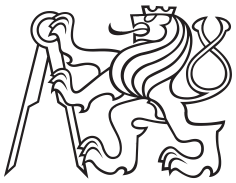
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Distributed training of neural networks

Bc. Petr Tománek

Supervisor: doc. Ing. Tomáš Pevný, Ph.D.
August 2021

Acknowledgements

I would like to thank my supervisor doc. Ing. Tomáš Pevný, Ph.D. for the opportunity to work on an interesting topic, for his patience and valuable guidance. His positive behaviour kept me through my work on this thesis.

Second, I would like to thank the faculty for taking great care of their student, having amazing professors and teaching personnel and creating an wonderful environment for writing this thesis.

Finally, I wish to thank my parents and my fiance Lucie, that have supported me during the hard times. I want to thank my friends and my siblings, who always gave me some things to think about, and motivated me when I was down.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stip- ulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to con- clude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on August 13, 2021

.....

Abstract

Goal of this thesis was to explore and compare the state-of-the-art methods of distributed gradient descent. This thesis focuses on improving the stochastic gradient descent method, that is being applied across Machine Learning. Based on the research we have selected algorithms, that are described and implemented in Julia programming language.

The data for purposes of analysis are selected from domain of cybersecurity with not only huge number of model parameters, but especially huge amount of training data. The analysis is performed on these datasets.

Multiple distributed algorithms of Stochastic Gradient Descent were implemented as part of this thesis, that achieve better results than state-of-the-art methods in regard to processing time.

Keywords: SGD, Stochastic gradient descent, distributed processing, Machine learning, Deep learning

Supervisor: doc. Ing. Tomáš Pevný, Ph.D.
Centrum umělé inteligence,
Resslova 9,
Praha 2

Abstrakt

Cílem této práce je prozkoumat a porovnat nejmodernější řešení pro distribuovanou metodu největšího spádu. Práce se zaměřuje na vylepšení metody největšího spádu (SGD), která je využita napříč strojovým učením. Na základě provedené rešerše jsou vybrány algoritmy, které jsou dále popsány a implementovány v programovacím jazyce Julia.

Pro účely analýzy jsou využita data z bezpečnostní domény nejen s velkým počtem parametrů modelu, ale především i velikostí trénovacích dat. Tato data jsou využita ve formě hierarchického víceinstančního modelu. Na těchto datech je následně provedena analýza.

V rámci práce vznikly implementace distribuovaných algoritmů metody největšího spádu, které dosahují lepších výsledků než nejmodernější řešení, vzhledem k rychlosti zpracování.

Klíčová slova: SGD, Metoda největšího spádu, distribované procesování, Strojové učení, Hluboké učení

Překlad názvu: Distribuované učení neuronových sítí

Contents

Appendices

Figures

Tables

Chapter 1

Introduction

1.1 Background

We are living in a golden age of data aggregation, statistics and data analysis. The human population is steadily increasing in numbers and there is an even greater growth rate in the amount of device usage (both online and offline).

It is estimated that in the year 2023 there will be 4.3 billion[1] mobile phone users. The amount of social media users is estimated to hit around half of the world's population in the year 2025[2]. At the time this thesis is being written, almost a third of the social media users are based in East Asia.

In the past, most devices used to have applications that worked with local data with little information aggregated for processing, but with improvements of technology [3] data aggregation is abruptly increasing.

The amounts of data generated by sensors of a single autonomous vehicle is around 12TB/day[4], with some articles even mentioning it is as high as

[1] https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

[2] https://www.statista.com/markets/424/topic/540/social-media-user-generated-content/

[3] The latest noticeable technologies are the introduction of 5G internet, the addition of Neural Engine cores in Apple M1 and the switch to ARM - RISC architecture, or NVME improvements.

[4] https://blocksandfiles.com/2020/02/03/autonomous-vehicle-data-storage-is-a-game-

750MB/s[5], and that is just one vehicle. Even a single study of drivers of autonomous vehicles by MIT amounted to around 3.5-billion video frames [FBG+17].

At the moment more than 90% of all the data generated has been generated in the past few years[6]. It is predicted that the speed of data generation is (much) faster than the speed of data processing and it is estimated that more than 50%[7] of data is the *dark data*[8].

1.2 Motivation

If all the data generated by the usage of social media were saved, it would fill all the storage in the world in few minutes. The only way to make use of the data is to process it. In conclusion processing as much data as possible in a reasonable time is the only option. With limited resources it requires finding the best possible algorithm.

Algorithms are essentially recipes. We, as a society, have recipes for almost everything. In some cases there are too many of them and we are overwhelmed by the amount. Finding the correct recipe can be close to impossible as some work well in one domain but are lacking in some attribute in a different one. It is often advised to work with a "good enough" solution, as finding the best solution is sometimes impossible. [9]. Improving the recipe, in this case the algorithm, in one domain comes at the cost of some other domain.

In computer science, the algorithms, or whole applications, act as recipes to solve select problems. The algorithms solve the problem by using a sequence of steps. It is not trivial to find a working algorithm for some cases. Furthermore, application quality may vary. Anyone can create an application.[10] To distinguish good algorithms from the rest there has to be some parameters by which they can be evaluated. Applications may be simple to solve some trivial problem, but are fairly complex to solve harder problems.

---

of-guesses/

[5] http://www.kurzweilai.net/googles-self-driving-car-gathers-nearly-1-gbsec

[6] https://www.sciencedaily.com/releases/2013/05/130522085217.htm

[7] https://priceonomics.com/companies-collect-a-lot-of-data-but-how-much-do/

[8] The type of data that is collected but never used.

[9] In real time. Some problems are in class NP-Hard, or even worse in recursively enumerable class of problems.

[10] In some programming languages it's easier than in others (eg. Julia vs C).

"*No application is flawless.*"

A developers' saying

In my work experience of more than 5 years as a software developer, I have never seen a perfect application. Every application has at least one problem in some domain. The application problem domains are mainly functionality, stability, processing, memory, or network inefficiency, or are good only in a case specific environment and do not work (as well) in other cases. Most of the time some inefficiency leads to other problems, like slowdowns in processing efficiency.

1.3 Data generation

The data generation spectrum is incredibly huge. Depending on the problem domain, be it application usage, sensor activity, or network activity, data is being generated all the time. Some data is generated automatically (eg. by periodic sensor scanning), other data is generated by some form of usage either by the user (eg. device usage, social media application usage or virtual reality application usage) or by some form of communication (eg. network activity).

As described before, the amounts of data generated is growing. Not only the amount, but even the size of data files (eg. due to sensors' improved sound and video recording resolution) and the type of data, for example that of new resources being tracked (eg. voice assistants and GPS navigation).

Current applications monitor their usage by customers, the users, to classify them into customer groups and better predict which products to offer. The social networks are created to keep the users addicted to their products, eg. to scroll more to entertain themselves longer on their platform.[11]

[11] Shoshana Zuboff's book Surveillance Capitalism, 2019

1.4 Data storage

Data stores, containing some structured data, are one of the problem domains in computer science. Data structures can be saved into simple text files, formatted files (eg. comma separated files, csv) or into some database systems.

In last more than three decades the standard for storing data was *relational database systems*. In relational databases the data is divided into subgroups, called tables, and the relations between the tables creates the complex data logic.

This relational model does work well with smaller amounts of data. Depending on the hardware, relational databases work well even for tens of Gigabytes (with some improvements, like indexing) but have problems with larger sizes.

Another problem is that relational databases are not scalable. Relational database is built to be run on a single computer. To improve the performance of a relational database it is necessary to buy very expensive components.

1.4.1 Big data

To solve the problem of increases in the amounts of data, a new term was introduced, *Big data* systems. Big data is a field of computer science methods solving various problems often containing larger amounts of data.

Big data systems solve the problem encapsulated in the abbreviation called the $5V$s. The *data volume*, possibly even Petabytes of data, *data variety*, possibly unstructured data (eg. text or images), *data velocity*, the incredible speed of data generation, *data variability*, changes in data structure, and the *data value*, the profitability of the saved data analysis.

The CAP theorem states that distributed databases cannot guarantee the following three properties at once: *consistency* - every read receives the latest write, *availability* - every request is served by the database, and *partition tolerance* - the system operates even when the connection between nodes fails. To comply as much as possible with the relational database standard, which

can provide all three CAP guarantees (as the data is not partitioned), big data is able to guarantee just two out of the three CAP requirements at once.

Big data instead guarantees the BASE shortcut: *basic availability* - reads and writes work on all nodes, but there are no consistency guarantees, *soft state* - the state may not yet propagate as there is no consistency guarantee, and the *eventual consistency* - after some time (if the system is working) the change will eventually propagate.

The NoSQL databases are further divided into categories. Core types are Key-value stores, Wide column stores, Document stores and Graph databases. Each of these stores and databases are useful in various places.

