced by 75%.

The last part of a CNN is the fully connected layer, where each node is connected to every node in the adjacent layers. In preparation for this part, as explained above, the images are convolved and pooled multiple times to reduce the size by keeping the important features and, before entering the fully connected layer, the maps are flattened. After this operation, the fully connected layer computes the class probabilities and predicts the label.

CNNs learn the same way as normal ANNs with forward pass and backpropagation, but here not only the weights and biases of the fully connected layers are trained, also the kernels for the convolutional layers. Again with these networks, the *gradient descent algorithm* is commonly used to find optimal network parameters.

2.3 Training

The main part of the *training* process is the optimization of the parameters. The goal is to minimize the loss function over all the data, i.e.,

$$\min_{\theta} L := \sum_{i=1}^{n} l(g(x_i; \theta), y_i), \tag{1}$$

by finding the suitable parameters θ.

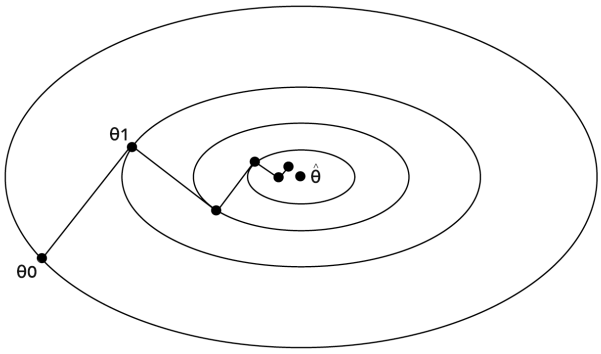


Figure 12. Gradient descent steps towards the local minimum at the center of the level curves. In the context of training a model, every point represents a value of the loss function, by optimizing the parameters of the model, the goal is to reach the minimizer $\hat{\theta}$, i.e. the smaller error possible when predicting the labels.

2.3.1 Gradient Descent

The minimization problem (1) is often solved using the *gradient descent* algorithm. This optimization algorithm is an iterative method that finds minima by using the gradient of the loss function L. As shown in Figure 12, the algorithm starts at point θ_0 and reaches the minimum by following the negative gradient. Each iteration is computed as

$$\theta_{k+1} = \theta_k - \alpha_k \nabla L_k,$$

where the next point θ_{k+1} is given by subtracting scaled gradient direction from the current point θ_k. The function that we want to minimize has to be differentiable, this is required as gradient descent is a first-order algorithm, thus it only requires the gradient when updating the parameters of the function.

The standard gradient descent algorithm, also known as *batch gradient descent*, works by using all the training data at every iteration. The entire training dataset is used in one forward pass to compute the average gradient used for the update. This can be challenging and computationally inefficient if we have a very large dataset. Assuming that L is Lipshitz continuous and convex, gradient descent converges at a rate of $1/t$ (with t being the iterations) [41].

2.3.2 Stochastic Gradient Descent

To reduce the computational cost, we can randomly pick one data sample at each iteration, this algorithm is called *stochastic gradient descent* [40]. By using only one example for each iteration, we are able to use fewer computations at the cost of having a lower convergence rate, as noise is added by the fact that at each update we may take a step that is too long or not in the best direction for the minimizer (Figure 13). The algorithm for SGD is shown below.

Algorithm 1 Stochastic Gradient Descent.

Require: Training Set T; Learning Rate η; Number of epochs E
  Initialize parameters θ
  for $i = 1$ to $E \cdot |T|$ **do**
    Pick data point (x_i, y_i) uniformly at random from T
    $grad = \nabla l_i(g(x_i; \theta), y_i)$ ▷ Compute gradient with respect to the parameters θ
    $\theta = \theta - \eta \cdot grad$ ▷ Update the parameters
  end for
  return θ

As mentioned, this algorithm can oscillate a lot but this behaviour can be mitigated by choosing a small *learning rate* η and decreasing it dynamically. Adaptively reducing the step length helps the algorithm to converge to the optimum point without jumping out. The convergence rate is $\mathcal{O}(1/\sqrt{e})$ [32] (where, in this case, e is the number of iterations and L is convex and Lipshitz continuous).

2.3.3 Mini-Batch Gradient Descent

In an effort to reduce the fluctuation and noise of stochastic gradient descent and the computational effort of batch gradient descent, a balance between both has to be found. This middle ground is *mini-batch gradient descent*, which updates the model parameters with respect to the average gradient using a mini-batch of data samples. This works by splitting the dataset into small batches and exploiting the more stable convergence of batch gradient descent, and the computational and memory efficiency of stochastic gradient descent (also it has fewer weight updates than SGD). When the algorithm passes through the dataset once, an *epoch* is passed, and B is the size of the batches. The algorithm works as follows:

Algorithm 2 Mini-Batch Gradient Descent.

Require: Training Set T; Learning Rate η; Number of epochs E; Batch size B
  Initialize parameters θ
  Shuffle Training Set T
  Divide Training Set T into batches of size B ▷ $1 < \mathrm{B} < |\mathrm{T}|$
  for $i = 1$ to $\frac{|T|}{B} \cdot E$ **do**
    Pick batch b
    $grad_b = \nabla \sum_{j=1}^{B} l(g(x_j; \theta), y_j)$ ▷ Compute gradient for all $x_j, y_j \in b$
    $\theta = \theta - \eta \cdot grad_b$ ▷ Update the parameters
  end for
  return θ

Batch size The choice for the *batch size* is crucial for the success of the learning algorithm. There is no consensus in the literature on which batch size to choose (see discussions on current state-of-the-art frameworks [28] and [42]). If the size is large this should lead to better throughput but poor generalization, if it is too small the algorithm may not converge. Reference [42] specifically shows that increasing the batch size initially decreases the required number of iterations until it is no longer the case. The maximum size can vary significantly between different tasks and depends on the model, algorithm and dataset. Furthermore, there is no evidence at the moment, that increasing the batch size degrades the quality of the model, but using regularization techniques is crucial. In [44] the use of vast batch sizes is recommended to achieve the benefits of decaying the learning rate. The best practice is to investigate batch sizes each time for each application.

Learning rate One possible solution to the degradation of the model's generalization ability to use large batch sizes, is the use of a large *learning rate*. As the batch size grows, the learning rate should also grow, but it is more advisable to keep both low (see [22]), where a high correlation between the two was found. The learning rate decides the step size taken by the algorithm, it is usually good practice to decay it, as we approach the minimizer in order to

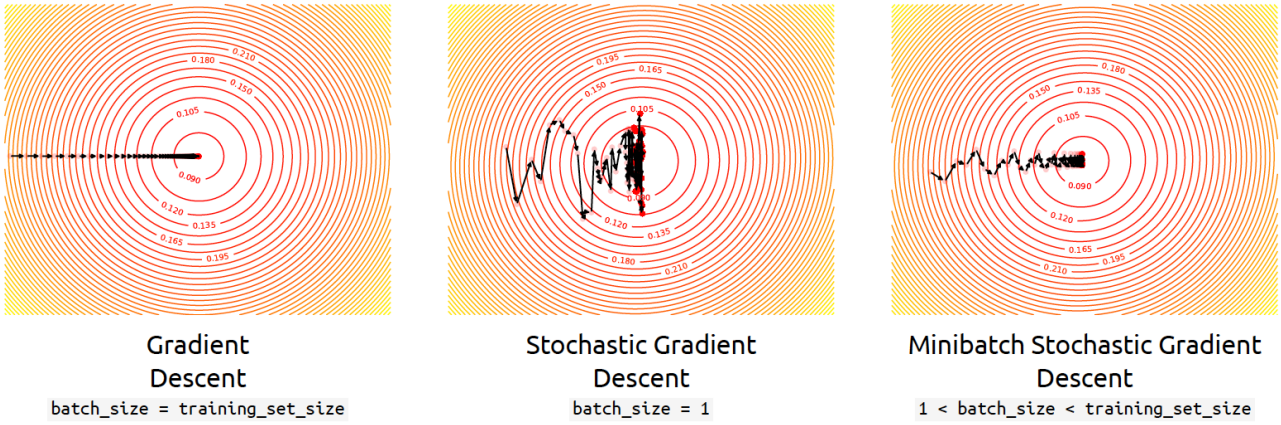


Figure 13. Comparison of gradient descent, stochastic gradient descent and mini-batch gradient descent. We can observe the noise introduced by SGD and how, merging the first two methods, mini-batch SGD manages to mitigate the fluctuation and shows good convergence [12].

prevent going out of the valley. It is of crucial importance to find the optimal learning rate for the particular task at hand.

Momentum Popular variants of stochastic gradient descent are *SGD with momentum* and *SGD with Nesterov momentum* [6]. These variants help to avoid local minima and can speedup the convergence by accelerating the algorithm. Given the momentum term μ, the parameters are updated by the following update rule:

$$v_t = \mu v_{t-1} - \eta \nabla l(g(x;\theta), y),$$

$$\theta = \theta + v_t.$$

The momentum term decreases updates that change direction, and as a result, the updates become faster when approaching the minimum, reducing the oscillation. The Nesterov momentum has a small change to the normal momentum, the gradient is computed from a position different from the current,

$$v_t = \mu v_{t-1} - \eta \nabla l(g(x;\theta + \mu v_{t-1}), y),$$

$$\theta = \theta + v_t.$$

This update "looks ahead" to where the future parameters will approximately be, preventing the algorithm from going too fast and overshooting.

2.4 Parallel Programming

In this section, we present the relevant concepts of *parallel programming* applied to machine learning.

2.4.1 Parallel hardware

GPUs, or *graphics processing units*, are the leading platform for running learning algorithms. These are specialized to rapidly perform parallel operations on data and are commonly used in personal computers, smartphones and game consoles. Another primary use of GPUs is in high-performance computing (HPC) for computing heavy and intensive tasks, for example for training machine learning models. For this reason in the last years, more than 70% of the experiments done in machine learning papers are performed using GPUs [6].

The algorithms that are suited for parallel execution need the correct hardware to be able to run. There are two types of parallel hardware architectures, *single-machine* and *multi-machine*.

Single-machine Single-machine is the foundation of every modern computer processing unit. Today, even smartphones are multicore systems and can exploit the available resources to execute multiple processes at the same time. This is also possible thanks to *out-of-order execution*, which lets the processor avoid being idle by executing the instructions in a different order from the original program. When programming an algorithm for such architectures, one has to choose between using *multiple threads* that share the same memory space, or *multiple processes* which have their own memory space.

Multi-machine Given the improvements in the hardware and the more accessibility to HPC clusters, the interest in training models on multi-machine has grown in the last few years. This type of architecture is used to train very large models that require a lot of computational power or very large datasets that require a lot of memory for storage, and that single machines would not be able to complete, or even fit. In this setting, the communication between the nodes can become a bottleneck when training on large a number of nodes. The latency, bandwidth and throughput on the communication network between the nodes are slower than intra-communication, for this reason, technologies like *InfiniBand* [36] have been developed to bridge the gap, and strategies to reduce communication overhead have to be employed. Frameworks like Message Passing Interface (MPI) and NVIDIA's NCCL, implement low-level communication functionalities that can be used by programmers to run their algorithms on these multi-node environments.

2.4.2 Model parallelization

To exploit the parallelization power of the hardware, multiple strategies can be used with different goals. These can be divided in three categories, *model parallelism*, *data parallelism* and *layer pipelining*.

Figure 14. Model Parallelism: the model is split between three nodes, the data is replicated. Image from [6].

Model parallelism The first strategy partitions the model between the nodes and thus splits the memory footprint. The same mini-batch is copied into all the nodes as well as a part of the model as shown in Figure 14. The challenge is the interdependency between the partitioned parts of the model as the layers of the networks are cut. The batches are processed at the same time on all the nodes, for this reason, for the forward pass, the nodes have to communicate the weights to the other interested nodes above and below. The splitting of the model should be carefully done in order to minimize the number of parameter exchanges. Strategies have been proposed to reduce communication but this type of parallelism is regarded as difficult and inefficient, due to the sequential nature of the forward pass. It can be considered the last resort and only really useful for situations where the model is too big for one node's memory. This is the reason behind the push to data parallelism [24].

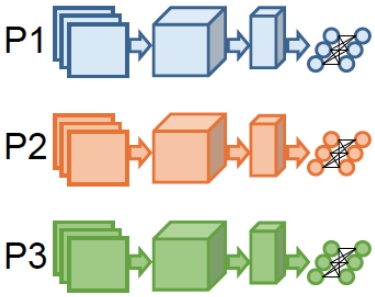


Figure 15. Data Parallelism: the data is split between three nodes, the model is replicated. Image from [6].

Data parallelism Data parallelism is a strategy that splits the data and replicates the same model across the nodes. The goal is to increase the throughput of the samples and exploit the parallelization capabilities offered by the hardware (Figure 15), as the forward pass and backpropagation are independent between batches. This strategy works by sending to each node a copy of the model and a partition of the data. Each node can then pass a mini-batch through the model and perform the backpropagation. At this point, the gradients are synchronized between the nodes, averaged, and used to update the local models. In this way, the models remain the same across all nodes. Another possibility is to share the updated model parameters directly and perform the same averaging. The mini-batch size can be given overall, and so each node uses $b = batch\_size/number\_of\_nodes$, or the size is given for each node and the effective overall size is $b = batch\_size * number\_of\_nodes$.
The communication overhead is reduced with respect to model parallelism, but different strategies can be employed

to further eliminate the overhead. As the communication cost is determined by the model size, the communication complexity of the algorithm increases with the model size.

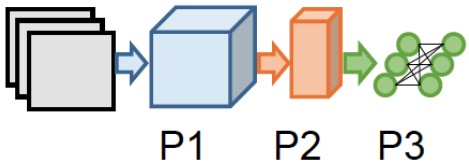


Figure 16. Layer Pipelining: the model is split vertically between three nodes, each node gets one or more entire layers. Image from [6].

Pipelining In Figure 16, layer pipelining is displayed. The model is split vertically across each node which computes the values for their own neurons and passes them to the next node which does the same. The partition of the layers can be decided to make the communication more efficient but the ideal partition is model-specific and depends on the number of nodes. As the layers stay entirely on the same node, there is no need to keep in memory the parameters of other layers, and the weights don't need to be synchronized and thus communicated.

Another kind of *Pipelining* can also be used to speed up computations and increase data throughput. This strategy works by overlapping the computations of the forward pass and backpropagation with weight update, as the data is available. Other possibilities for parallelism consider hybrid approaches, by combining the strategies presented above.

2.4.3 Centralization

After the backpropagation phase where each node computes the gradient with respect to each parameter, there is the *optimization step*, where the parameters of the model are updated. This step can be *centralised* or *decentralized*.

Centralized The centralized optimization step works by having a *parameter center* that stores the model parameters. The gradient computations are executed in the nodes and sent to the parameter server. The parameters are then updated in the server and sent back to the nodes. If this is used in the context of data parallelism, the server averages the gradients before changing the parameters. Centralized optimization asks the nodes to compute the gradients, therefore, distributing this expensive task, while letting the server keep track of the current state of the model. This creates a producer-consumer relationship between the nodes and the server where the model is always consistent across the nodes. The simplification and consistency of the data handling are counteracted by the fact that after each update of the model, the server has to send the network to each node and, for large networks, this can quickly increase the communication overhead.

Decentralized Decentralized optimization gives the nodes the task to update and maintain the model while keeping it synchronized between each other. The only communication needed is the synchronization between the nodes, and this can be mitigated by using different strategies. If the strategy involves a synchronization at each iteration, then the model will be consistent across the nodes.
If the communication takes place at different intervals with gradient accumulation and without updating the local model, then the model is still consistent across the nodes and this strategy effectively uses an overall batch size that is bigger than the local batch size (*overall_batch_size* = *local_batch_size*interval*).
Instead, if the nodes update the local models in the interval between the synchronizations, then the local models associated with each node will differ. This means that each node picks another path on the loss function landscape and explore the local minima and maxima around them. After some iterations, the models are synchronized by averaging and, effectively, the nodes "get back together". From this point, the nodes can differ again and be synchronized again. If the interval is too large, the model may diverge too much and the averaging may destroy the progress made. Rather, if the interval is selected carefully, the exploration of the nodes can be helpful to avoid getting stuck in sharp local minima, this could affect the quality of the model. If one node is stuck in a local minimum but the others are outside, the averaging of the parameters will take the node that is stuck out. The minimum is then found when the majority of nodes have found it.

2.4.4 Scheduling

The scheduling of the algorithms can be *synchronous*, where all the computations across nodes occur at the same time, and *asynchronous*, where there is no barrier for the computations. The former ensures that the nodes execute determined operations at the same time by sacrificing an efficient use of the resources. The latter prefers to keep the hardware occupied but risk the degradation of the model.

Synchronous In a centralized synchronous setting, each node has to wait for the parameter server to update the model and sent it back. In turn, the server has to wait for each node to send the gradients before doing the update. This last phase has the processing time of the slowest in the group. The synchronization ensures that every local model is the same across nodes and the parameter server.
In a decentralized synchronous setting, the nodes do all the computation inside and just share parameters. When they are instructed to share and wait for the informations, the fastest nodes need to wait for the slower ones. If some interval between the synchronizations is given, the nodes do the computations until they have to share. In this case, the nodes are then brought to the same level by averaging the parameters, before starting the exploration of the loss landscape again. Choosing the interval affects the time spent evolving the local models in proportion to the synchronization time.

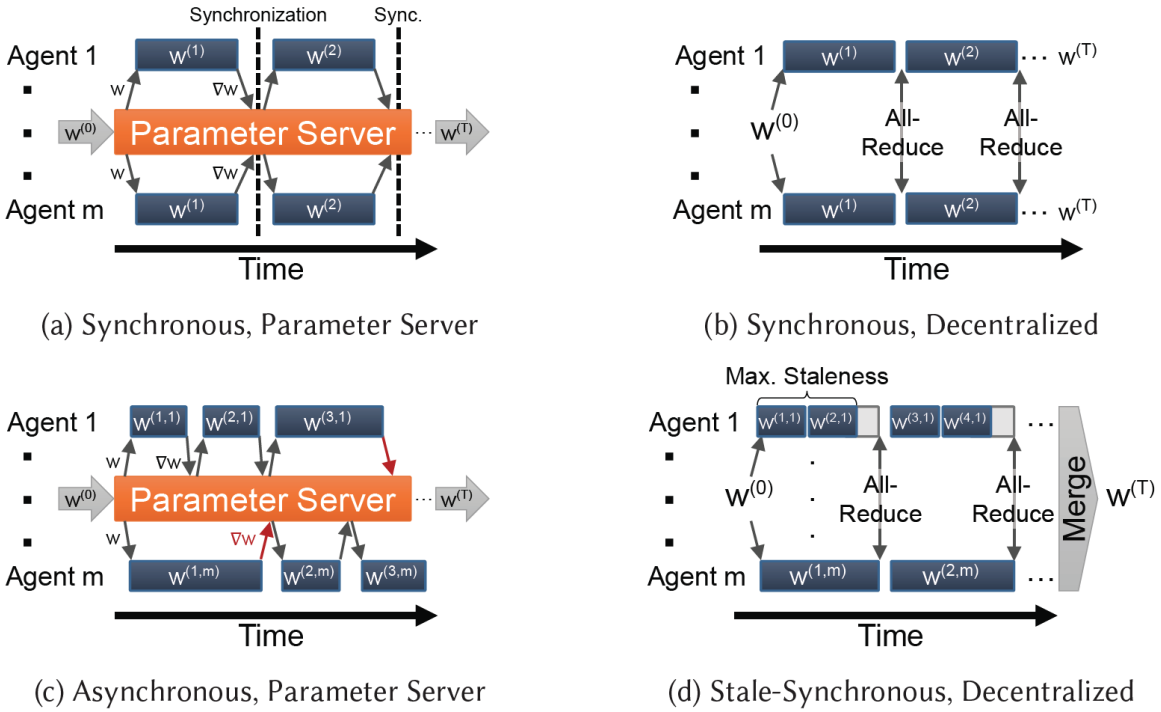
Asynchronous In a centralized asynchronous setting, the nodes send the updates to the parameters server when it is done processing the mini-batch and the server answers directly with the updated model. When the second node sends the gradients, the server computes the updates and immediately sends back the updated model. With this strategy, only one node at a time has the up-to-date model together with the server. This means that after one update, the other nodes are effectively working on a stale model, hence they are using parameters that are outdated and the work could be potentially wasted. If the learning rate is small, even some staleness can improve the model and this strategy maximizes the work time of the nodes. One possible solution to prevent staleness from degrading the model is to bound the asynchronicity that is allowed. If a node stays too much behind in updating the model, the server can reject the gradients and just send back the up-to-date model, or it can perform a global synchronization step (Stale-Synchronous Parallelism (SSP) [20]).
In a decentralized asynchronous setting, the nodes can communicate, when they want, the parameters to the others that can use this information to update the local models. The models, in this context, diverge, as there is no barrier to stop the computations of the gradients and the update of the parameters (nodes don't wait for each other). Also here there can be an interval of iterations before communicating the local model to the others, it is good practice to keep this interval short to prevent the degradation of the model.

2.4.5 Communication patterns

In presence of multiple nodes, the communication overhead can take up a good part of the performance of the algorithm. For this reason, the communication pattern itself is an important part of the definition of the overall strategy.

When dealing with a parameter server, the communication to and from this node is what matters. If the nodes are physically distant from the server, or if the server is not able to deal with a lot of nodes, a good strategy is to split the server job into two or more servers. This way the communication does not create bottlenecks as some parts of the model are in one server, and the others in the other server. This requires a more complex communication strategy where each node has to send and receive the informations to and from multiple servers. Another pattern is given by the splitting of the server into each node. Each node has to send the updates to a specific node depending on the parameter storage location. This pattern also avoids bottlenecks on the network by distributing the communication at the cost of having more complex communication logic into the nodes. This creates a half decentralized strategy.
The decentralized case gives us some more alternatives when it comes to communication patterns. Instead of letting each node communicate with all the others, different patterns can be employed. One pattern could be that each node only communicates with two other nodes, creating a ring structure. The nodes can let the information flow around before starting the new iteration, or they can just stop spreading the information and train the model that is the average of the other two, these two in turn are the average of this node and the other two, and so on. This pattern reduces the number of communication channels, in turn reducing the general complexity of the communications. Another pattern is the hierarchical tree, where the root communicates with two nodes, the leaves with one, and the nodes with three. This is also another way to reduce communication complexity by trimming down the exchanges.

In Figure 17, the two optimization and two scheduling strategies are illustrated [6].



(a) Synchronous, Parameter Server

(b) Synchronous, Decentralized

(c) Asynchronous, Parameter Server

(d) Stale-Synchronous, Decentralized

Figure 17. Communication strategies used for training deep learning models. Image from [6].

2.4.6 Consistency

Consistency across nodes has been already mentioned. Some strategies require the local models to be consistent or, as seen, not too far from each other and not for too long. Especially in asynchronous algorithms, the local models change and evolve differently from each other. There have been proposed strategies that use *inconsistent models* and that provide optimal convergence rates. For instance, the HOGWILD! shared-memory algorithm that allows processors to access the memory and overwrite it destroying each other's work (this is SGD without locking)[33] and it has been shown to converge for sparse problems. This algorithm has since been used in distributed memory clusters.

Some strategies revolve around sparse parameter updates. Especially for large networks, not every parameter changes at each iteration. This can be exploited by synchronizing only some times during training, or only at the end (post-training model averaging), as the local models are not expected to differ too much from each other. These methods can converge but the degradation of the model is more likely to happen, as the different nodes may just converge to different local minimizers.

3 Project requirements and analysis

3.1 Hardware

In this section, the relevant hardware, which was used for running the tests, is presented. The first cluster was used for testing the code and learning about the scheduling system. The CSCS supercomputer was then used to run the numerical experiments.

3.1.1 Euler Cluster

The first cluster is located at Euler institute of USI [50]. The full specification can be found in the Appendix 8.1.

Cluster specification This cluster is composed of 41 nodes and is managed by a master node. The nodes run *CentOS 8.2.2004.x86_64* and provide a wide range of scientific applications. Resource allocation and job scheduling are performed by *Slurm*.

Hardware specification The following are the *multi-gpu* nodes used in the first phase of this work.

COMPUTE NODE - Multi GPU nodes	
Nodes	icsnode[41-42]
CPU	2 x Intel Xeon Silver 4114 CPU @2.20GHz, 20 (2 x 10) cores
RAM	100GB DDR4 @2666MHz
HDD	1x 1TB SATA 6Gb
INFINIBAND ADAPTER	Intel 40Gbps QDR
GPU	**icsnode41:** 2 x NVIDIA GeForce GTX 1080 Ti Titan 11GB GDDR5X 3584 CUDA cores **icsnode42:** 2 x NVIDIA GeForce GTX 2080 Ti Titan 11GB GDDR6 4352 CUDA cores

Network specification

INFINIBAND	Intel True Scale Fabric 12200 Switch 36 ports 40Gbps QDR

3.1.2 Piz Daint

Named after a prominent peak in Grisons that overlooks the Fuorn pass, this supercomputer is the flagship system for the national HPC Service [12]. First installed in 2012, it has become one of the most powerful supercomputers in the world.

Hardware specification Piz Daint is a hybrid Cray XC40/XC50 system.

Cray XC40/Cray XC50	
XC50 Compute Nodes **This is the hardware used for the numerical experiments**	5704 x Intel® Xeon® E5-2690 v3 @2.60GHz (12 cores, 64GB RAM), NVIDIA® Tesla® P100 16GB
XC40 Compute Nodes	1813 x 2 Intel® Xeon® E5-2695 v4 @2.10GHz (2 x 18 cores, 64/128 GB RAM)
Login Node	Intel® Xeon® CPU E5-2650 v3 @2.30GHz (10 cores, 256 GB RAM)

Network specification

Cray XC40/Cray XC50	
Interconnect Configuration	Cray Aries routing and communications ASIC and Dragonfly network topology

Figure 18. The Piz Daint supercomputer. Image from [12] (CC BY-SA 3.0).

Memory and Storage specification

Cray XC40/Cray XC50	
Hybrid Memory Capacity per Node	64 GB; 16 GB CoWoS HBM2
Multicore Memory Capacity per Node	64 GB, 128 GB
Total System Memory	437.9 TB; 83.1 TB
Sonexion 3000 Storage Capacity	8.8 PB
Sonexion 1600 Storage Capacity	2.5 PB

Performance analysis

Cray XC40/Cray XC50	
Peak Floating-point Performance per Hybrid Node	4.761 Teraflops
Peak Floating-point Performance per Multicore Node	1.210 Teraflops
Hybrid Peak Performance	27.154 Petaflops
Multicore Peak Performance	1.731 Petaflops
Sonexion 3000 Parallel File System Theoretical Peak Performance	112 GB/s
Sonexion 1600 Parallel File System Theoretical Peak Performance	138 GB/s

3.2 Software

Written in python, this project is based on the open-source library *PyTorch*, primarily developed by Facebook's AI Research lab.

3.2.1 PyTorch

PyTorch is a scientific computing package used in machine learning research for applications such as computer vision and language processing. PyTorch is widely used today and one example of this is Tesla's Autopilot [38].

Tensors Using the computing acceleration power of GPUs, PyTorch uses *Tensors*, generic n-dimensional arrays, to manipulate large set of data. Similar to NumPy arrays, these arrays have a rich set of operations that can be performed on CUDA-capable GPUs.

Modules PyTorch defines the *nn.Module* class for defining neural networks. This is done by building the network block by block. These blocks can be linear, convolution, relu, pooling and others layers and functions. Furthermore, a Module defines the *forward()* function, this takes the input, for example a tensor, and passes it through the defined layers. For example, given a Linear module, the *forward* function computes the output by multiplying the input with the weights and by adding the biases.
Using these modules, calling the *forward()* function defined inside, we are able to do the *forward pass*, compute the gradients for the parameters using the *autograd engine* (*backpropagation phase*), and update them in the *optimizer step*.

Autograd *Autograd* is the automatic differentiation package that is used for the backpropagation phase. When the *backward()* is called on the loss, autograd computes the gradients of the loss for each parameter and stores them in the *grad* attribute of the respective parameter in the tensor.

Optimizer After that autograd has computed the gradients, the *optimizer* performs the updates of the parameters with the *optimizer.step()* method. The *zero_grad()* method is used to zero the gradients.