Key-value stores are useful for simple problems, like session data or user profiles. Wide column stores are for structured flat data, like event logs and content management. The document stores have the whole data stored in some structure, like JSON formatted text file from event logs and web analytics. Graph databases are stores of nodes and their relationships, like social networks and routing.

This thesis uses the data generated in, or rewritten to, JSON format, based on the security domain, with logs from routers and malware network communication for algorithms analysis. More details about the data used in Section 5.2, Experimental datasets.

1.4.2 Data privacy concerns

Base solution of data analysis is to collect the data on the server, where the sampled data is prepared to datasets. When data is on the servers, the data analysis is run and for example neural networks are used to create models.

Most of the data is not allowed to be shared or transferred, because it is private. But applications try to empower users with AI, like personalised choice of music, videos, movies and news, improving photos or advice on sporting performance.

The problem may not be only the privacy of the data, but the aggregation of data may not be possible. Reasons stretching from the owner's choice, to the privacy or security reasons. The aggregation of the data may be

impossible to solve as the devices will not be connected to the network any given time or devices may be disconnected completely.

Possible solution having privacy of data at its core is Federated Learning. ederated learning is learning on edge devices using local data without sharing them. It is a very interesting solution but it is out of the scope of this thesis and will not be described further. All of the solutions in this thesis are traditional centralized and decentralized solutions. This will not be stated further.

The drive to get better models and better predictions means to save as much data there is[12]. The collection of data is so fast that no human being can read it all. Luckily the humans aren't the only ones who can create the models by learning from data. New field of study has emerged to solve this problem - Machine learning.

1.5 Problem statement

Inter-device communication is one of the biggest data generators as point to point communication between devices is still done with some proxy. In almost any network it is done via network router and possibly servers in the middle of the conversation as well.

In computer security, or cyber-security, it is necessary to distinguish malicious actors from actors with fair intentions. There are the applications on the computer, the devices on the network or the communication on the network. In any case, finding out the intruder, or malicious intent, is crucial as soon as possible.

The machine learning tries to solve these challenging issues[13]. These types of challenging issues are often complex and require heavy processing power. The main problem in Machine Learning is to solve optimization problems with high dimensions of parameters and huge amounts of data.

The method called Gradient Descent is quite simple and solves many non-trivial problems, but with larger amounts of data and higher dimensions of

[12] All the data the companies are allowed to take. There are some restrictions, some ways to keep safe, but the collection of data is pushed to the limits.

[13] Issues like predicting the attacker and taking action against the intent.

the parameter space it works slower and slower. This is well improved by processing random smaller subsets of data - this method is called "Stochastic Gradient Descent". Ultimately even Stochastic Gradient Descent doesn't converge as fast with huge datasets, as processing smaller batches leads to slower convergence. To improve convergence speed and find good enough solutions even faster, Stochastic Gradient Descent needs to be paralellized.

Unfortunately even parallelism has limits. The first is the cost limit, where the cost of slightly better components grows exponentially. The second is that, if the costs are not the problem, buying better than the best components is not an option. The only option is to use distributed processing. The essence is to distribute the problem to many computers. It is a much less costly option and the main attribute is that it is scalable, using clusters of more basic computers.

The Julia programming language is relatively new. It is designed for mathematical and numerical computing which is amazing for high performance Machine Learning problems with large datasets. It has many qualities but as it is fairly new, some of the machine learning algorithms, or their parallelised and distributed versions, are not yet implemented. The most of the implemented Stochastic Gradient Descent versions in Julia are (to my knowledge) purely serial[14].

The main machine learning framework in Julia Programming Language is *Flux.jl*[ISF+18] containing fundamentals like learning models and data structures. Additionally, multiple libraries exist to either simplify the process of creating the machine learning task or allow more support, like *Zygote.jl*[Inn18a]. The only exception of parallelisation of machine learning tasks is *PrayTools.jl*[15].

This thesis is targeting the data analysis using machine learning method in distributed systems and therefore we will describe these parts in more detail. In the end of this chapter we will sum up the challenges for stochastic gradient descent in the parallel and distributed systems and define the problems for next chapter.

[14] With exception of parallelisation from Julia basic commands like @*parallel*
[15] https://github.com/pevnak/PrayTools.jl

1.6 Objectives

The first objective of this diploma thesis is to research and analyse the Stochastic Gradient Descent method with additional research into prior work in the field of Stochastic Gradient Descent in a parallel and distributed setting, as well as to find and select multiple state-of-the-art distributed algorithms to be implemented.

The second objective is to implement the selected state-of-the-art distributed algorithms and create a public library for the Julia Programming Language, and to analyse the implemented algorithm with different initial parameters, different settings and different datasets.

1.7 Outlines

This paper is divided into two parts based on the objectives. The first part is research, the second part is implementation and analysis.

This Chapter 1 introduces the problem domain by briefly explaining basic concepts in broader context. Chapter 2 talks about concepts of data analysis, explains training of model and later emphasises on gradient descent and machine learning. Chapter 3 talks about concurrency concepts and defines the stochastic gradient descent types in distributed systems. Chapter 4 summarizes research in the topic of stochastic gradient descent in distributed systems and defines the algorithms to be implemented.

Chapter 5 defines performance metrics, describes experiment datasets and gives short introduction on algorithm parameters. Further analysis discusses the experiment results and reasons about future analysis.

Chapter 6 concludes the work and discusses further work.

Chapter 2

Data analysis

Process of data analysis can be generalized into a process consisting of the following steps[OS13].

1. Data requirements
2. Data collection
3. Data processing
4. Data cleaning
5. Exploratory data analysis
6. Modelling and algorithms
7. Data product
8. Communication

This thesis focuses on number 6 - Modelling and algorithms. First we will define data analysis notion with emphasise on model creation and model training. Later we will focus on improving the model training in more complex systems. We shall define machine learning in the following section. Specifically, we shall be using distributed machine learning algorithms.

2.1 Model creation

The analysed dataset creates a mathematical *model* of the problem. The model simplifies the logic behind the dataset trends. It adds the possibility of predicting a value based on arbitrary input.

The model exists in the form of a function. This function takes data input, which we will call the data *sample*, and creates an output, called a *prediction*. The complexity of the model often negatively correlates with interpretability. Simple models are more easily explained, as the function of the model parameters may be simply displaying trends like in Linear Regression, whereas complex models, like Neural Networks, chaining multiple layers of neurons, are much harder to interpret.

$$\text{model}(X) = \theta^T X = \hat{y} \tag{2.1}$$

An example of a model is the prediction based on model parameters shown in Formula 2.1. The model is a function with θ as its parameter and the variable X represents data for which the prediction should be made. The resulting $\hat{y}$ is the prediction.

Assessing the quality of the model is done using multiple methods, based on the predicted value. If the simple model is inaccurate, that is if the predicted value is far from the real value or different in case of classification, it is necessary to use more complex methods.

We are trying to achieve the best model possible for a specific problem and as such we need to improve the initial model.

2.2 Model training

The set of input samples used for training is called a *training dataset*. The learning methods, often called *optimizers*, try to minimize the value of some objective function. The *objective function* is a function that is optimized during the training, in the context of minimization it is a cost function. The *cost*

function is the total of all the individual loss functions potentially combined with regularization, whereas the individual *loss function* is a measure of the penalty of single sample.

In the literature the terms *objective function*, *cost function* and *loss function* are interchangeable[GBC16]. We will be using the loss function as an objective function in this thesis.

$$\text{loss}(X|\theta) = \frac{1}{m}\sum_{i=1}^{m}\text{loss}_i(\text{model}(x_i|\theta), y_i) \tag{2.2}$$

where loss_i is individual loss based on an model estimate and real value of single sample i. We define loss as sum of all the individual losses.

$$\underset{\theta}{\text{argmin}}\Big(\text{loss}(X|\theta)\Big) \tag{2.3}$$

The training of model is a process of changing the model parameters or weights and bias of the model by minimizing the loss. The learning process of the model is by solving the Formula 2.3, that is minimizing the loss of model based on the observed data samples.

Measuring the quality of a model is based on the set of samples called the *testing dataset*. There are multiple methods for model quality evaluation, but we will mention only a few that will be explained in more detail in the Chapter 5, Experimental results.

Accuracy is the probability of a data sample being classified correctly.

Loss function depends on selection in learning, often selected variant is cross entropy

To have as accurate a model as possible it is necessary to get as much data as possible. It is obvious that smaller amounts of samples are easier and therefore faster to process, as having bigger amounts needs more calculation

and therefore makes the processing harder. A less obvious problem is that the amounts may be too enormous to even be processed in finite time.

One of the methods that gradually improves the solution by moving closer with the intermediate result is called Gradient Descent. This method was invented in 1847[Lem12] and is attributed to Cauchy, but similar methods were proposed by several others.

2.2.1 Gradient descent

Gradient descent is a functional approach where the result is being approximated with some regard to the difference of an actual, possibly measured, value and the value predicted by a model. In statistics, the difference is called an error or the residual and is often used as a cost function. The Gradient Descent algorithm gradually minimizes the cost function.

Using differentiation we can approach the local minimum using the slope of the function, where the slope of a function is described by the first derivative.

A single iteration of the gradient descent method works in two phases:

calculating gradient the method approximates the model of the objective function by partial derivation of each of the parameters, creating a gradient G_t

updating parameters the method subtracts a portion of a gradient, called the learning rate η, from the previous solution, creating new parameters θ_{t+1}

$$\theta_{t+1} = \theta_t - \eta G_t \tag{2.4}$$

One of the challenges of gradient descent is choosing the best step size η. In case we choose an excessive step size, the function can diverge from the objective. On the other hand, if we choose a step size which is too small, it will take an infinite amount of time to find the objective function for the

parameters. Often the 1% step size ($\eta = 0.01$) seems like a good fit. It is simple enough for the calculation, yet big enough to make change in a relatively small amount of iterations.[1]

Stochastic Gradient descent

It is infeasible to use the gradient descent algorithm in cases of Big Data datasets. Even using gradient descent on fairly big datasets (hundreds of Megabytes of formatted text) can get computationally expensive and require huge amounts of memory. Therefore it is necessary to run the gradient descent with smaller batches instead of whole dataset.

Algorithm 1: Flux SGD implementation

```
function train!(loss, ps, data, opt; cb =()->())
  ps =Params(ps)
  ...
  @progress for d in data
    try
      gs =gradient(ps) do
        loss(batchmemaybe(d)...)
      end
      update!(opt, ps, gs)
      cb()
    catch ex
      ...
    end
  end
end
```

Introducing stochasticity to gradient descent allows for selection of random subset of training samples. Additionally smaller batch size allows for the processing of large datasets with a greater amount of iterations and similar results. The random sampling of a training dataset, can greatly improve the speed of processing of the gradients as the gradient descent method calculates only the smaller portion of training dataset, called a "mini-batch". SGD

[1] There are many types of step sizes. Some optimizers are using a decay rate to decrease η in time as the calculation comes closer to the objective, eg. ADAM. Some optimizers calculate the biggest improvement possible in the iteration based on the parameter and the gradient, eg. the backtracking Armijo rule.

selects random samples from the training dataset as the mini-batch and calculates the gradient descent with a "small training dataset".

The implementation of Stochastic Gradient Descent algorithm in Flux Algorithm 1[ISF+18][Inn18b] is considered to be state-of-the-art. We consider the implementation of Algorithm 1, as the base to be compared with. Our implementations of distributed SGD will be described in the Chapter 4, Prior art.

Stochastic Gradient Descent is being used by many systems, from very simple to much more complex. One of the more complex examples is machine learning with the neural networks. The neural networks are usually trained using Stochastic Gradient Descent or its variants[GBC16].

2.2.2 Broyden-Fletcher-Goldfarb-Shanno algorithm

Similar to Gradient Descent the Broyden-Fletcher-Goldfarb-Shanno algorithm, also known as BFGS, is a functional approach of approximation of objective function using descent direction and hessian matrix (e.g. second partial derivatives) which is the main difference to SGD . It is in the family of quasi-Newton methods used to solve large-scale optimizations with smaller number of parameters.

Approximating Hessian matrix of model with many parameters requires large amounts of memory which grows with square of the number of parameters and is therefore very computationally expensive. Even its more recent variant Limited-memory BFGS, also known as L-BFGS, is still very resource heavy.

Compared to SGD the L-BFGS is much more computationally intensive and uses huge amount of memory. The processing complexity of BFGS algorithm is based on approximation of second derivatives, creating Hessian matrices, in contrast to first derivative used in Gradient Descent. L-BFGS requires calculation of gradient on all samples in each step, whereas SGD is working with minibatches. With these problems, usage of BFGS or its variant L-BFGS is infeasible for optimization of large number of model parameters. Recent neural networks have hundreds of billions of learnable parameters.

2.3 Machine Learning

Machine learning is a field of computer science containing a set of methods that gradually improve a model based on a training sample input. Learning from samples in case of the machine learning methods may be seen as a change of the model state based on data in training data input. Machine learning is divided into multiple categories, based on the type of learning, type of prediction or clustering.

The experimental part of this thesis contains implementations of learning algorithms from a subset of machine learning called *feature learning*. Feature learning algorithms are divided into multiple types of learning, based on the training data: *supervised learning*, where the training dataset is mapped to some label, *unsupervised learning*, where the training dataset does not have any mapping to the label and the output needs to be guessed based on the patterns, and semi-supervised learning, which is partially divided into supervised and unsupervised training data. We are using supervised learning for our analysis in Chapter 5, Experimental results.

2.3.1 Neural Networks

A Artificial Neural network (ANN) is an network of interconnected artificial neurons. Most artificial neural networks consist of multiple layers: the input layer, the output layer and possibly multiple hidden layers in the center. The artificial neural network is considered shallow if it has zero or one hidden layer. If the network comprises of more than single hidden layer it is known as a deep neural network.

Neural networks belong to the class of feature learning algorithms and therefore for their learning process the ANN use both labeled and unlabeled training data. The network may consist of multiple layers that are connected in a feed-forward system.

Artificial neural network algorithms were invented for artificial intelligence and resemble the biological brain. The artificial neural network consists of artificial neurons in layers, opposed to the biological brain's biological neurons. The model is then approximated from the previous layer. A single neuron in a layer consists of a bias and vector of weights of neurons, or inputs, from the previous layer. The inputs are multiplied by the weights and accumulated

with subtraction of bias to produce neurons output value.

Artificial neural networks can derive relations in complex data with enough iterations. Shallow neural network uses at most one hidden layer and as such it has been proven as an universal approximator[LLPS93]. This means that when given enough layers and appropriate weights, deep learning algorithms are able to approximate continuous functions to any degree of accuracy (if and only if the network activation function is non-polynomial).

The deep learning family consists of many methods and architectures. Some simple methods can be easily understood, but some methods are too complex to be understood. Often the interpretability of the model negatively correlates with the complexity of the learning methods, as every increase in complexity decreases the understandability of a model.

In this thesis we are looking at the Neural Network as a black box, that is a function where we are able to calculate the sub-gradient (it's approximation) without our knowledge how it is processed. We are not solving any particular problem or category of Machine Learning or Data Analysis. Instead we are solving the general problem of Stochastic Gradient Descent in distributed systems. The Stochastic Gradient Descent method may be used in any machine learning algorithm with loss being expressed as in Formula 2.2, where y_i can be empty.

2.4 Possible improvements of SGD

The stochastic gradient descent is very well thought out and efficient method. It works well with large datasets, huge amounts of model parameters and it finds the solution[2].

2.4.1 Tuning of the method parameters

In basic Gradient Descent there is only one parameter, the learning rate $\eta \in (0, 1)$. Small change in learning rate may bring very different results in few iterations. As mentioned selecting η close to 0 means minimal change,

[2]If the models parameters are differentiable

on the other side selecting η close to 1 with brings divergence. Learning rate is dependent on the problem and its data.

The Stochastic Gradient Descent brings new parameter - size of minibatch. Batch size affects processing speed of gradients and memory usage. This parameter is especially useful in parallelisation of algorithm to allow for faster processing of iterations.

The variants of Gradient Descent change the learning rate method, e.g. ADAM has three more parameters in addition to η.

Balancing all of the algorithms parameters is hard as the best setup is dependent on problem. More complex Stochastic Gradient Descent method variants will bring more parameters to be selected.

2.4.2 Improving the processing or convergence time

We have defined Stochastic Gradient Descent's iteration as having two steps: calculating gradient G_t and then updating parameters θ_{t+1}.

These steps are done serially in order. In any case of improvement by using divide-and-conquer algorithm the methods will need to run these steps and add possibly some more for communication.

In this chapter we have now explored the problem domain of data analysis and gradient descent in general. In the following Chapter 3, Concurrency of processing, we will improve measurable performance of SGD by being in purely serial problem space to new dimension of speedup based on distributing processing into multiple workers. We will describe the concurrency domain in more detail.

Chapter 3

Concurrency of processing

In the first part of this chapter we will discuss processing of tasks, with an emphasis on improving run time. We will show the issues that may slow down the execution of serial task execution and provide some long-standing solutions. Concurrency will be described both theoretically and practically with regard to the real world.

Second part is mainly be focused on parallelisation and the distribution of previously serial tasks.

This thesis studies data analysis using machine learning methods in distributed systems and therefore we will describe the distributed parts in more detail. At the end of this chapter we will sum up the challenges for stochastic gradient descent in parallel and distributed systems and create algorithm detail definitions for the next chapter.

3.1 Concurrency in real life

Since ancient times many recipes have been inherently sequential, in computer science we call it *serial processing*. Recipes like "First cut the vegetable, then start boiling water; when water is boiling, throw in vegetables.." are easy to follow. We know[1] what action to do at what time as the order is described

[1] If we understand what exactly the recipe describes..

and is straightforward. The problem with processing recipes serially, is that some actions, like "when" and "while" may require us to wait some time period (e.g. "wait for water to boil").