Dataset In order to train the neural network, the dataset has to be loaded and shuffled. This is done with the *Datasets* class and, by setting the train flag upon instantiation of the Dataset, it is possible to create a *train set* and a *test set*.
When using mini-batch stochastic gradient descent, the *Dataloader* class is used to slice the dataset and shuffle the samples. This class provides an iterator that can be used to loop over and fetch a different batch every time.

3.2.2 DistributedDataParallel

PyTorch has several tools for data parallelism built-in. One of them is the *DataParallel* module for single-process multi-threaded data parallel training on single-machine architectures that use multiple GPUs. The module for multi-process data parallel training across multiple GPUs and machines is called *DistributedDataParallel*(DDP). This module enables multi-machine data parallelism by providing communication strategies and other functionalities. The difference between these two packages is that DDP uses multiprocessing (a process is created for each GPU), while DataParallel uses multithreading. Thus, it is recommended to use DDP to do multi-GPU training, even if there is only a single node [9]. The building blocks of DDP are depicted in Figure 19.

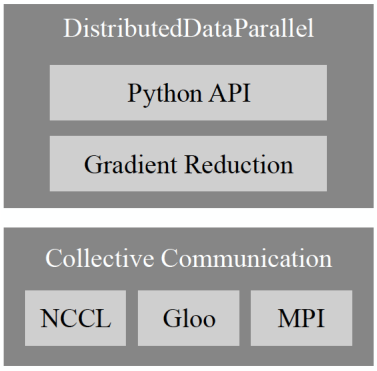


Figure 19. The structure of the DistributedDataParallel framework. Image from [27].

Communication backend The DDP module provides the communication strategies for the distributed training. This allows synchronizing the gradients across all nodes before the optimizer step is taken. Performing the synchronization at every iteration ensures that the parameters of all local models are the same. There are three communication backends supported by the DDP: the NVIDIA Collective Communication Library (NCCL), Message Passing Interface (MPI) and Gloo. For Linux, the Gloo and NCCL backends are default in PyTorch (NCCL only when building with CUDA) and MPI is an optional backend. For the training with GPUs and with InfiniBand connection between the nodes, the backend NCCL is recommended [10]. NCCL implements multi-GPU and multi-node communication primitives optimized for NVIDIA GPUs and Networking [34].

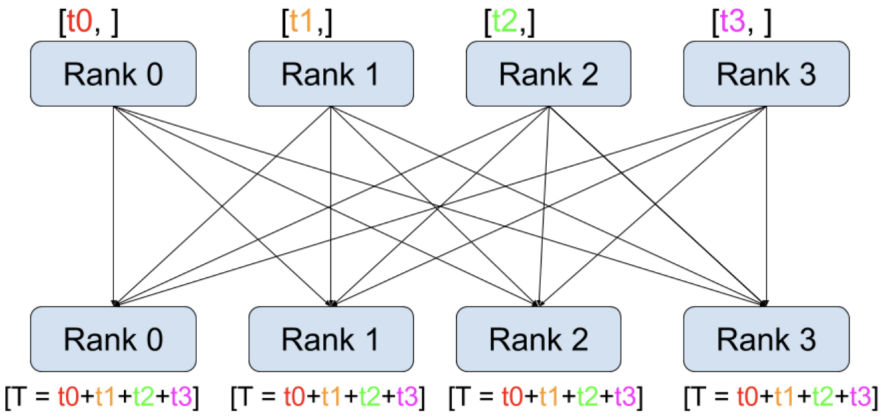


Figure 20. Example of AllReduce operation. Image from [37].

AllReduce The operation of synchronizing the gradients is carried over by the collective communication API called *AllReduce*. This API is supported by the three communication backends supported by DDP. *AllReduce* expects each involved process to send an equally-sized tensor. These tensors are then used for a collective arithmetic operation such as sum, prod, min and max, the result of which is sent back to each participant. This operation only starts when all processes have provided their own tensors, thus being synchronized communication (Figure 20).

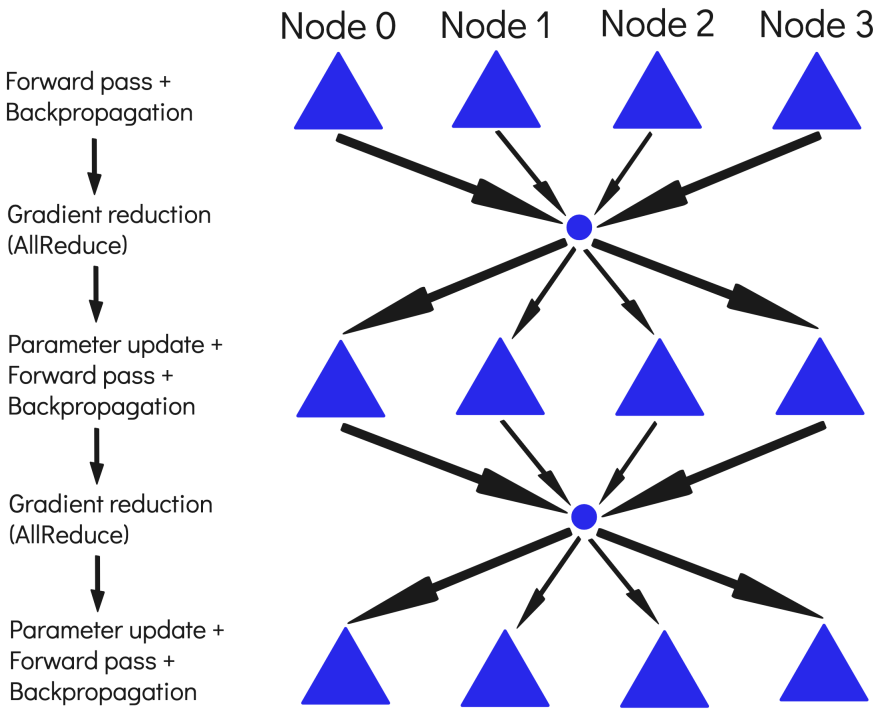


Figure 21. Example of gradient reduction performed by DistributedDataParallel.

Gradient reduction The DDP module ensures model consistency by replicating the same model across all nodes and by letting all of them update their local models with the same averaged gradients. The averaging of the gradi-

ents is performed using the AllReduce operation on the gradients. To speed up this phase, DDP overlaps computation and communication. This is achieved with the use of *buckets* where the parameters are inserted. These buckets tie together different parameters. As soon as the backpropagation is finalized for all parameters in a given bucket, the gradients are communicated to other nodes (Figure 22).

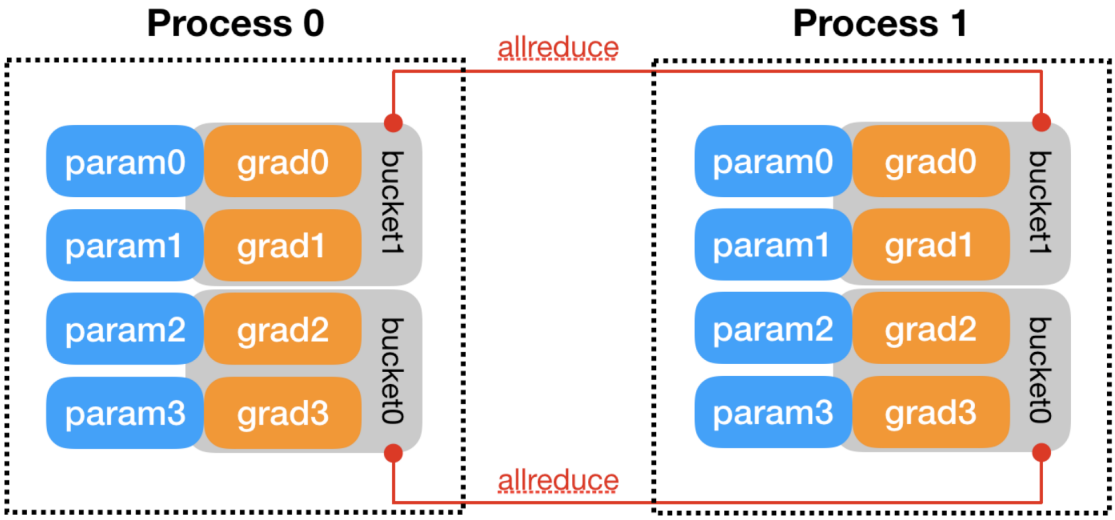


Figure 22. Example of AllReduce operations executed asynchronously to maximize the bandwidth utilization.

The training process works as follows:

- A *ProcessGroup* has to be created to put all the nodes in the same group and to assign the ranks to each node.
- The DDP module takes the model at the node with rank 0 as reference and broadcasts the *state_dict* (a Python dictionary object that maps each layer to its parameter tensor) to all other nodes in the group. As a result, all the local models located at different nodes have the same state.
- The parameters are organized into buckets in the reverse order with respect to the normal parameter order from a given model. This is due to the fact that DDP expects gradients to become ready during the backward pass in that order. Figure 22 illustrates the gradients 0 and 1 which are in bucket 1, and gradients 2 and 3 which are in bucket 0.
- During the forward pass, where the input is propagated through the network, the *autograd graph* is constructed by keeping track of all the operations that created the output. This directed acyclic graph is used in the *backward()* function to automatically compute the gradients using the chain rule. This graph can be traversed looking for the parameters of the model that are not involved in the computations. DDP sets these unused parameters as *ready for reduction*. This is useful as, during the backpropagation part, DDP has to wait only for the gradients associated with parameters that are not ready, avoiding waiting for parameters that will never change.
- The backpropagation phase is then started to compute the gradients with respect to the parameters. These gradients are then marked as *ready for reduction* and, when all the parameters inside a particular bucket are ready, the bucket is asynchronously sent and the AllReduce API waits for the same bucket from all other nodes. Once the AllReduce operation has received all the buckets containing the specific set of gradients, these are averaged and stored in the nodes. (Figure 21).
- DDP blocks waiting for all the AllReduce operations on all buckets. After this process, all the gradients are written in the *grad* field of every parameter in the tensor. These fields are now the same across all nodes.
- Finally the optimizer step updates the parameters using the newly computed gradients. As all the local models start from the same state and update the weights with the same gradients, the resulting state of the local model is the same for each node.

Below, we describe important details regarding the aforementioned distributed training process.

Initialization The initial setup can be different on different environments. The most important parts are:

- Setting up the address and port of the first node with rank 0:

```
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '12355'
```

- Create the group of nodes with *dist.init_process_group()*, choose the communication backend and set the world size.

Distribution of the dataset The *DistributedSampler* class is used to split the training data across the nodes by passing the world size and the rank of the current node. This class uses the world size to load all the samples that have an index that corresponds to the rank plus t times the world size. By doing this there is no overlap and no information is exchanged between the nodes.

Model wrapping After moving the model to the GPU (using *model.to(device)*), the model has to be wrapped with the DDP class. This effectively replicates the model from node with rank 0 to all other nodes. This is done with $model = nn.parallel.DistributedDataParallel(model)$.

Backpropagation After wrapping the model with the DDP class, the training of the model works as normal: forward pass $outputs = model(images)$ and loss computation $loss = criterion(outputs, labels)$. When calling *backward()* to start the backpropagation, the gradient reduction process of DDP starts, the gradients are computed and synchronized. After this phase is done, when calling *step()* the weights and biases of the local models are updated.

No synchronization DDP offers the interface *no_sync(model)* that stops the gradient synchronization. When inside the no_sync() context, calling the *backward()* method computes the gradients of the local network, without triggering the AllReduce operation. With this interface, we are able to accumulate the gradients and the next iteration outside this context will call the AllReduce again to synchronize the local models.

In chapter 8.2, the code used for testing the algorithms is presented with the description of the methods described above.

3.2.3 Slurm and bash scripting

The Slurm Workload Manager is a job scheduler for Linux used in the Euler Cluster and in the Piz Daint supercomputer. It was used to submit the jobs and to pass the relevant environment configurations. Some relevant commands are: *sinfo* (view information about nodes and partitions), *squeue* (view information about the jobs located in the scheduling queue), and *srun* (run jobs).
Bash scripting can be used to run multiple jobs and to pass the necessary configurations. Here is an example:

```
#!/bin/bash -l

#SBATCH --job-name=cnn                 #job name
#SBATCH --time=00:45:00                #max execution time
#SBATCH --nodes=32                     #number of nodes
#SBATCH --ntasks-per-core=1            #number of tasks per core
#SBATCH --ntasks-per-node=1            #number of tasks per node
#SBATCH --cpus-per-task=12             #number of cpus per tasks
#SBATCH --hint=nomultithread           #don't use extra threads with in-core multi-threading
#SBATCH --constraint=gpu               #only gpu nodes
#SBATCH --account=c24                  #cscs account
#SBATCH --output=o.out                 #output file name
#SBATCH --error=e.err                  #error file name

#load necessary modules for Piz Daint
module load daint-gpu
module load PyTorch

#set up correct communication backend
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
export NCCL_DEBUG=INFO
export NCCL_IB_HCA=ipogif0
export NCCL_IB_CUDA_SUPPORT=1

#run the program using python and by passing the command arguments
srun python main.py -bs 50 -s 2 -e 20
```

4 Project design

In this section, we describe a novel algorithm for training deep neural networks. The algorithm can be seen as an extension of the parallel SGD (P-SGD) algorithm. However, it uses fewer synchronization steps, which reduces the communication cost and improves scalability.

4.1 Baseline Parallel SGD

First, we present variant of SGD, called *parallel stochastic gradient descent* (P-SGD). The most popular implementation of P-SGD is the synchronous P-SGD algorithm, as it is reliable and stable. This algorithm takes advantage of the distributed processing environment in order to speedup the training. In these settings, each node has a copy of the model (DNN) and a partition of the dataset. Every node runs the SGD algorithm, most commonly the mini-batch version. At each iteration, the locally computed gradients are shared and averaged between all nodes and each node can then update its local model using this averaged gradient. By doing this, the algorithm maintains the model consistency across nodes. The pseudocode for this algorithm can be found in Algorithm 3 (the red parts highlights at the synchronization steps).

Algorithm 3 Synchronous Mini-Batch Gradient Descent (P-SGD).