One of the solutions is changing the order as it may decrease the amount of time needed to complete the whole task. This is called *scheduling*. Scheduling is one of many combinatorial optimization problems. It is for a single worker in the polynomial class of problems, yet there are many complications of the problem of scheduling that can make the problem much harder. Depending on the complexity of the tasks, the amount of workers and other conditions, like the order of the tasks, this problem may be even in class of NP-hard[2] problems.

In case the order of the tasks is rescheduled, e.g. "First start boiling water, then cut the vegetables, when water is boiling throw in vegetables..", some task can be done during the waiting period. In this case the vegetable cutting while we are waiting for the water to boil. But the complication may be that the time to cut vegetables is different than the time for water to boil.

In case of slow cutting or a larger amount of vegetables, the water starts boiling earlier than all the vegetables are cut and this leads to problem with solving two problems at the same time[3]. On the other hand in case of a small amount or expert cutting skills, the cutting part is done faster and there is additional idle time[4]. The ideal variant is that the cutting problem is finished at the time the water starts boiling.

Another solution to decrease the processing time was mentioned in the previous paragraphs. It is possible to parallelise. For example, one cook may be creating the side dish while making a main dish. While working on a series of steps there are a few problems with the order, but when doing things in parallel, things start to get even more complicated. The complication may lie in the synchronization of finishing of two separate tasks, like having the main course and the side dish at the same time.

To allow even better parallelisation, the tasks can be distributed into multiple workers. In scheduling there are differences in organization. In one case the workers do the same work at the same time, which is easier to organize. The other case is that each worker works on a different task, where the organization of the tasks is key.

[2] As some variants are closely related to Travelling Salesman Person (TSP), which is well known NP-hard problem.

[3] Solvable by splitting the tasks, if the tasks are divisible

[4] Solvable by doing additional tasks, otherwise the only option is waiting.

3.2 Concurrency in computer science

In computer science, instead of recipes we have algorithms. Serially running algorithms are easy to follow and do not create any disruptions in processing. The main problem in serial algorithms is the wait time. Therefore we use parallelisation of tasks as most problems are (at least in part) parallelisable.

The parallelisation of the problem is done on a single computer using different threads. There are two possibilities of parallelisation on a single computer - running the same code and running different code. A run with same code may be done automatically using single instruction multiple data (SIMD). In case of differing code, meaning different instructions, it is run on different threads in multi-threading systems. Multi-threading allows for the concurrent processing of tasks either on a single processing unit (using scheduling of threads) or more often on multiple processing units.

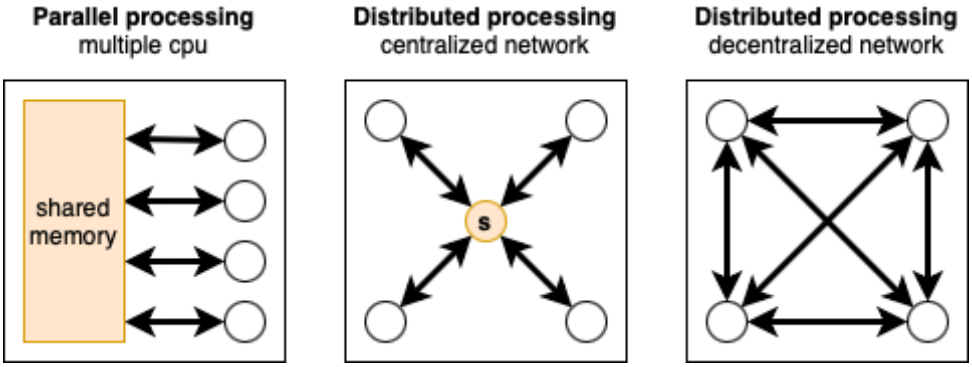


Figure 3.1: Parallel and distributed network types

Algorithms can be parallelised using one of two concurrency model types. The first is the *shared memory model*. It is a local parallelisation model, where all the workers communicate through shared memory and is used for parallelisation. The second is *message passing model* where the messages are passed through the network. This allows for both parallel and distributed processing.

3.2.1 Shared Memory systems

The shared memory model is a model for parallelisation on a single computer. In the shared memory system, workers write the data to shared memory and this way allows all the other workers to read the latest. This introduces the

problem of overwriting data.

Problem: Costly updates in shared-memory systems

Parallelisation on shared-memory systems is possible by having a thread for every worker. Each thread reads the data, calculates the solution, in the case of SGD gradients, and writes to memory. The reading and calculating parts are memory safe but the writing part is unsafe.

We intentionally overlook the caches in threads and simplify the problem by focusing only on shared memory. Threads reading without writing to memory are memory safe. On the other hand, threads writing to memory are not. In case of more than one thread writing to a single memory segment, the memory may become corrupted[5]. Because of this problem, reads and writes to memory are synchronized either on the hardware level by bus locking or a similar locking software implementation.

Locks allow for single write without memory corruption and therefore add one level of memory safety[6]. On the other hand, locking is very time consuming as creating a lock, blocking reads and writes and the rest of the synchronisation process generates massive overhead. Because of locking, the updates in shared-memory systems are very costly.

Furthermore every thread will have its own data value in the cache. To keep all of the threads synchronized, the processor must update all of the thread caches. This action stops the processing for lot of cycles. There are many optimizations to achieve the eventual consistency of the cache, but it still means locking and cache misses.

Costly updates are often the bottleneck of parallel running workers, as there is need for a locking-based mechanism in the shared-memory concept. The memory locking of the solution memory space allows to keep all data in the latest state without loss of precision by overwriting. There are a lot of solutions for solving costly updates, but every solution for that problem seems to bring other often unwelcome aspects to light.

[5]Memory cannot guarantee the data in the correct order, or even that the memory segments will not be mangled. An example may be having two threads write to a single memory segment. A result without locking would have some parts from the first and other parts from the second thread.

[6]Locks do not fix problems with phantom writes and similar problems.

3.2.2 Message Passing Interface systems

In the message passing model the transferred data will not get overwritten, but the challenge is to transfer data between the workers.

In the literature[GKGK03] the message passing system's total time of message transfer consists of three parts:

Startup time (t_s) Time spent on preparation on both sending and receiving nodes,
like programming routes

Per-hop time (t_h) Time spent on transfer between multiple nodes as function of hops,
including network delays and switch latencies

Per-word transfer time (t_w) Time spent based on the length of the message,
including all overheads, like bandwidth

The total time of receiving the message traversing multiple hops is equal to Formula 3.1, where m is equal to size of message in words and l is equal to amount of communication links to traverse.

$$t_{total} = t_s + (m * t_w + t_h) * l \tag{3.1}$$

The hop time t_h is in most platforms very small and may be omitted. We however do not use these terms in this thesis as the amount of transferred data is for both gradients and parameters the same.

We can distribute the data of harder parallelisable problems and process them on clusters of computers. In computer science, distributed processing has more types of network topologies and their specific communications. The networks are divided into two main categories: *centralized networks*, with a server in the middle and server - worker communication, and *decentralized networks*, with special structures and communication between the workers.

Centralized networks

The centralized network is the network that has a device in the middle. It is the most commonly used distributed solution called *star network* topology, as it resembles a star.

The central unit, often called master or a server, aggregates data, whereas the other nodes, called workers, are computation heavy and communicate the processing results to the server. The central unit is communication heavy and may get overloaded with communication. The slowdowns on the central unit often propagate to the workers as the communication in the network is often the biggest overhead.

Decentralized networks

Decentralized networks on the other hand try to solve the problem of overloading the central unit by removing it and keeping the communication between the workers. The amount of communication between the nodes can be decreased by choosing specialized algorithms based on this network topology.

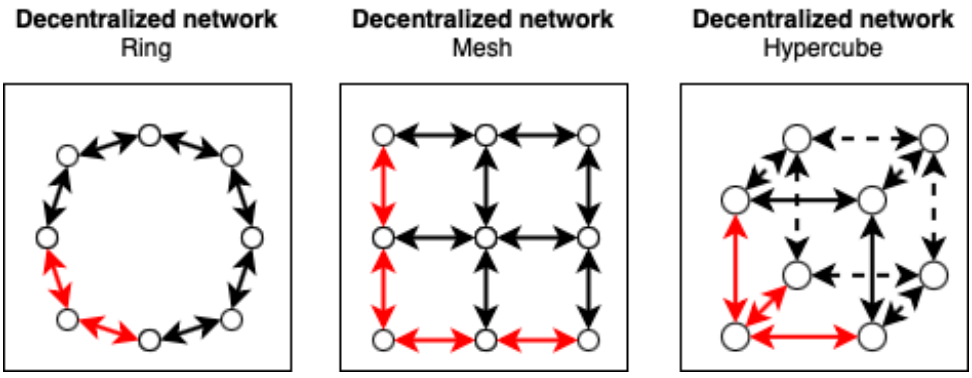


Figure 3.2: Decentralized distributed network topologies

To give an example, for the *Hypercube network* the *AllReduce* method, aggregating the data on the nodes, can be faster if the workers aggregate the data and in the next communication propagate already aggregated data, therefore instead of n communications for a single node there are only $log_2(n)$ communications.

3.3 Distributed systems and SGD

3.3.1 Definition: Network topology type

The selection of network topology is a key factor as the algorithm logic is topology-dependent. Often algorithms are specifically designed for only one specific type of topology.

Problem definition

The communication between two nodes on the network consists of multiple tasks mentioned in Chapter 3.2.2, Message Passing Interface systems. With addition of waiting to get response the problem may become very grave. Most of the solutions for minimization of communication are centralized, meaning a single server and multiple workers. The "single server on network" parallelism can be the biggest problem as the amount of communication between server and all workers is so immense it may overwhelm the server.

The central unit can get stuck, congested or is not scalable enough. Consequences can be vast. Adding more computers means no additional speedup, even worse, there shall be slow-downs or maybe even no processing at all (if the central unit gets stuck).

Qualities of parallelization

In parallel systems, the communication takes place between the workers and shared memory. Aggregation of the model parameters is in the shared memory. All the data, including the model parameters, training dataset and testing dataset, is on a single computer. Improving the processing time is done by improving the computer hardware. The parallel solution is not scalable to more computers.

Qualities of distributed systems - centralization

Distributed centralized systems are built on top of the communication between the server node and worker nodes. The server node and master node exchange global parameters and local parameters or local gradients. The master node may be overwhelmed by the communication. Aggregation of global model parameters takes place on the server node, with special cases having aggregation of local parameters on worker nodes. The training dataset may be partitioned to all worker nodes, testing dataset and global parameters are on the server node. Improving the processing time is done by adding more worker nodes, changing mini-batch size and decreasing the amount of data communication. The centralized solution is well scalable to tens of worker nodes.

Qualities of distributed systems - decentralization

Distributed decentralized systems are built on top of the communication between any worker nodes. The nodes exchange local parameters or local gradients. No single node should be overwhelmed by the communication. Aggregation of local parameters happens on worker nodes. The testing and training datasets may be partitioned to all worker nodes, local parameters are located on the worker nodes. Improving the processing time is done by adding more worker nodes, changing the mini-batch size and decreasing the amount of data communication. The decentralized solution is very well scalable to hundreds of worker nodes.

3.3.2 Definition: Communication type

Both parallel and distributed synchronized systems communicate data from workers. As has been mentioned before, it is either model parameters or gradients. The ideal case is to have all the workers in the communication finish at the same time, but that will not the actual case, as for example some computations are more complex than others. The bottleneck is created by waiting on slower-running workers.

Problem definition

Synchronization in parallel and distributed systems can be described as achieving aggregation of the results from all of the workers. In SGD in general, synchronization means waiting for results from all workers until they finish the iteration. Different speeds of processing generates waiting between iterations for all finished workers and with hundreds or thousands of iterations the problem may become so grave that it is the bottleneck of the algorithm.

Synchronous workers without the latest parameters cannot calculate the next iteration and must wait for communication of model parameters. This generates unwanted staleness of remaining workers. The finished worker could be doing computations for the next iteration, or possibly help slower workers.

There are multiple reasons for differing speeds of aggregation. The most notable are

1. different hardware of processing units
2. network transfers worker-server for centralized, or worker-worker in decentralized systems
3. processing times of gradients[7]

Qualities of synchronous communication

Synchronization means aggregation of all the data, either gradients or local model parameters, on the server node in centralized and on all worker nodes in decentralized systems. The advantage of synchronous communication is that it always uses the latest state of the global model parameters and is able to converge like a serial SGD. The disadvantage is that it has to wait on all the workers, where some slow workers generate big waiting overhead.

[7]The processing times may have a difference in speed of aggregation attributed to processing complexity, or for example background processes on worker processors.

Qualities of asynchronous communication

Asynchronous communication means aggregation of the data from a single node, either gradients or local model parameters, on the server node in centralized and on some worker node in decentralized systems. The advantage of asynchronous communication is that it does not have to wait on slow workers and therefore the speed of iteration is rapid. The disadvantage is that it does not use the latest state of global model parameters therefore the convergence may be slower than serial SGD in terms of iterations, or in the worst case even diverge.

3.3.3 Definition: Periodicity of communication

The communication on the network can be divided into multiple parts that are described in Section 3.2, Concurrency in computer science. In this section we are mainly considering the amount of communication in regard to the *total time of the transfer*. The overhead of communicating every iteration is slowing the processing as the time spent in the communication (transferring huge amounts of data, waiting for response, etc.) on any node means no calculating during that period on that node.

Problem definition

Huge data sizes and number of sources generating new data are increasing. It simply means there is more data to be transferred. In context of data analysis this is not the only problem. The bigger problem is the huge number of data (small updates) during the processing phase of SGD on multiple computers on a distributed system's network. Communication via network can be extremely costly.

In any synchronized solution the communication must happen in every iteration. The problematic part is the amount of data transferred. Both the dataset sizes and sizes of model parameters, or even their gradients, can be very large. With addition of waiting on receiving and responding this problem may kill the promising algorithm.

Qualities of semi-(a)synchronous communication

The combination of a synchronous and asynchronous approach. This type of communication is often implemented as a communication after some period of work on workers. The algorithms often communicate either local model parameters or some portion of the local model parameters with subtracted global model parameters creating a delta that acts like a gradient in some direction. This communication allows for exploration in both synchronous and asynchronous communications. It also tries to solve the problems with waiting on slow workers in case of semi-synchronous communication as it gives more time to the slow workers to catch-up.

3.3.4 Parallelisation of SGD

In previous Chapter 2.4.2, Improving the processing or convergence time, we have defined the steps of Stochastic Gradient Descent algorithm. To parallelise the method we can chose divide-and-conquer algorithm to select sub-problems and try to process them faster.

First step, the calculation of gradients, can be parallelized if we select centralized distributed topology. But it means that in any iteration t communication of the gradients $G_{t,i}$ to server either from any worker $i \in workers()$, or from all of the workers based on selection of communication type. Server will then calculate the global model parameters $\widetilde{\theta}_t$. To keep the latest data on workers this brings additional step of communication of model parameters to worker(s).

calculating gradients $G_{t,i}$ Done on workers

communicate gradients Overhead - Worker sending, server receiving

updating parameters $\widetilde{\theta_{t+1}}$ Done on server

communicate parameters Overhead - Server sending, worker receiving

This means communication will happen every iteration. To allow decentralized distributed topology the solution needs to change again. The worker must be able to update its local parameters. Again there is a choice of communication of worker i with worker j, or of all workers to all workers based on selection of communication type.

calculating gradients $G_{t,i}$ Done on workers

communicate gradients Overhead - Worker sending, worker receiving

aggregation of gradients G_t Overhead - Done on workers

updating parameters θ_{t+1} Done on workers

To allow less frequent communication based on selection of communication type semi-(a)synchronous communication with communication period τ a new solution would need to combine the both previous solutions. Workers would calculate the gradients and update local model parameters while not in communication iteration. The communication iteration would happen every iteration $t \equiv 0 \pmod{\tau}$. In the communication iteration in case of centralized distributed topology would workers send their local model parameters and wait for global model parameters from server. In case of decentralized distributed topology the workers would exchange local parameters.

3.3.5 Introduced problems by parallelisation of SGD

Parallelisation adds possibility of calculating gradient and/or updating parameters, but in case algorithm is incorrectly designed, set up or used the processing may stop. We call the stopped processing waiting. This waiting generates overhead on either worker, server, or in worst case both.

We define here two slowdown problems :

server is not processing all workers calculate gradients/updates

workers are not processing either server is busy, or some worker is stalling

One of more severe problems is if worker or server is shut down from network. In case of worker in asynchronous algorithms this is recoverable, as the network will work, but in case of synchronous communication this is fatal as the server cannot progress. For centralized network it is unrecoverable if the server fails

Part I

Research

Chapter 4

Prior art

In this chapter, we will describe the research of algorithms in prior related academic work. We will describe multiple algorithms from different papers aiming to improve the Stochastic Gradient Descent with regard to the speed-up of the run time by using parallelization in different domains and parts of algorithms. Authors try to improve the run time of the algorithms while minimizing the decrease in convergence.

Every algorithm will have a small description of how the algorithm works, figure describing the run of the first iteration (or similar amount of processing) and the pseudo-code (based on Julia programming language) of an algorithm.

The Stochastic Gradient Descent has multiple parts in which it can be improved. Authors of papers introduced later are working on and often fixing the problems found in the previous papers. These algorithms are mentioned in chronological order as the papers were introduced.