Require: Training Set T; Learning Rate η; Number of epochs E; Batch size B; Number of nodes K
  Initialize parameters θ and synchronize them across all nodes
  Shuffle Training Set T
  Divide Training Set T by the number of nodes K and distribute them across all nodes
  Divide each node's training set into batches of size B ▷ $1 < \mathrm{B} < \frac{|T|}{K}$
  for $i = 1$ to $\frac{|T|}{K \cdot B} \cdot E$ **do**
    Pick batch b ▷ b from the local dataset partition
    $\mathrm{grad}_{node} = \nabla \sum_j l(g(x_j; \theta), y_j)$ ▷ Compute gradient for all $x_j, y_j \in b$
    $\mathrm{grad}_{sync} = \frac{1}{K} \sum_{k=0}^{K-1} \mathrm{grad}_{\mathrm{node}_k}$ ▷ Synchronize gradients and compute the average
    $\theta = \theta - \eta \cdot \mathrm{grad}_{sync}$ ▷ Update the parameters
  end for
  return θ

By sharing and averaging the gradients at each iteration, the algorithm maintains the model consistency across all nodes (see Figure 23). In the next chapter, we propose an alternative algorithm, which does not require gradient synchronization at each iteration.

4.2 Hybrid Synchronous Parallel SGD

The main goal of this project is to propose a novel parallel SGD algorithm that reduces the communication cost required by the baseline P-SGD method while maintaining the generalization properties of the DNN. This is achieved by combining multiple learning strategies. Our algorithm considers training with a prescribed computational budget, defined in terms of a maximum number of epochs. This assumption is often used in a commercial environment, where budget and deadlines often don't admit the possibility to train the networks until the desired performance is achieved. The time or computational cost are bounded and the algorithm should produce relevant results without exceeding these resources.

When dealing with enormous datasets and very large neural networks, parallel learning is the only computational feasible way for solving such big tasks. Scaling to tens or hundreds of nodes makes the communication complexity more relevant. The current state-of-the-art, P-SGD algorithms, use large batch sizes to improve time performance and throughput [44]. Recall Chapter 2.3.3, the ideal choice of the batch size depends on the particular problem at hand, but there is a trend towards using big batch sizes[44], even if this can lead to worse generalization [30]. Some solutions to this phenomenon are proposed. For example, using the P-SGD for the first half of the training phase and letting the nodes train for N iterations before averaging the parameters of the model across all nodes in the second phase [44]. This method, called *post-local SGD*, significantly improves the generalization performance compared to large-batch training. Other solutions involve linear scaling of the learning rate with the number of nodes in order to obtain better generalization [12]. The fact that each dataset and learning task behave differently with different training algorithms also impacts the scalability. The bigger the dataset, the bigger is the possibility to take advantage of more nodes and bigger batch sizes by fine-tuning the hyperparameters of the algorithm.

Our algorithm is designed to exploit the performance gains for large batch sizes while avoiding the generalization gap. To this aim, we split the training (the number of epochs) into three stages.

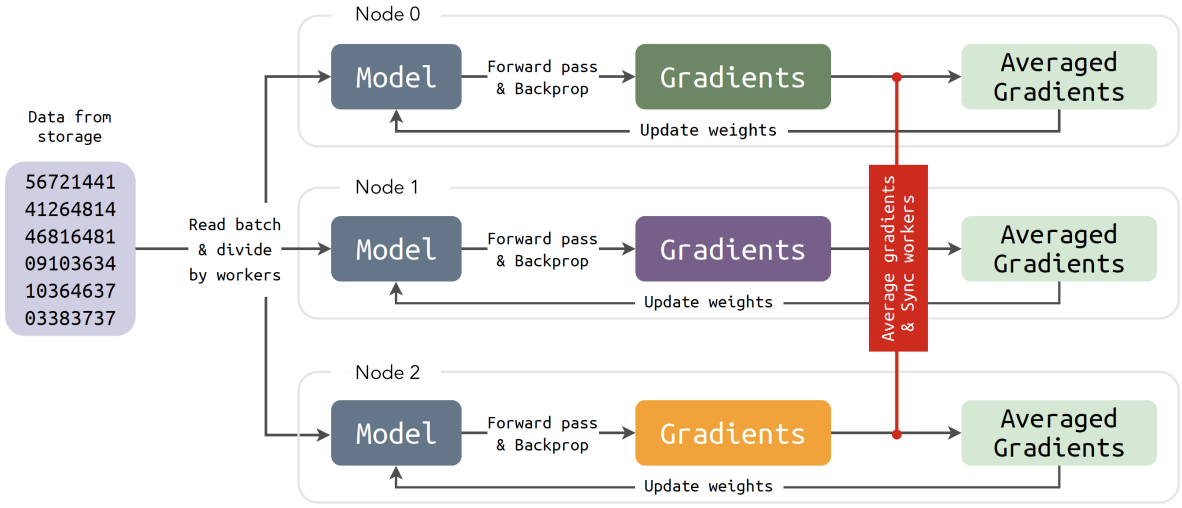


Figure 23. Example of the distributed training using data parallelism (P-SGD). Averaging and synchronization are performed at each iteration, this is the depiction of P-SGD. Image from [12].

4.2.1 First stage

The first stage consists of the *plain* implementation of P-SGD (Figure 23). Here the synchronization between the nodes happens at each iteration (after each mini-batch has been processed). The nodes communicate the gradients with each other, employing the AllReduce API. In our implementation, this is achieved by using the DistributedDataParallel module that implements this communication step using the *buckets* system (Chapter 3.2.2), which makes the communication between the nodes more efficient, compared to a naive implementation. Note, the models are synchronized across all nodes.

The goal of this stage is to provide the model with a good search *direction* towards the minimum of the loss function. The batch size that is given as input is effectively multiplied and scaled by the number of nodes, i.e., *batch_size*K*. During this phase, the learning algorithm is able to make use of all the potential of the DDP module.

4.2.2 Second stage

In the second stage, we take advantage of the fact that bigger batch sizes are able to estimate the overall gradient more precisely. For this reason, we accumulate gradients by delaying the synchronization step and the weight updating step. We allow the AllReduce to take place only on every W-th iteration. Thus, we are effectively using a batch size of $B \times K \times W$, where B is the original batch size and K denotes number of nodes.

With this part of the algorithm, we are able to reduce the communication cost between the nodes by a factor of W, while benefiting from the use of the bigger batch size. This stage improves the network capability to fit the data, in turn increasing the risk of overfitting. As shown in Figure 13, SGD has a lot of fluctuation as it progresses based on one sample at a time. When using mini-batch SGD we eliminate some of this fluctuation by reducing the error in gradient estimation. The same is valid when increasing the size, indeed the fluctuation is attenuated even more for larger mini-batches.

4.2.3 Third stage

The result of the first and second stages is a model that is trained using two different, increasing, batch sizes. These two stages can be seen as a *pre-training* for the local models. In this last stage, we use another strategy for training the model that we call *influenced synchronous* training. This stage is of particular importance in order to avoid overfitting.

This stage is inspired by *decentralized asynchronous* training [24], where each node can explore the landscape of the loss function by having a model that is different from the others. The re-parametrization happens asynchronously

on parameters that are not up-to-date (or simply different from the ones that will be picked by the next node). In our case, the models will diverge from each other. Moreover, they never exchange the parameters of the model, only the gradients. That means that the models will continue to diverge until the end, sharing only the gradients at some fixed interval R. The gradients shared at this step are only the gradients of the batch of this specific iteration.

This strategy is synchronous, as the gradient synchronization happens at the same time for each node by using the same DDP implementation as for the first two stages. The difference is that we perform the update step after each iteration without local gradient accumulation. In Figure 24, we can see each node performing the training locally by updating the local parameters, but without sharing any information with other nodes. Consequently, the models diverge from each other over time. In Figure 23, we depict the iteration where the gradients are shared before each update step.

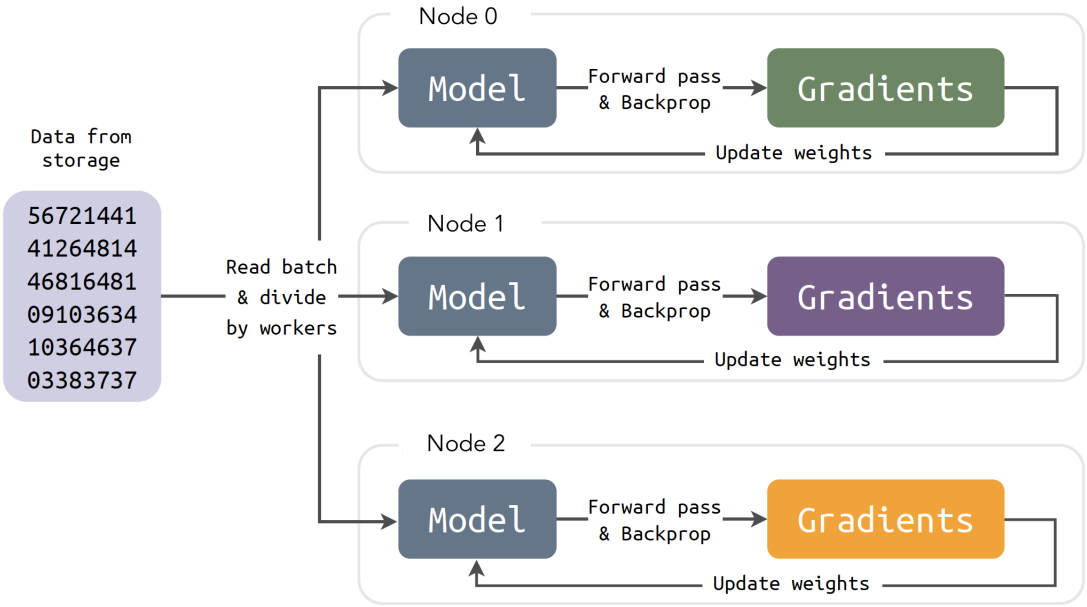


Figure 24. An example of distributed training where each node evolves their own model using their partition of the dataset. This depicts the behaviour of our algorithm in the third stage, in between the synchronization iterations. Image from [12].

We say that the algorithm at this stage is *influenced*, in the sense that each model is trained independently but with the *influence* of the others. That is because sharing the gradients every R iterations changes the direction of the iteration path and the new direction is equal for every node (even though they are on different parts of the loss landscape). As models associated with different nodes take different steps on the landscape (e.g. for one node iteration path see Figure 13), after R iterations, a step is taken using the same search direction for that specific iteration. This is how they *influence* each other by exchanging the information regarding the search direction they want to take (essentially where is their minimum). By averaging these directions, we obtain a new global search direction that uses the information gathered by all nodes. This type of training is able to explore the loss landscape to increase the representation capacity of the network, but also potentially to avoid local minima (and improve generalization). As each node is updating the model freely based on their partition of the dataset, this partition should be sufficiently large. This imposes an upper limit on the number of nodes that can be used for a given dataset, based on its size.

There is also another motivation for sharing only the gradients instead of all the parameters of the model. By sharing all the weights and biases, the algorithm has to first compute the gradients and update the local model. After this, it can send the local parameters to the other nodes, compute the average, and update its own local model a second time. By sharing only the gradients, the whole process requires fewer steps. The nodes compute the gradients and directly share them with the others as they become available. The nodes take the averaged gradients and update the network. This is computationally less expensive as it requires only one optimization step. At the same time, we are also able to exploit the efficient implementation of DDP module.

At the end of the training process, the models are shared and averaged. This way, the new *knowledge* acquired by each node on its own partition of the dataset is shared. The algorithm is demonstrated in Figures 25 and 26.

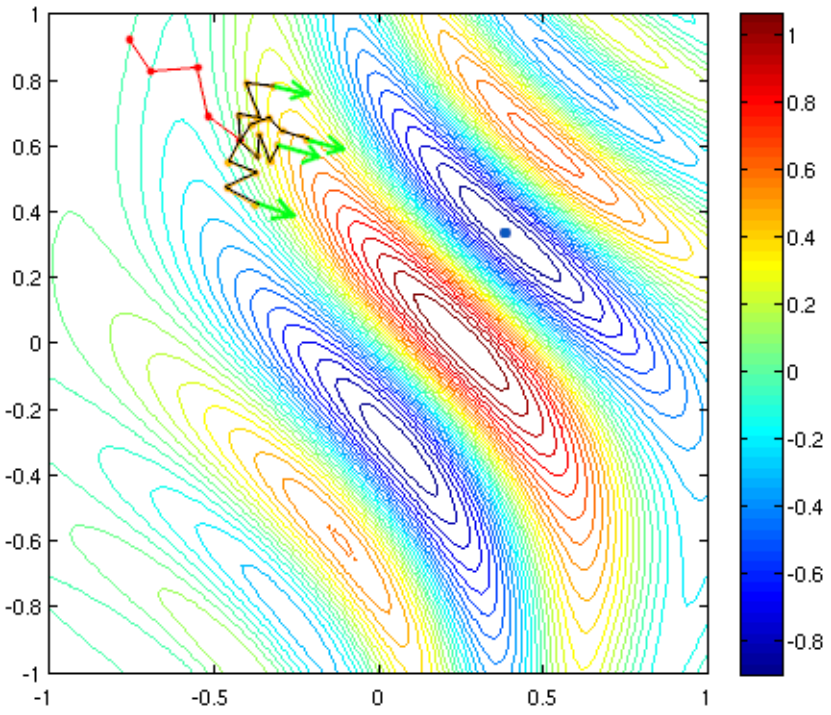


Figure 25. Example of an iteration path generated by our algorithm. The red line represents the sequence of iterates generated by the algorithm during the first two stages where the model is perfectly replicated on each node. The black lines represent the third stage where the nodes update parameters locally, without any synchronization for R iterations. We see that the sequence of iterates generated on each node differs (*exploration phase*). The green line represents the shared direction for the next iteration.