4.1 Definition of terms used in this chapter

We will now use these previously defined terms that will be used throughout this thesis. We are adding the definition and variables in the algorithms to give more clarity.

p Number of workers, in algorithms the function is $nworkers()$

θ Model parameters at node, in algorithms the variable is ps

$\widetilde{\theta}_t$ Center, or global model parameters in t-iteration

θ_t^i Worker-i, or local model parameters in t-iteration

G Gradient at node, in algorithms the variable is gs

G_t^i Worker-i gradient in t-iteration

G_t^* Sum of worker gradients, in t-iteration, $G_t^* = \sum_{i=1}^{p} G_t^i$

η Learning rate, or step-size, in algorithms the variable is opt

τ Communication period, in algorithms the variable is tau

ρ Exploration amount, in algorithms the variable is rho

4.2 Bulk Synchronous SGD[RUL10]

The first paper's goal is to implement parallel algorithms, make improvements or create extensions to the state-of-the-art algorithm called MapReduce. Bulk synchronous systems were created to aggregate data on a server with minimal overhead of the network transfer. The name *Bulk Synchronous* was used as the data on workers was aggregated and sent to the other nodes, possibly a server, as a bulk during periodic checkpointing.

The authors introduced the algorithm using a similar approach to Bulk Synchronous systems, being in a centralized network and having a central unit. The central unit is called a synchronous parameter server. The approach is called Bulk Synchronous SGD method as the gradients are sent in bulk synchronously. The workers periodically send their gradients to the server, the server sums the gradients together, calculates the model parameters and returns the newly calculated parameters to the workers.

$$G_{t^*} = \sum_{j}^{n} G_{t,j} \tag{4.1}$$

where G_{i^*} is the sum of gradients in iteration t, the $G_{t,j}$ is the gradient in iteration t of the worker j.

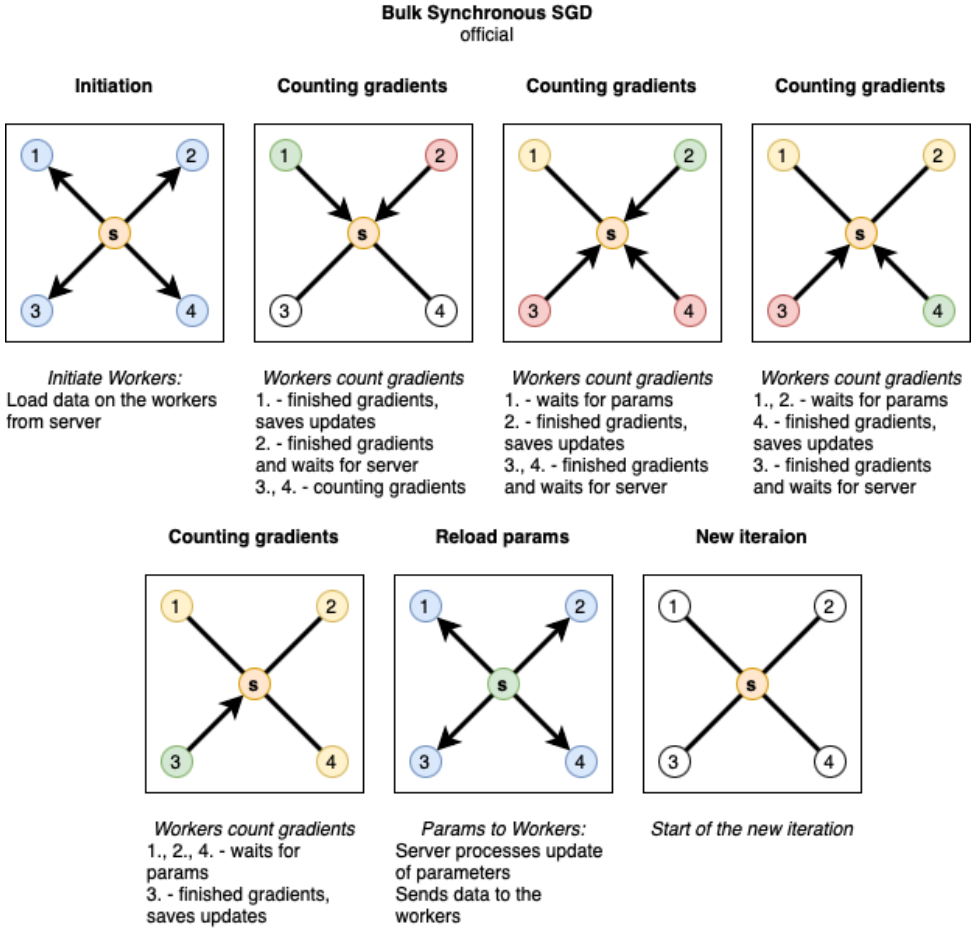


Figure 4.1: Bulk Synchronous algorithm displayed on minimal run.

After the iteration is complete on all workers and all the gradients from the workers are received by the server, the server calculates the mean gradient for the iteration.

$$G_t = \frac{G_{t^*}}{n} \tag{4.2}$$

where G_{t^*} is the sum of gradients in iteration t, the n is number of workers that processed the gradients and G_t is mean gradient of iteration t. The mean gradient is then used to approximate the iteration.

In the Figure 4.1 we have shown a randomized run of single iteration on all workers where server gradually collects the gradients. After successfully aggregating all of the worker gradients, the server then calculates the mean

gradient and then updates the parameters. The blue worker nodes are loading the parameters from the server and then change to white worker nodes where they calculate the gradients. When the worker is finished calculating gradients, it turns either green or red. The green worker nodes are saving the newly calculated gradients onto the server, while the red nodes are waiting to send the gradients to the server. Meanwhile, the yellow workers are waiting for the server to send the newly calculated parameters.

Algorithm 2: Bulk Synchronous algorithm on Server

```
@everywhere function train_BulkSync!(loss, ps, data, opt,
      server_channel; cb =()->(),max_iteration =100)
   t =0 # iteration
   current_workers =workers()
   while t <max_iteration
      next_workers =[]
      send_params_to_workers(current_workers, ps)
      G_{t*} =empty_gradient()
      while !empty(current_workers) &&fetch(server_channel)
         G_{t,j}, worker_j =receive_grad_from_worker()
         G_{t*} .+=G_{t,j}
         remove!(current_workers, worker_j)
         push!(next_workers, worker_j)
      end
      gs =G_{t*} /n
      update!(opt, ps, gs)
      t +=1 # iteration + 1
      current_workers =next_workers
   end
end
```

There are two main waiting periods. Both waiting periods begin after the worker finished calculating the gradient. The first waiting period is when the worker finished calculating the gradient and wants to send the data to server. As server is processing different requests, the worker has to wait. The second waiting period is much more serious, as it affects all the workers that have finished calculating gradients and are waiting for global parameters. This problems comes from struggling workers. We can see these problems in the Figure 4.1, both when the workers are ready to send the gradients (red) and when the workers are waiting to receive the updated parameters (yellow).

The Bulk Synchronous SGD is a distributed synchronous algorithm that collects all the gradients, then counts the mean gradient and with that mean

Algorithm 3: Centralized SGD algorithm on Workers

```
@everywhere function train_centralized_worker!(ps::Params,
      server_channel, worker_channel)
   t =0 # iteration
   while fetch_server_params() !=Nothing
      xs =receive_server_params(worker_channel)
      batch_data =random_minibatch()
      gs, y =calculate_gradient(loss, ps, xs, batch_data...
         )
      send_grad_to_server(server_channel, gs)
      t +=1 # iteration + 1
   end
end
```

gradient updates the parameters of the model.

4.3 Hogwild! SGD[RRWN11]

Hogwild! was one of the first papers aiming to improve the run time of the parallel SGD using the shared-memory approach with local parallelization of SGD without memory locking. In the paper the authors mention the problem of slow updates of parameters in the model because of locks. The authors describe previous work in many cases as "*performance-destroying memory locking*". In the paper authors describe that processing of gradients on workers is very efficient but updating the parameters with centralized unit (both in shared memory and parameter server) brings big slowdowns because of waiting periods.

By improving the updating scheme they achieve an impressive speed-up. The idea of Hogwild! is to use shared memory and update the parameters without any hardware or software memory locks. When the worker finishes calculating the gradient and is ready to update, it can update the central parameters without waiting[1].

[1] In most cases. Unless it is not atomic and therefore it is implemented with a compare-exchange instruction, which can fail.

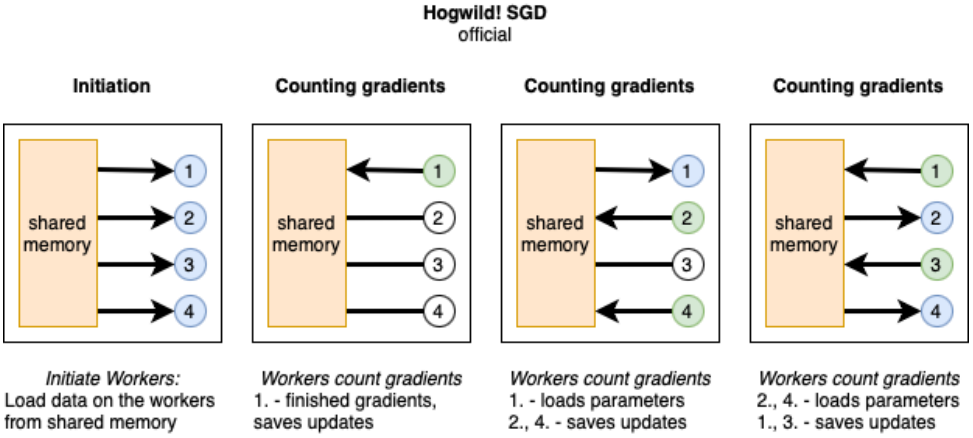


Figure 4.2: Hogwild! algorithm displayed on minimal run.

In the Figure 4.2 we have shown in a single iteration[2] on all workers that updates are not bound to one single worker at a time. The blue worker nodes are loading the parameters from shared memory, after loading they become the white worker nodes, while they are calculating the gradients. After finishing, they become the green worker nodes and save the newly updated parameters straight into shared memory.

In this method there is no waiting. Neither for parameters, nor for gradients. But this solution works only on some very restrictive conditions. The Hogwild! is described as working well under two conditions:

The first condition. is that the problem must be a sparse separable cost function (graph cuts and similar problems), where update of the parameters targets only a small subset of parameters.

$$\theta = \sum_{e \in E} \theta_e \tag{4.3}$$

where e is subset of E and θ_e is vector containing a single parameter from of all parameters in the vector θ on the coordinate of e.

The second condition. is that the update of single parameter has to be an atomic operation. The authors mention that on some CPUs and GPUs

[2]With exception of worker 1 doing 2 iterations, all others finished 1 iteration.

it is implemented and on other processing units it may be done using a compare-exchange instruction.

$$\theta_i \leftarrow \theta_i - a_i \tag{4.4}$$

where a_i is some coefficient from equation $\theta_{t+1,i} = \theta_{t,i} - \eta b_i G_i$. The b_i is a vector with coefficients in the interval $\{0, 1\}$, having only the elements with index $i \in e$ equal to 1 and the elements with index $i \notin e$ equal to 0. The G_i is a gradient of $\theta_{t,i}$. The η is the step size of algorithm.

Processors are all granted equal access to shared memory. It is possible to overwrite the previous value but in the sparse problem it is shown in the paper that it doesn't happen that often and therefore Hogwild! achieves a nearly optimal convergence rate. The authors display a nearly linear speedup based on the number of processors on sparse learning problems.

Algorithm 4: Hogwild! algorithm on Workers

```
@everywhere function train_Hogwild!(loss, ps, data, opt;
      cb =()->(),max_iteration =100)
   t =0 # iteration
   while iteration <max_iteration
      e =load_random_sample(E)
      ps =load_current_parameters(e)
      batch_data =random_minibatch()
      gs, y =calculate_gradient(loss, ps, batch_data...)
      for i in e
         save_parameters_to_memory(ps, G_i)
      end
      t +=1 # iteration + 1
   end
end
```

As the Hogwild! algorithm is a local asynchronous parallel method, it only has the workers and no server node. The results can be returned by the last finishing worker.

This algorithm is not distributed and will not be implemented in this thesis.

4.4 Downpour SGD[FBG+17]

The Downpour SGD is an asynchronous distributed algorithm. The authors have aimed to fix the problems from both of the previously mentioned algorithms - Bulk Synchronous SGD[RUL10] in Section 4.2 and Hogwild! SGD[RRWN11] in Section 4.3.

The Hogwild! SGD[RRWN11] is an asynchronous parallel algorithm, but has many constrains, like only being able to solve sparse problems and working in the shared memory model. Creating distributed shared memory is considered to be impossible. To rewrite the Hogwild! algorithm into a distributed algorithm with shared memory scheme would lead to overwhelming synchronization.

On the other hand, the Bulk Synchronous SGD[RUL10] algorithm takes a centralized, distributed a d synchronous approach. It has a server in the central unit that is solving the problem of loading the gradients from the workers and updating the model parameters. Yet the necessity of synchronization means that there are long pauses introduced when waiting for the gradients from all the workers.

To allow a distributed run of the Hogwild! it is necessary to switch from a Shared memory model to a Message Passing system. In this case the problem with shared memory is reduced by creating a single central unit that acts like shared memory, just like the Bulk Synchronous SGD. Downpour is generally slower than the Hogwild! algorithm, as the message has to be passed from the worker to the server. Communication over the network instead of simply saving into memory generates huge overhead. Although the saving times are much bigger, the Downpour algorithm has the ability to scale to multiple computers.

Difference in the updating of parameters is that it is calculated in every single worker-server communication. Unlike the Bulk Synchronous SGD, where it is done in each iteration for all workers (with mean gradients), and in Hogwild!, where it is written straight to memory with sparse parameters at any time (with atomic updates in memory).

In the Figure 4.3 we show a single iteration on all workers where the server collects the gradient from a worker and updates the parameters. The algorithm starts with loading the parameters to the workers, shown as blue worker nodes. Then the workers change the color to white, where they

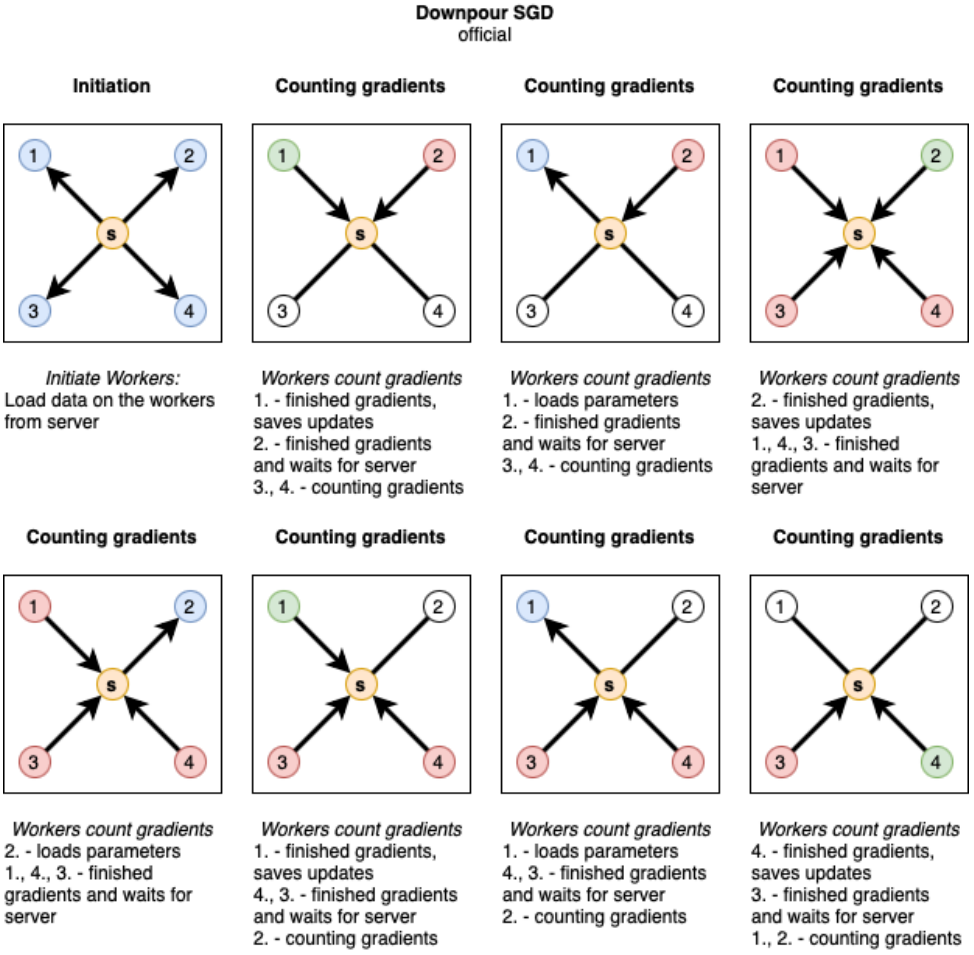


Figure 4.3: The Downpour algorithm displayed on a minimal run.

calculate the gradients. Finished worker nodes are then shown in green, when saving the newly calculated gradients to the server, or in red, when waiting to send the gradients to the server. This method has still some waiting periods. The only type is where the workers are ready to send the gradients (red) and the server node is busy communicating with a different node.

The server receives the request, updates the global model parameters and sends the global model parameters back to the worker. Worker code is the same as in Bulk Synchronous SGD method in Algorithm 3. In this thesis, we chose the approach where each update of parameters means a new iteration, but it can be changed to a different approach, eg. the approach where a new iteration begins only after all the workers finished the iteration (in a more synchronous fashion).