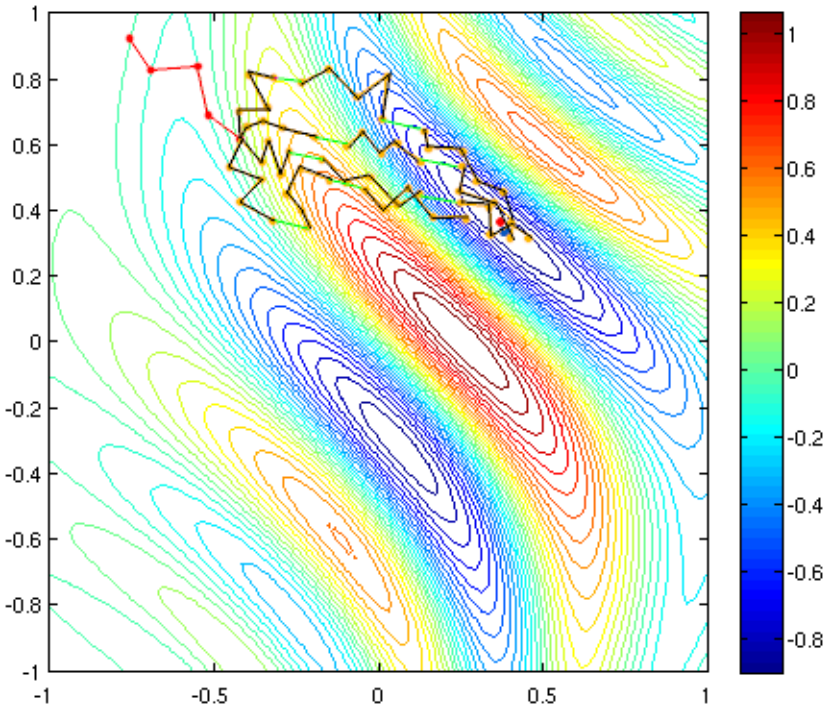


Figure 26. Example of an iteration path example by our algorithm. The black lines represent the third stage and the green lines are the shared directions. As the batch size is smaller compared to the first two parts, there is more oscillation. The algorithm chooses the direction *influenced* by the majority of the nodes. They converge to the global minimum at the blue point (0.4,0.33). The red dot near the minimum is the final state of the model after the averaging of all the parameters.

4.3 Algorithm

Here we present our *hybrid algorithm for distributed data parallel training using mini-batch SGD*. The three parts are clearly visible and immediately after, there is the final parameter synchronization. With this algorithm the number of communication steps goes from $\frac{|T|}{K \cdot B} \cdot E$ for P-SGD down to $(\frac{|T|}{K \cdot B} \cdot \frac{E}{3}) + (\frac{|T|}{K \cdot B \cdot W} \cdot \frac{E}{3}) + (\frac{|T|}{K \cdot B \cdot R} \cdot \frac{E}{3})$.

Algorithm 4 Hybrid Parallel Mini-Batch Gradient Descent for neural networks.

Require: Training Set T; Learning Rate η; Epochs E; Batch size B; Number of nodes K;
Require: Gradient accumulation parameter W; Interval for synching in third stage R
Initialize parameters θ and synchronize them across the nodes
Shuffle Training Set T
Divide Training Set T by the number of nodes K and distribute them across all nodes
Divide each node's training set into batches of size B ▷ $1 < B < \frac{|T|}{K}$
for $i = 1$ to E **do** ▷ This loop runs on every nodes
  if $i <= \frac{E}{3}$ **then** ▷ First stage
    for $q = 1$ to $\frac{|T|}{K \cdot B}$ **do**
      Pick batch b and do the *forward pass*
      $\text{grad}_{node} = \nabla \sum_j l(g(x_j; \theta), y_j)$ ▷ *Backpropagation* for all $x_j, y_j \in b$
      $\text{grad}_{sync} = \frac{1}{K} \sum_{k=0}^{K-1} \text{grad}_{\text{node}_k}$ ▷ Synchronize gradients and compute the average
      $\theta = \theta - \eta \cdot \text{grad}_{sync}$ ▷ Update the parameters
    end for
  end if
  if $\frac{E}{3} < i <= \frac{2 \times E}{3}$ **then** ▷ Second stage
    for $q = 1$ to $\frac{|T|}{K \cdot B}$ **do**
      if q %$W == 0$ **then**
        Pick batch b and do the *forward pass*
        $\text{grad}_{node} = \text{grad}_{node} + \nabla \sum_j l(g(x_j; \theta), y_j)$ ▷ *Backpropagation* for all $x_j, y_j \in b$
        $\text{grad}_{sync} = \frac{1}{K} \sum_{k=0}^{K-1} \text{grad}_{\text{node}_k}$ ▷ Synchronize and average the accumulated gradients
        $\theta = \theta - \eta \cdot \text{grad}_{sync}$ ▷ Update the parameters
      end if
      if q % W ! $= 0$ **then**
        Pick batch b and do the *forward pass*
        $\text{grad}_{node} = \text{grad}_{node} + \nabla \sum_j l(g(x_j; \theta), y_j)$ ▷ *Backprop* and grad accumulation for all $x_j, y_j \in b$
      end if
    end for
  end if
  if $i >= \frac{2 \times E}{3}$ **then** ▷ Third stage
    for $q = 1$ to $\frac{|T|}{K \cdot B}$ **do**
      if q % $R == 0$ **then**
        Pick batch b and do the *forward pass*
        $\text{grad}_{node} = \nabla \sum_j l(g(x_j; \theta), y_j)$ ▷ *Backpropagation* for all $x_j, y_j \in b$
        $\text{grad}_{sync} = \frac{1}{K} \sum_{k=0}^{K-1} \text{grad}_{\text{node}_k}$ ▷ Synchronize gradients and compute the average
        $\theta = \theta - \eta \cdot grad_{sync}$ ▷ Update the parameters
      end if
      if q % R ! $= 0$ **then**
        Pick batch b and do the *forward pass*
        $\text{grad}_{node} = \nabla \sum_j l(g(x_j; \theta), y_j)$ ▷ *Backpropagation* for all $x_j, y_j \in b$
        $\theta = \theta - \eta \cdot \text{grad}_{node}$ ▷ Update the parameters
      end if
    end for
  end if
  Synchronize and average all local models (using the AllReduce API) ▷ Last step of the algorithm
end for

5 Numerical Experiments

5.1 Model problem

5.1.1 MNIST

In this work, we investigate the performance of our hybrid synchronous P-SGD algorithm on classification task with the *MNIST database*. MNIST is made of 60,000 examples of handwritten digits images [26]. This database is widely used in machine learning. It was created in 1998 by "re-mixing" and normalizing the black and white images from NIST's original datasets. The resulting images contain grey levels as a result of using an anti-aliasing technique. Afterwards, the images were centered in a 28x28 image (Figure 27).

Figure 27. Example of images from MNIST test dataset. Image from [45].

5.1.2 Network architecture

The network used for testing the algorithms is a simple convolutional neural network that takes a 28×28 image as an input (Figure 28). The image is then convolved with 16 kernels of size 5×5 and stride 1, passed through a ReLU layer and a max-pool layer with kernel size 2×2 and stride 2. The resulting 16 images of size 12×12 are then again convolved with 16 kernels of size 5×5 and stride 1, passed through a ReLU layer and a max-pool layer with kernel size 2×2 and stride 2. After the feature extraction phase, the feature maps are passed through a fully connected layer after being flattened. The output size is 10, one for each of the 10 possible digits.

5.1.3 Minimization problem

The loss function used for the numerical experiments is the *cross entropy loss*, defines as

$$l(y_{true_i}, y_{predicted_i}) = -\sum_i y_{true_i} log(\phi(y_{predicted_i})),$$

with $\phi(y_i) = \frac{e^{y_i}}{\sum_k e^{y_k}}$, see Section 2.1.1. This loss function computes the cross entropy between output and the label and it is useful for training classification problems with C possible output classes. For our experiments, the minimization problem (1) becomes:

$$\min_\theta L := -\sum_{i=1}^{N}\sum_{c=1}^{C} y_{\text{true}_i} log(\frac{e^{g(x_i;\theta)_c}}{\sum_{k=1}^{C} e^{g(x_i;\theta)_k}}), \tag{2}$$

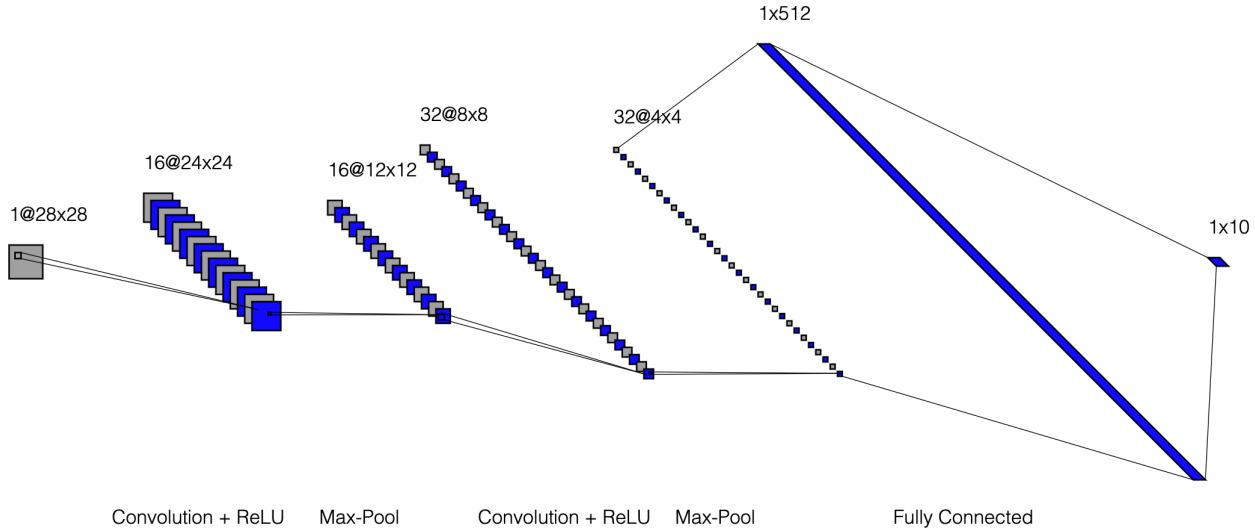


Figure 28. Convolutional neural network representation.

where N is the number of data samples, y_{true_i} is the correct probability (0 or 1) for a given image i and class c, and $g(x_i;\theta)_c$ is the value corresponding to the class c for the output of the model for image x_i. The *CrossEntropyLoss()* function takes the unnormalized output of the network over the batch and the labels (in our case a tensor containing the correct number depicted in the image, for each image in the batch).

5.2 Numerical Experiments

In this chapter we present the experiments, report numerical results, and show the capability of our algorithm.

5.2.1 Description

The goal is to design an algorithm that is able to reduce the communication overhead by keeping the same accuracy on the test data. In order to achieve this, the testing is conducted in two phases.

The metrics we use for the analysis of the performance are *test accuracy*, *time* and the number of *communication steps*. *Accuracy* is measured by using the test set of MNIST and it is given by the percentage of images that the model is able to classify correctly. *Time* is measured only for the training phase, the setup and final testing are excluded. In the training phase, the time we are measuring consists of the I/O performance (loading the batch), cache performance, bus and network speed between the nodes, and computing time. The time is measured in the node with rank 0 and reported in seconds. Each test is performed three times and the average is reported. Lastly, we focus on the number of *communication steps*, as our aim is to reduce it.

First phase of testing In the first phase, we test different communication strategies with different hyperparameters. The goal of this phase is to test our first ideas and learn more about the behaviour of the optimization problem at hand. These tests are performed first on the *Euler cluster*, where the network and structure of the implementation are tested. During these tests, a problem with our first implementation emerged, as it discards the accumulated gradients and only synchronizes the last computed gradient. For this reason, we do not report in detail the obtained results. Instead, we summarize what we learned. Subsequent tests are run on the *Piz Daint* supercomputer, where we test the hyperparameters that are used in the second phase.

Second phase of testing In this phase we perform systematic testing using the information gathered in the first phase. By measuring the performance of different synchronization strategies, we are able to get the desired results.

5.2.2 Numerical Results

Serial test The first tests are performed with the serial implementation of mini-batch gradient descent, see Algorithm 2. In Table 1 the results are shown. We can observe that increasing the batch size reduces the training time, and in this case, it also increases generalization. We consider the obtained results as a baseline, in terms of test accuracy. Therefore, we aim to train with parallelization, models which generalize as well as one obtained using serial training.

Batch size	Accuracy(%)	Time(s)
50	97.41	67.56
150	98.87	35.78
250	98.91	32.47

Table 1. Test accuracy and training time for serial mini-batch gradient descent with respect to the batch size.

First phase The testing of the first versions of the implementations focuses on the selection of appropriate number of nodes and on understanding which communication strategies result in models with the highest test accuracy. The initial testing is performed with 4, 8, 16 and 32 nodes in order to understand the behaviour of the algorithm with increasing parallelism. This decision is based on the fact that our network is relatively small and the MNIST dataset is also small. The other tested variable is the batch size. We opt for three sizes, i.e., *50, 150, 250*, partially based on findings reported in the literature [30]. We also use the input batch size without scaling it down for the number of nodes. We limit the scope of these tests in order to keep under control the complexity of this project. The learning rate is kept fixed during this phase, with a value of $\eta = 0.1$. The maximum number of epochs is also fixed at 20, as preliminary tests indicate that it is sufficient in order to train the model properly, without overtraining.

The other component to test is the communication strategy itself. We first test the normal P-SGD, recall Algorithm 3. Then, we move into testing more complex communication strategies. Here are the findings of this phase:

- Our implementation of the P-SGD algorithm is able to achieve an accuracy higher than 98% in the majority of the tests. This algorithm is our baseline for this phase. The obtained results show signs of accuracy degradation for 32 nodes and a batch size of 250. Since the bigger number of nodes has an impact on the generalization capability of the model, the accuracy dropped to 95.3% for this case. We also observe that when training with fewer nodes, the best performance is achieved when using a bigger batch size (with $K = 4$, the best accuracy (98.83%) is achieved with batch size 250).

- The times for the training with normal P-SGD are smaller for training with the bigger batch size available, i.e., 250. This is expected as increasing the batch size increases throughput and thus lowers the training time. In Table 2 the best (lowest) training times are reported for each node, and which batch size was used to obtain the best time. As we can observe, the time increases after going from 8 to 16 nodes and even more from 16 to 32. Thus, the obtained results suggest that for our classification MNIST problem, going above 16 nodes is not beneficial and the benefit of parallelization is lost. When training with $K = 32$ nodes, the lowest time is achieved with a smaller batch size (150). The reason for this behaviour is not clear. Probably, if the setup of the *dataloader* is not optimal when loading bigger mini-batch sizes, more computing resources are required than necessary. Also, if we look at Table 1, for a mini-batch size of 250, P-SGD on 8 nodes is only 4 times faster than serial mini-batch gradient descent. We will discuss more in-depth on the dataloader setup in the second phase.

Number of nodes	Time(s)	Batch size
4	12.54	250
8	8.21	250
16	9.80	250
32	15.88	150

Table 2. The best execution times for P-SGD algorithm with respect to the number of nodes and which batch size is used (time measured in the node with rank 0).

- As a next step, we investigate the impact of the gradient synchronization frequency on the test accuracy. As mentioned above, the implementation for these tests is not correct but we want to include some findings here as we are able to see the degradation of the accuracy when training on a larger number of nodes. For example, the best accuracy for the algorithm that shares the gradient computed at each 10th iteration and discards the rest can be found in Table 3. The accuracy starts to decrease compared to P-SGD as we effectively train on fewer data by dropping the gradient in between the synchronization steps. The number of epochs is still sufficiently large to achieve 95%, or more. When training with 4 nodes, the best test accuracy is achieved with a batch size of 150. This can be explained by the fact that with a mini-batch size of 50, when using 4 nodes, the effective mini-batch size is $50 \cdot 4 = 200$. Using a bigger mini-batch size as input, the algorithm performs better on the MNIST dataset.

Number of nodes	Accuracy(%)	Batch size
4	97.34	150
8	97.09	50
16	97.75	50
32	95.09	50

Table 3. Best test accuracy with respect to the number of nodes and batch size, for the algorithm that shares the last gradient computed at each 10th iteration, while discarding the other computed gradients.

- Next, we test algorithm which divides the training into three parts:

  1. Gradient synchronization at each step (P-SGD).
  2. Synchronization of the gradients at the W step, discard in between gradients
  3. Synchronization of the gradients at the R step, discard in between gradients (with $R > W$)

  This structure is not intentional, as the implementation discards the gradients in the second and third parts. The results here are not consistent but promising. For some strategies (different values for R and W) the best accuracy of above 98% is consistently achieved for a number of nodes equal to 4 and 8, with degradation occurring afterwards. The best results in terms of test accuracy are achieved with $W = 4$ and $R = 8$, and it effectively lowers the number of communication steps by 55% with respect to P-SGD, Algorithm 3.

- The times for all the tests from this first phase are surprisingly consistent across all communication strategies (P-SGD, gradient synchronization at different intervals, and the three-part algorithm with different values for R and W). In Table 4, we report the intervals where the tests fall in terms of time. From these results, it seems that the various communication strategies have no impact on the training time. This is a consequence of having a small network and thus the communication overhead is really small, and it is also the result of not tuning correctly the parameters of the *dataloader*. This second cause is addressed in the second phase.

Number of nodes	Time(s)
4	12-14
8	8-10
16	9-11
32	14-16

Table 4. Interval of the best times for all the communication strategies of the first phase (time measured in the node with rank 0).

Second phase After the first phase, we fix the implementation problems and we investigate more complex algorithms and also why there seems to be no speedup in time when changing communication strategies and when increasing the number of nodes from 8 to 16. We also incorporated *Nestrov momentum* to enhance the convergence of the proposed algorithm. We also try to scale the learning rate proportionally to the number of nodes and batch sizes. This however has a negative impact on the accuracy of the models. For this reason, we discard the use of scaling factors for the learning rate.

Another hyperparameter that we test is the *num_workers* argument of the *dataloader* class, the data loading utility of PyTorch used to load the images into the memory of each node. The *num_workers* is used for multi-processing applications and sets how many subprocesses are used to perform the data loading. Every time an iterator of a DataLoader is created, these worker processes are used. For example, in our implementation, when calling *enumerate(train_loader)*, *num_workers* loading processes are used to load the images. To maximize the efficiency of our algorithm, we test different configurations of num_workers, for a specific number of nodes and batch sizes.

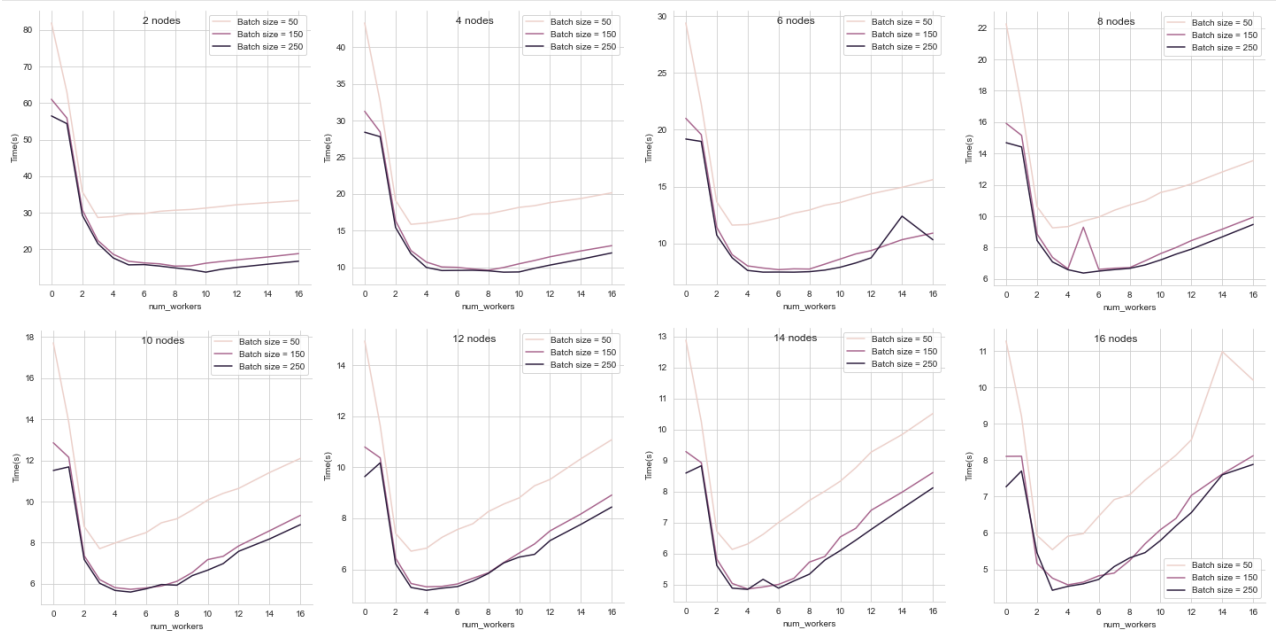


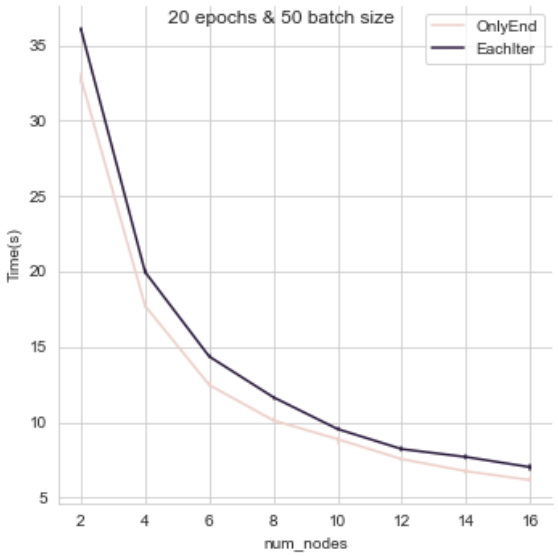
Figure 29. The training time with respect to a varying number of workers for a different number of nodes and batch sizes.

In Figure 29, we can see that, for each combination of batch size and number of the nodes, there is a num_workers that minimizes the training time. This explains the times we got from the testing in the first phase, recall Tables 2 and 4, as we are using the number of nodes as num_workers in that implementation. From the results in Figure 29, it is clear that choosing this argument correctly for the dataloader can have a huge impact on the training time. These findings are incorporated in our final implementation.

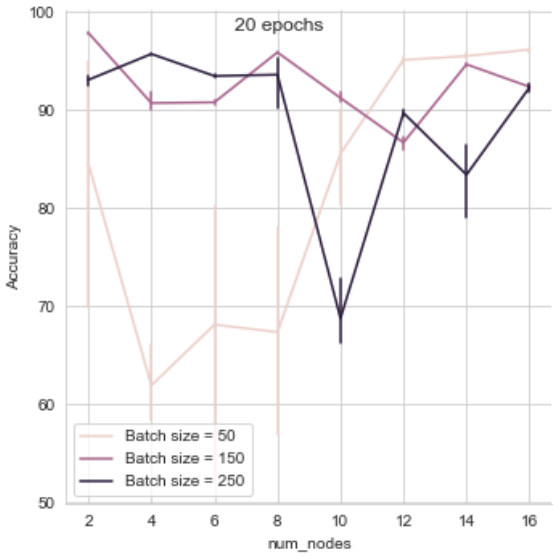
The next tests are used for tuning the hyperparameters of the implementation of our algorithm to achieve the best possible performance. We first test the test accuracy of our algorithm with different mini-batch sizes, i.e., 32, 48 and 64 (see Figure 31). As we can see, the best batch size for our algorithm is 64. We also limited the maximum number of nodes to 16, as beyond this number, the time and accuracy deteriorate. We show in Figure 30 (a) the time performance of the *EachIter* algorithm (we use *EachIter* to refer to the implementation of normal P-SGD), and of the algorithm that lets the local models train individually and only averages the parameters at the end of the training (*OnlyEnd*). This plot shows the communication overhead of synchronizing at each step. As we can see, the difference is small but present. Next, we study the accuracy obtained by the *OnlyEnd* method. Figure 30 (b) demonstrates that the method prefers bigger batch sizes, but nevertheless it is not able to achieve high accuracy consistently. For this reason, our algorithm employs a hybrid technique for training.

Our algorithm The hyperparameters of our algorithm are $W = 8$ and $R = 12$, see Algorithm 4. These hyperparameters were selected as the best during the hyperparameter search. Using these particular set of hyperparameters, we are able to train our model by using less than half the communication steps. In fact, the number of communication steps drops to 40% of the steps required by the standard P-SGD. Figure 31 shows the accuracy and time performance of our algorithm. In Figure 31 (a), it is possible to see the important differences between batch sizes when it comes to accuracy. Decreasing the batch size impacts the model's generalization capability. As we can see, a batch size of 64 is the best, as it consistently maintains the accuracy above 98%, up to 14 nodes. In Figure 31 (b) the execution times are shown. Increasing the batch size increases throughput and thus lowers the training time. From the Figure 32, we can observe that the parallel efficiency is decreasing as the amount of nodes increases. This behaviour is expected, as the network and dataset under consideration are fairly small.

In Figure 33 we illustrate more in-depth details about our hybrid synchronous P-SGD algorithm. In Figure 33 (a) we observe that the ideal choice for num_workers decreases when increasing the number of nodes. This plot also shows again how the speedup doesn't scale linearly. In Figure 33 (b) the accuracy against the epochs is plotted. The algorithm starts with good accuracy and grows constantly towards 99%. This growth is not smooth and it gets slower when increasing the number of nodes. At epochs 7 and 15 we have the change of stage in the hybrid synchronous P-SGD algorithm. Bigger sets of nodes show more pronounced jumps when the new stage has started. With $K = 14$ at epoch 15 there is a pronounced dip in accuracy, but immediately after jumps back higher than before.

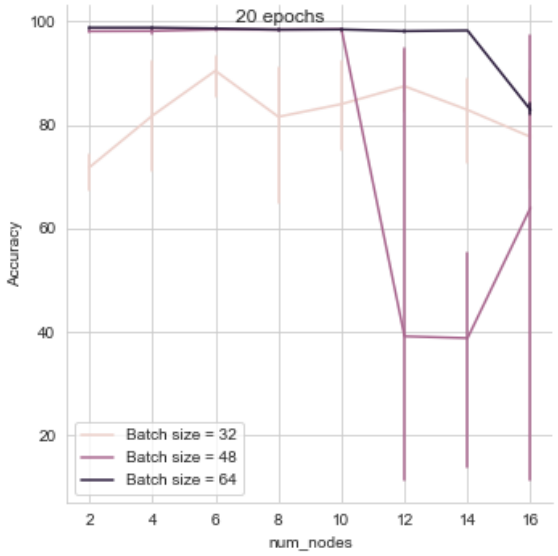


(a) The execution times for the *OnlyEnd* and *EachIter* algorithms.

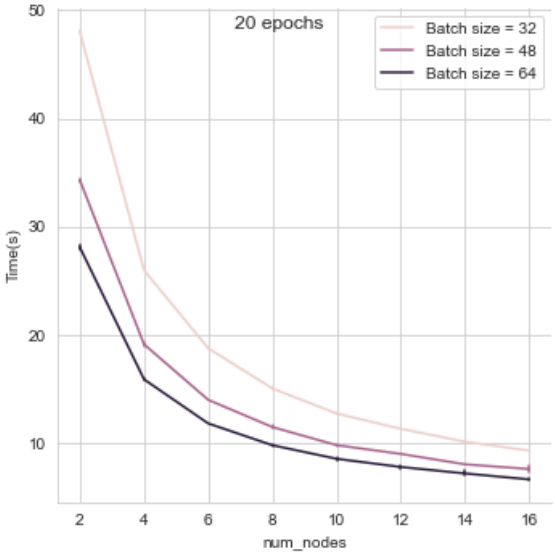


(b) The accuracy with respect to different batch sizes of the *OnlyEnd* algorithm.

Figure 30. Computational cost and test accuracy with respect to increasing the number of nodes for OnlyEach and EachIter algorithms.



(a) The accuracy against the number of nodes. The three lines are the average of three tests for a given batch size. Vertical lines represent the result of the three tests.



(b) The time performance for different batch sizes. The three lines are the average of three tests for a given batch size. Vertical lines represent the result of the three tests.

Figure 31. Accuracy and time performance of our hybrid synchronous P-SGD algorithm.

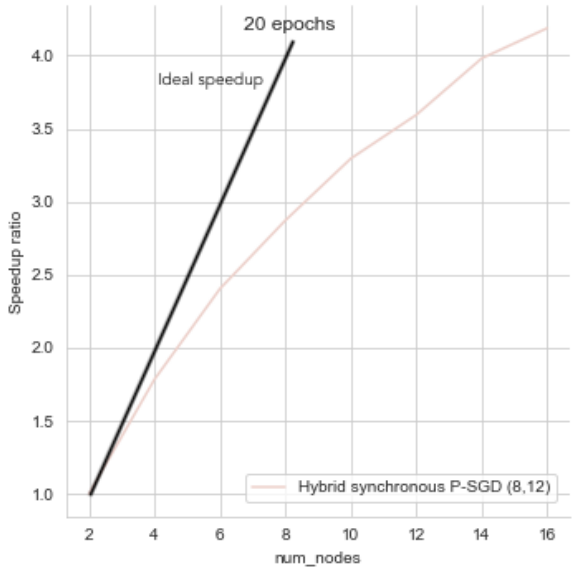
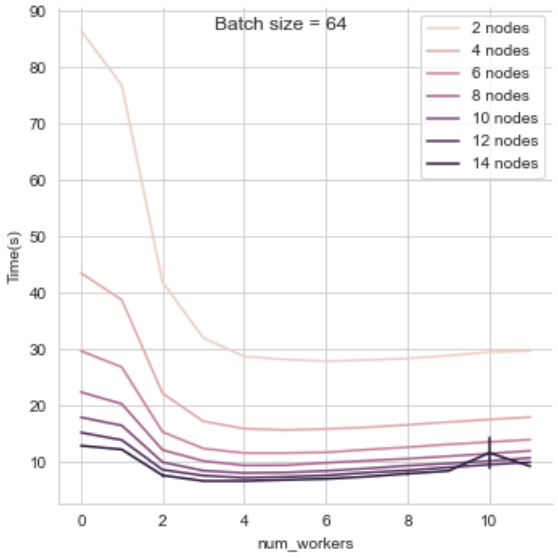
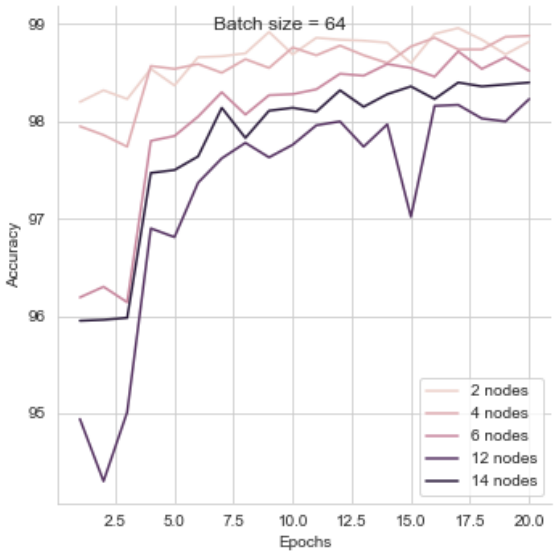


Figure 32. Speedup of our hybrid synchronous P-SGD algorithm with respect to time.

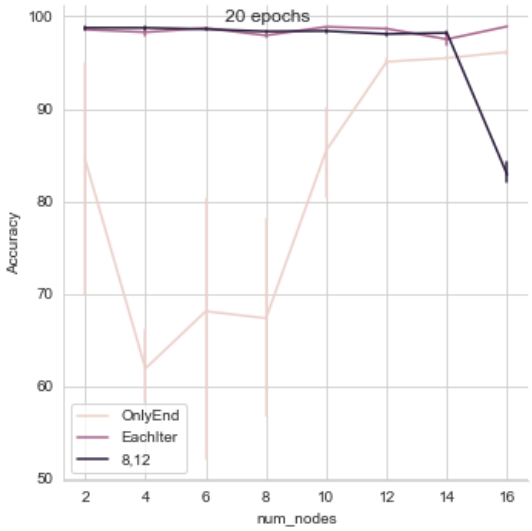


(a) The time of our hybrid synchronous P-SGD algorithm with respect to the *num_workers* argument.

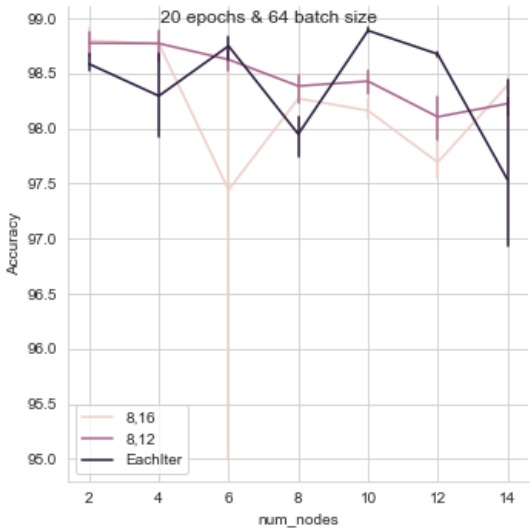


(b) The accuracy performance at each epoch of our hybrid synchronous P-SGD algorithm.

Figure 33. Impact of the *num_workers* argument and the convergence history of our hybrid synchronous P-SGD algorithm.



(a) The accuracy for *OnlyEnd*, *EachIter* and our hybrid synchronous P-SGD algorithm (*8,12*).



(b) The accuracy of our hybrid synchronous P-SGD algorithm (*8,12*) with another version of our algorithm (*8,16*) and *EachIter* with respect to the increasing number of nodes.

Figure 34. Test accuracy comparisons of our hybrid synchronous P-SGD algorithm (here called (*8,12*), as for the parameters W and R).

Multiple choices for W and R are tested, but the best result in terms of accuracy comes from the choice of $W = 8$ and $R = 12$. We also run other tests where the first stage is longer, or where the second stage is longer. The generalization on these tests is consistently lower than our hybrid synchronous P-SGD algorithm because of overfitting. In Figure 34 we show other comparisons with our algorithm with respect to test accuracy. In Figure 34 (a) we can see our algorithm compared to the two extremes of the strategy spectrum, namely OnlyEnd (where the local models are trained separately and averaged at the end) and EachIter (normal P-SGD). Up to 14 nodes we are able to match the best performance by EachIter. Our algorithm also trains 2-15% faster, but the difference decreases with the increase in the number of nodes. In Figure 34 (b) we compare the hybrid synchronous P-SGD algorithm, with hyperparameters $W = 8$ and $R = 12$, and the EachIter algorithm, with another version of our algorithm with different hyperparameters ($W = 8$, $R = 16$). Our algorithm ($W = 8$ and $R = 12$) is able to stay consistently above the (*8,16*) strategy in terms of accuracy.. It also outperforms EachIter, with respect to test accuracy, up to 8 nodes, while using 60% fewer communication steps.

6 Conclusion

In this work, we proposed a novel hybrid synchronous P-SGD algorithm, that is able to reduce the communication steps of the standard P-SGD algorithm by 60% while retaining good generalization properties of the model. The proposed algorithm is designed to run for a fixed computing budget (user-specified number of epochs) and utilizes a hybrid structure that is synchronous during communication. The algorithm has three stages. In the first stage, it uses the normal P-SGD for training the network. In the second stage, it uses P-SGD with gradient accumulation. In the last stage, the local models are trained independently and they share, at a fixed interval, the gradients computed on the last local batch. The gradients are then averaged and used by all nodes to update their local models. By doing so, the nodes *influence* each other by sharing a global search direction. After the last stage, the local models are averaged. This structure enables the use of two different, increasing, batch sizes that provide a *pre-training* for the local networks (in the first two stages). The last stage is used to avoid overfitting and further reduce the communication steps. The obtained results provide a promising starting point for future research into hybrid distributed SGD algorithms with a focus on reducing communication. Here, we summarize the conclusions and findings of this project:

- Reduced communication is desirable since communication overhead can be the performance bottleneck in distributed systems when dealing with very big networks and datasets. Reducing the communication cost should not deteriorate the generalization properties of the network. The use of the hybrid approach proposed in this work allows for decreasing the communication overhead by increasing the complexity of the algorithm. The proposed algorithm with increased complexity manages to retain the generalization ability of the model and avoids overfitting.
- The proposed algorithm could enable training for more epochs while keeping the communication cost low and could produce models with even better accuracy.
- The success of any learning algorithm is task-specific. It depends on the task that we are about to solve, i.e., model and the dataset, and their size. These factors determine the best hyperparameters to use and fine-tuning the algorithm is fundamental for achieving the best results. In our case, the characteristics of the task also determine the best parameters for the synchronization strategy. Each task has a limited possible speedup given by the size of the dataset and the network. If the number of nodes becomes too big, the size of the local dataset becomes too small and the network has to use really small mini-batch sizes. This could introduce noise and lead to worse convergence.
- The model and dataset used for this project are too small for more than 16 nodes. The communication overhead is really small as the number of network parameters is small. It is anyway possible to see consistent improvement in training time when using our hybrid P-SGD algorithm, with respect to the standard P-SGD.
- The batch size plays a fundamental role in distributed training. When training with few nodes, the batch size can increase and speed up computations. If the number of nodes increases, we should decrease the batch size. Using a combination of two different effective batch sizes as in our algorithm, allows the model benefit from training on large batch sizes while mitigating the risk of overfitting by using a smaller mini-bath size in the first stage.
- Scaling the learning rate with respect to the number of nodes didn't improve the performance of our algorithm.
- The *num_workers* argument of the dataloader has a big impact on the execution time of the algorithms. The number of processes that are used to load the images can speed up computations by providing the perfect amount of throughput to the network. It can also slowly decrease the performance by consuming resources needed for the training. The best number of workers depends on the dataset and hardware. The size of the images, the number of images to load and the available resources of the hardware can shift the optimal number of workers, and impact the training time.
- When accumulating the gradients, the accumulation should be performed for a significant amount of iterations (in our case 8). Lower numbers don't seem to enable the mechanism that helps the training on different sets of nodes to be consistent. For this reason, our algorithm works really well for 2 nodes as well as for 14. By accumulating for a longer period, the two effective batch sizes are far apart from each other and we exploit the benefits of training using large batch sizes.

6.1 Future Work

This project shows that there is space for hybrid learning techniques in distributed training with the SGD method. In the future, we plan to:

- Investigate how the parameters of the algorithm change when increasing the size of the dataset and the model at the same time as the number of nodes (weak scalability).
- Investigate strong scalability by using a bigger problem. Using this project as a starting point, the behaviour of our algorithm should scale up to bigger tasks. The decrease in training time should also become more apparent.
- Investigate better communication strategies for increasing the speedup by scaling the learning rate when adding more nodes.
- Investigating the impact of the *num_workers* argument when scaling up the problem and number of nodes.

7 Acknowledgements

I thank Prof. Rolf Krause for the opportunity to work with him and I also thank Dr. Alena Kopaničáková for all the help given throughout this project. I thank the Swiss National Supercomputing Centre (CSCS) for giving me access to Piz Daint, one of the most powerful supercomputers in the world. I would also thank the Università della Svizzera Italiana (USI) for lighting up again my long-lost passion for computer science.

8 Appendix

8.1 Euler Cluster

Hardware full specification

LOGIN NODE	
Nodes	icslogin01
CPU	2 x Intel Xeon E5-2650 v3 @2.30GHz, 20 (2 x 10) cores
RAM	64GB DDR4 @2133MHz
HDD	1 x 1TB SATA 6Gb
INFINIBAND ADAPTER	Intel 40Gbps QDR

COMPUTE NODE - Fat nodes	
Nodes	icsnode[01-04,07]
CPU	2 x Intel Xeon E5-2650 v3 @2.30GHz, 20 (2 x 10) cores
RAM	128GB DDR4 @2133MHz Note: icsnode07 is populated by 512GB RAM
HDD	1 x 1TB SATA 6Gb
INFINIBAND ADAPTER	Intel 40Gbps QDR

COMPUTE NODE - GPU nodes	
Nodes	icsnode[05,06,08-16]
CPU	2 x Intel Xeon E5-2650 v3 @2.30GHz, 20 (2 x 10) cores
RAM	128GB DDR4 @2133MHz Note: icsnode15 is populated by 512GB RAM
HDD	1 x 1TB SATA 6Gb
INFINIBAND ADAPTER	Intel 40Gbps QDR
GPU	**icsnode[05-06]:** 1 x NVIDIA A100-PCIe-40GB Tensor Core GPU 40GB HBM2 **icsnode[08,13]:** 2 x NVIDIA GeForce GTX 1080 Ti Titan 11GB GDDR5X 3584 CUDA cores **icsnode[09-12,14-16]:** 1 x NVIDIA GeForce GTX 1080 Founders Edition 8GB GDDR5X 2560 CUDA cores

COMPUTE NODE - Regular nodes	
Nodes	icsnode[17-39]
CPU	2 x Intel Xeon E5-2650 v3 @2.30GHz, 20 (2 x 10) cores
RAM	64GB DDR4 @2133MHz
HDD	1 x 1TB SATA 6Gb
INFINIBAND ADAPTER	Intel 40Gbps QDR

COMPUTE NODE - Multi GPU nodes	
Nodes	icsnode[41-42]
CPU	2 x Intel Xeon Silver 4114 CPU @2.20GHz, 20 (2 x 10) cores
RAM	100GB DDR4 @2666MHz
HDD	1 x 1TB SATA 6Gb
INFINIBAND ADAPTER	Intel 40Gbps QDR
GPU	**icsnode41:** 2 x NVIDIA GeForce GTX 1080 Ti Titan 11GB GDDR5X 3584 CUDA cores **icsnode42:** 2 x NVIDIA GeForce GTX 2080 Ti Titan 11GB GDDR6 4352 CUDA cores

Storage specification

24-bay storage with RAID controller
16-bay JBOD storage
16-Gb Fibre Channel controller

8.2 Code

```
############## Import all necessary packages ##############
import os
import torch
import subprocess
import torch.distributed as dist
import torch.nn as nn
import time
import argparse
import torchvision.transforms as transforms
import torchvision.datasets as dsets
import socket
import math
import pandas as pd
# Import the DistributedDataParallel module needed for distributed training
from torch.nn.parallel import DistributedDataParallel as DDP

############## Define the convolutional neural network ##############
class CNNModel(nn.Module):
    # Definition of the CNN
    def __init__(self):
        super(CNNModel, self).__init__()
        # First Convolutional layer + ReLU
        # This layer takes 1 image and outputs 16 feature maps by using a kernel of size 5 and stride 1
        self.cnn1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5, stride=1, padding=0)
        self.relu1 = nn.ReLU()
        # First Max pooling layer
        # This layer uses a kernel of size 2 and default stride 2
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)
        # Second Convolutional layer + ReLU
        # This layer takes 16 feature maps and outputs 32 feature maps
        self.cnn2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=1, padding=0)
        self.relu2 = nn.ReLU()
        # Second Max pooling layer
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)
        # Fully connected layer
        # This layer takes as input the 32 4x4 feature maps and has an output of size 10
        self.fc1 = nn.Linear(32 * 4 * 4, 10)
    # Defines the computation performed at every forward pass
    def forward(self, x):
        # First Convolutional layer + ReLU
        out = self.cnn1(x)
        out = self.relu1(out)
        # First Max pooling layer
        out = self.maxpool1(out)
        # Second Convolutional layer + ReLU
        out = self.cnn2(out)
        out = self.relu2(out)
        # Second Max pooling layer
        out = self.maxpool2(out)
        # This flattens the feature maps
        out = out.view(out.size(0), -1)
        # Fully connected layer
        out = self.fc1(out)
        return out

############## Define the main function ##############
def cnn(rank, world, args):
    # Set the seed for generating random numbers
    torch.manual_seed(0)
    # Use the CNN define above
    model = CNNModel()
    # Set the device to run the algorithm on GPU
    torch.cuda.set_device(device)
    device = torch.device("cuda:0")
    model.to(device)
    # Wrap the model with the DDP module
    # The model's parameters from the node with rank 0 are broadcasted to the other nodes
    ddp_model = DDP(model, device_ids=[0])

    batch_size = args.batchsize
    numw = args.nw
```

```
    # Choose the best num_workers argument based on testing
    num_workers_cases = {
        2: {50: 3, 64: 6, 150: 8, 250: 10},
        4: {50: 3, 64: 5, 150: 8, 250: 9},
        6: {50: 3, 64: 4, 150: 6, 250: 7},
        8: {50: 3, 64: 4, 150: 6, 250: 5},
        10: {50: 3, 64: 4, 150: 5, 250: 5},
        12: {50: 3, 64: 4, 150: 4, 250: 4},
        14: {50: 3, 64: 4, 150: 4, 250: 4},
        16: {50: 3, 64: 3, 150: 4, 250: 3},
    }
    def adapt_num_workers(world_size, batch_size):
        return num_workers_cases.get(world_size, {}).get(batch_size, numw)
    n_w = adapt_num_workers(world, batch_size)
    # Select the loss function
    criterion = nn.CrossEntropyLoss().to(device)
    # Select the optimization method
    # Define baseline optimizer with particular hyperparameters
    optimizer = torch.optim.SGD(ddp_model.parameters(), lr=0.1, momentum=0.9, nesterov=True)
    # Download the train dataset
    train_dataset = dsets.MNIST(root='./scratch/snx3000/sgonalve/project/data',
                                train=True,
                                transform=transforms.ToTensor(),
                                download=True)
    # Split the train set for each node by evenly distributing the samples without duplication
    train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset,
                                                                    num_replicas=world,
                                                                    rank=rank)
    # Define num_workers that will load the train dataset on each node and the local batch size
    train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                               batch_size=batch_size,
                                               shuffle=False,
                                               num_workers=n_w,
                                               pin_memory=True,
                                               sampler=train_sampler)
    # Download the test dataset
    test_dataset = dsets.MNIST(root='./scratch/snx3000/sgonalve/project/data',
                               train=False,
                               transform=transforms.ToTensor())
    # Define num_workers that will load the test dataset and the batch size
    test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                              batch_size=batch_size,
                                              shuffle=False,
                                              num_workers=n_w,
                                              pin_memory=True)

    num_epochs = args.epochs
    iter = 0

    # Keep track of the time on node with rank 0
    if rank == 0:
        start_time = time.time()
    # Train the model
    for epoch in range(num_epochs):
        # First part of the algorithm
        if epoch < math.ceil(num_epochs / 3):
            for i, (images, labels) in enumerate(train_loader):
                # Send images and labels of this batch to GPU for processing
                images = images.to(device, non_blocking=True)
                labels = labels.to(device, non_blocking=True)
                # Perform the Forward pass to get the predictions
                outputs = ddp_model(images)
                # Compute losses
                loss = criterion(outputs, labels)
                # Set gradients to None
                optimizer.zero_grad(set_to_none=True)
                # Backpropagation and gradient synchronization across nodes
                loss.backward()
                # Update the parameters of the model
                optimizer.step()

                iter += 1
```

```
        # Second part of the algorithm
        elif math.ceil(num_epochs / 3) <= epoch <= 2 * math.ceil(num_epochs / 3):
            for i, (images, labels) in enumerate(train_loader):
                # Send images and labels of this batch to GPU for processing
                images = images.to(device, non_blocking=True)
                labels = labels.to(device, non_blocking=True)
                # This is the synchronization iteration
                if iter % 2 == 8:
                    # Perform the Forward pass to get the predictions
                    outputs = ddp_model(images)
                    # Compute losses
                    loss = criterion(outputs, labels)
                    # Backpropagation and gradient synchronization across nodes
                    loss.backward()
                    # Update the parameters of the model
                    optimizer.step()
                    # Set gradients to None
                    optimizer.zero_grad(set_to_none=True)
                else:
                    # Context manager to disable gradient synchronizations across nodes
                    # Here the gradients are accumulated without updating the parameters of the model
                    with DDP.no_sync(ddp_model):
                        # Perform the Forward pass to get the predictions
                        outputs = ddp_model(images)
                        # Compute losses
                        loss = criterion(outputs, labels)
                        # Backpropagation
                        loss.backward()

                iter += 1

        # Third part of the algorithm
        else:
            for i, (images, labels) in enumerate(train_loader):
                # Send images and labels of this batch to GPU for processing
                images = images.to(device, non_blocking=True)
                labels = labels.to(device, non_blocking=True)
                # This is the synchronization iteration
                if iter % 12 == 0:
                    # Perform the Forward pass to get the predictions
                    outputs = ddp_model(images)
                    # Compute losses
                    loss = criterion(outputs, labels)
                    # Set gradients to None
                    optimizer.zero_grad(set_to_none=True)
                    # Backpropagation and gradient synchronization across nodes
                    loss.backward()
                    # Set gradients to None
                    optimizer.step()
                else:
                    # Context manager to disable gradient synchronizations across nodes
                    # Here the gradients are computed and used for updating the parameters of the local model
                    with DDP.no_sync(ddp_model):
                        # Perform the Forward pass to get the predictions
                        outputs = ddp_model(images)
                        # Compute losses
                        loss = criterion(outputs, labels)
                        # Set gradients to None
                        optimizer.zero_grad(set_to_none=True)
                        # Backpropagation
                        loss.backward()
                        # Update the parameters of the local model
                        optimizer.step()

                iter += 1

    # Model averaging across nodes
    for p in ddp_model.parameters():
        size = float(dist.get_world_size())
        # Allreduce the parameters by sum
        dist.all_reduce(p.data, op=dist.ReduceOp.SUM)
        # Each nodes computes the average
```

```
        p.data /= size
    if rank == 0:
        stop_time = time.time() - start_time
        # Compute the accuracy of the trained model
        correct = 0
        total = 0
        # Iterate through test dataset
        for images, labels in test_loader:
            # Send images and labels of this batch to GPU for processing
            images = images.requires_grad_().to(device)
            labels = labels.to(device)
            # Perform the Forward pass to get the predictions
            outputs = ddp_model(images)
            # Get predictions from the maximum value
            _, predicted = torch.max(outputs.data, 1)
            # Compute total number of labels
            total += labels.size(0)
            # Add to correct predictions
            if torch.cuda.is_available():
                correct += (predicted.cpu() == labels.cpu()).sum()
            else:
                correct += (predicted == labels).sum()
        # Compute accuracy
        accuracy = 100 * correct / total

############## Setup ##############
if __name__ == '__main__':
    # Parse the arguments
    parser = argparse.ArgumentParser()
    parser.add_argument('-bs', '--batchsize', default=150, type=int)
    parser.add_argument('-e', '--epochs', default=16, type=int)
    parser.add_argument('-p', '--p', default=1, type=int)
    parser.add_argument('-nw', '--nw', default=6, type=int)
    args = parser.parse_args()
    # Setup the environment
    os.environ['MASTER_PORT'] = '29501'
    os.environ['WORLD_SIZE'] = os.environ['SLURM_NNODES']
    os.environ['LOCAL_RANK'] = '0'
    os.environ['RANK'] = os.environ['SLURM_NODEID']
    node_list = os.environ['SLURM_NODELIST']
    master_node = subprocess.getoutput(
        f'scontrol show hostname {node_list} | head -n1')
    os.environ['MASTER_ADDR'] = master_node
    if not dist.is_initialized():
        # Initialize the process group and choose the communication backend
        dist.init_process_group(backend="nccl")
    world_size = dist.get_world_size()
    rank = dist.get_rank()
    # Initiate the training of the CNN
    cnn(rank, world_size, args)
```

References

[1] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha. Unsupervised real-time anomaly detection for streaming data, 2017.

[2] D. Amodei et al. Deep speech 2: End-to-end speech recognition in english and mandarin, 2016.

[3] E. Anello. Visualizing the feature maps and filters by convolutional neural networks. https://medium.com/dataseries/visualizing-the-feature-maps-and-filters-by-convolutional-neural-networks-e1462340518e, 14.01.2022.

[4] Apple. Blurring an image. https://developer.apple.com/documentation/accelerate/blurring_an_image, 14.01.2022.

[5] D. Bank, N. Koenigstein, and R. Giryes. Autoencoders, 2020.

[6] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, 2018.

[7] Brock, Donahue, and Simonyan. Large scale gan training for high fidelity natural image synthesis, 2019.

[8] D. C. Cireşan, A. Giusti, L. M. Gambardella, and J. Schmidhuber. Mitosis detection in breast cancer histology images with deep neural networks, 2013.

[9] T. Contributors. Cuda semantics. https://pytorch.org/docs/stable/notes/cuda.html#use-nn-parallel-distributeddataparallel-instead-of-multiprocessing-or-nn-dataparallel, 19.01.2022.

[10] T. Contributors. Distributed communication package - torch.distributed. https://alband.github.io/doc_view/distributed.html, 19.01.2022.

[11] P. Covington, J. Adams, and E. Sargin. Deep neural networks for youtube recommendations, 2016.

[12] CSCS. Piz daint. https://www.cscs.ch/computers/piz-daint/, 17.01.2022.

[13] M. Dash, H. Liu, and J. Yao. Dimensionality reduction for unsupervised data, 1997.

[14] C. Diehl, T. Sievernich, M. Krüger, F. Hoffmann, and T. Bertram. Umbrella: Uncertainty-aware model-based offline reinforcement learning leveraging planning, 2021.

[15] A. Dosovitskiy, J. T. Springenberg, M. Riedmiller, and T. Brox. Discriminative unsupervised feature learning with convolutional neural networks, 2014.

[16] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position, 1980.

[17] E. Gbenga Dada et al. Machine learning for email spam filtering: review, approaches and open research problems, 2019.

[18] R. Gupta. Cnns for time series applications. https://waterprogramming.wordpress.com/2021/06/14/cnns-for-time-series-applications/, 14.01.2022.

[19] B. Hayete, F. Gruber, A. Decker, and R. Yan. Identification of latent variables from graphical model residuals, 2021.

[20] Q. Ho et al. More effective distributed ml via a stale synchronous parallel parameter server, 2013.

[21] F. Kamalov, L. Smail, and I. Gurrib. Stock price forecast with deep learning, 2021.

[22] I. Kandel and M. Castelli. The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset, 2020.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks, 2012.

[24] M. Langer, Z. He, W. Rahayu, and Y. Xue. Distributed training of deep learning models: A taxonomic perspective, 2020.

[25] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Handwritten digit recognition with a back-propagation network, 1989.

[26] Y. LeCun, C. Cortes, and C. J. Burges. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/, 24.01.2022.

[27] S. Li et al. Pytorch distributed: Experiences on accelerating data parallel training, 2020.

[28] T. Lin, J. Lee, K. K. Patel, S. U. Stich, and M. Jaggi. Don't use large mini-batches, use local sgd, 2020.

[29] X. Mao et al. Least squares generative adversarial networks, 2017.

[30] S. McCandlish, J. Kaplan, D. Amodei, and OpenAI Dota Team. An empirical model of large-batch training, 2018.

[31] V. M. Mnih et al. Human-level control through deep reinforcement learning, 2015.

[32] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust stochastic approximation approach to stochastic programming, 2009.

[33] F. Niu, B. Recht, and C. Ré. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent, 2011.

[34] NVIDIA. Nvidia nccl. https://developer.nvidia.com/nccl, 19.01.2022.

[35] A. Olaode, G. A. Naghdy, and C. Todd. Unsupervised classification of images: A review, 2014.

[36] G. F. Pfister. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O*, 42(617-632):10, 2001.

[37] PyTorch. Writing distributed applications with pytorch. https://pytorch.org/tutorials/intermediate/dist_tuto.html#point-to-point-communication, 19.01.2022.

[38] Pytorch. Pytorch at tesla. https://www.youtube.com/watch?v=oBklltKXtDE, 20.01.2022.

[39] H. Robbins and S. Monro. A stochastic approximation method, 1951.

[40] H. Robbins and S. Monro. A stochastic approximation method, 1951.

[41] M. Schmidt. Cpsc 540: Machine learning - convergence of gradient descent (lecture slides), Winter 2017.

[42] C. J. S. Shallue et al. Measuring the effects of data parallelism on neural network training, 2019.

[43] A. Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network, 2018.

[44] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le. Don't decay the learning rate, increase the batch size, 2018.

[45] J. Steppan. Sample images from mnist test dataset. https://en.wikipedia.org/wiki/MNIST_database#/media/File:MnistExamples.png, 24.01.2022.

[46] A. Tamar, Y. Wu, G. Thomas, S. Levine, and P. Abbeel. Value iteration networks, 2017.

[47] Tesla. Artificial intelligence and autopilot. www.tesla.com/AI, 12.12.2021.

[48] TheEconomist. The world's most valuable resource is no longer oil, but data. https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data, 07.01.2022.

[49] P. Tiwari, S. Mehta, N. Sakhuja, J. Kumar, and A. K. Singh. Credit card fraud detection using machine learning: A study, 2021.

[50] USI. Institute of computational science. https://www.ics.usi.ch/, 17.01.2022.

[51] G. Vachkov and H. Ishihara. On-line unsupervised learning for information compression and similarity analysis of large data sets, 2007.

[52] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator, 2015.

[53] X. Wang et al. Esrgan: Enhanced super-resolution generative adversarial networks, 2018.

[54] Y. Wu et al. Google's neural machine translation system: Bridging the gap between human and machine translation, 2016.