Algorithm 5: Downpour algorithm on Server

```
@everywhere function train_Downpour!(loss, ps, data, opt,
      server_channel; cb =()->(),max_iteration =100)
   t =0 # iteration
   while t <max_iteration &&fetch(server_channel)
      j, gs_j =load_grads_from_worker(server_channel)
      update!(opt, ps, gs_j)
      send_params_to_worker(j, ps)
      t +=1 # iteration + 1
   end
end
```

4.5 Elastic Averaging SGD[CPM+16]

While studying the problem of stochastic gradient optimization in a distributed computing environment under communication constraints, the authors found that the overhead caused by the amount of communication with the centralized unit is the greatest problem. To decrease the amount of communication, they used the technique called Elastic Averaging. Communication with the server only happens in τ iterations, instead of every iteration. They call period of each τ iterations, containing the communication iteration, is a single *clock cycle*.

The authors' base algorithm, the *Elastic Averaging Stochastic Gradient Descent*, on equivalence of two equations for minimizing function called the *Global Variable Consensus* or *Augmentability* in the literature. The worker in the elastic averaging method is calculating the gradients and updating the local parameters.

The frequency of communication between every worker and server is controlled by the term τ. The term τ is regarded as *communication period* in the paper.

In the paper, the authors describe the magnitude of ρ to represent the amount of exploration that is allowed in the model. To allow more exploration in the local workers, the term ρ can be decreased and therefore allow θ^i to fluctuate further from the central parameter $\tilde{\theta}$. The idea of Elastic Averaging SGD is described as allowing the local workers greater exploration and the

server to perform the exploitation, which is different from the main focus in the literature to find the convergence of center parameters.

$$\text{elastic}_t^i = \eta\rho(\theta_t^i - \widetilde{\theta}_s) \tag{4.5}$$

In Formula 4.5 we define elastic difference used in this method.

4.5.1 Elastic Averaging SGD - synchronous

Authors define Synchronous Elastic Averaging SGD algorithm with update steps for worker (in Formula 4.6) and update step for server (in Formula 4.7).

$$\theta_{t+\tau}^i = \theta_t^i - \eta(g_t^i(\theta_t^i) + \rho(\theta_t^i - \widetilde{\theta}_t)) \tag{4.6}$$

$$\widetilde{\theta}_{t+\tau} = \widetilde{\theta}_t - \eta\sum_{i=1}^{p}\rho(\theta_t^i - \widetilde{\theta}_t) \tag{4.7}$$

where $\widetilde{\theta}_t$ denotes the server parameters, also called center parameters, and the θ_t^i are the parameters on worker i, also called the local parameters. The term η is a learning rate of Stochastic Gradient Descent and the term ρ is quadratic penalty that ensures that local workers do not fall into the attractors that are far away from the server parameter.

The elastic averaging introduced in the paper the simplifying variables $\alpha = \eta\rho$ and $\beta = p\alpha$. The authors call the α moving rate and that β leads to an elastic symmetry in the update rule. Further more authors describe that the update rule of the center parameters $\widetilde{\theta}_t$ is taking the form of moving average over space and time.

$$\theta_{t+\tau}^i = \theta_t^i - \eta g_t^i + -(\theta_t^i) + \alpha(\theta_t^i - \widetilde{\theta}_t)) \tag{4.8}$$

$$\widetilde{\theta}_{t+\tau} = (1-\beta)\widetilde{\theta}_t + \beta\left(\frac{1}{p}\sum_{i=1}^{p}\theta_t^i\right) \tag{4.9}$$

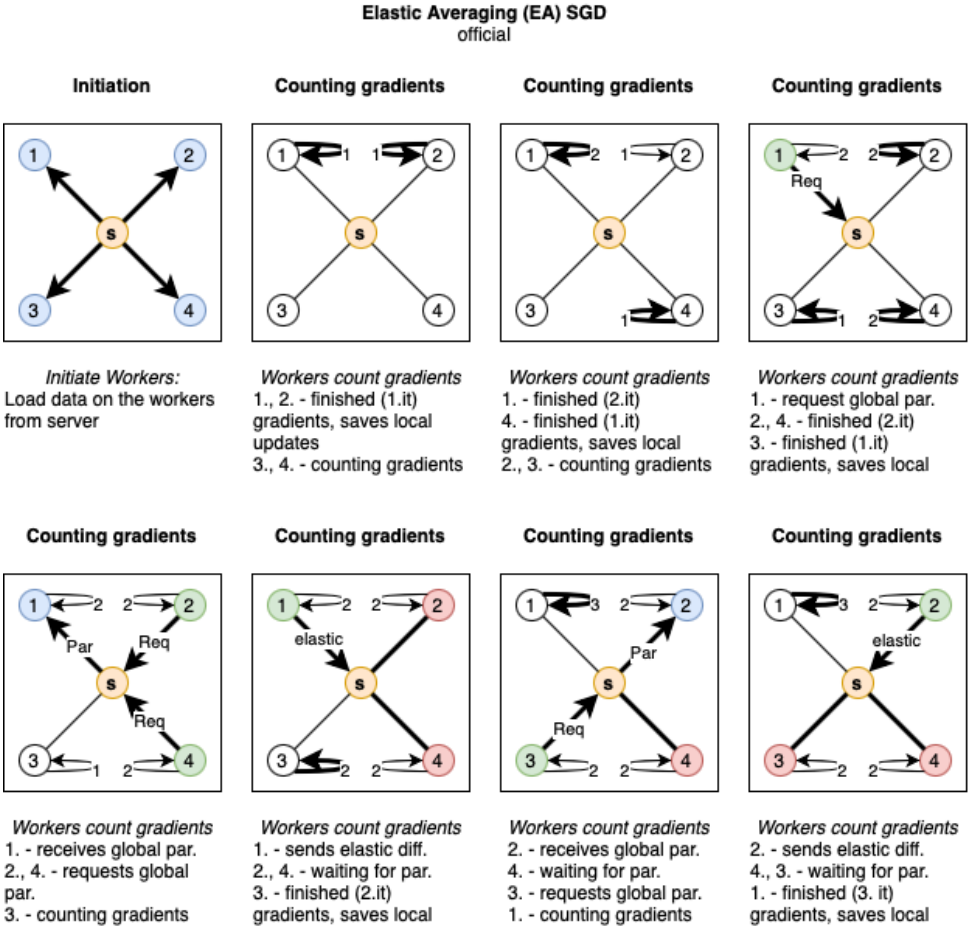


Figure 4.4: Elastic Averaging algorithm displayed on minimal run with $\tau = 2$.

In Synchronous Elastic Averaging algorithm there is no difference in iteration of global parameters and the local parameters and therefore $s = t$.

The single iteration t on the worker i consists of having local parameters θ_t^i from previous iterations, calculating the gradient and updating the local parameters. In contrast the iteration on worker being in communication period consists of receiving the center parameters $\widetilde{\theta}_t$, having its local parameters $\theta_{t+\tau}^i$ and calculating the elastic difference (in Formula 4.5) for next iteration of both local parameters θ_{t+1}^i and global parameters. After calculating the worker sends the elastic difference back to the server node. The server sums all of the elastic differences, creates mean elastic difference and subtracts it from the global parameters creating next server iteration $\widetilde{\theta}_{t+\tau}$ (in Formula 4.7).

We classify this algorithm as Semi-synchronous centralized SGD algorithm.

Algorithm 6: Synchronous Elastic Averaging algorithm on Server

```
@everywhere function train_EA_sync!(loss, ps, data, opt,
      server_channel; cb =()->(),max_iteration =100)
   t =0 # iteration
   while t <max_iteration
      finished_workers =[]
      summary =empty_params_vec(ps)
      while finished_workers <nworkers()
            &&fetch(server_channel)
         j, Δθ_{t,j} =delta_from_worker(server_channel)
         push!(finished_workers, j)
         summary .+=θ_{t,j}
      end
      summary /=nworkers()
      update!(opt, ps, -summary)
      send_params_to_workers(ps)
      t +=τ # iteration + τ
   end
end
```

4.5.2 Elastic Averaging SGD - asynchronous

As in Synchronous EASGD the authors define Asynchronous Elastic Averaging SGD algorithms update steps. For worker it remains the same (in Formula 4.6), but for server the change is in update by single value (in Formula 4.10) instead of mean value.

$$\widetilde{\theta}_{s+\tau} = \widetilde{\theta}_s - \eta\rho(\theta_t^i - \widetilde{\theta}_s) \tag{4.10}$$

where $\widetilde{\theta}_s$ denotes the server parameters having different iteration s, where $s \geq t$, from worker i, having local parameters θ_t^i on iteration t.

Update the global parameters in every communication the server is rapidly progressing in the amount of iterations.

As in the synchronous version, a single iteration t on the worker i consists of having local parameters θ_t^i from previous iterations, calculating the gradient

Algorithm 7: Asynchronous Elastic Averaging algorithm on Server

```
@everywhere function train_EA_async!(loss, ps, data, opt,
      server_channel; cb =()->(),max_iteration =100)
   t =0 # iteration
   while t <max_iteration &&fetch(server_par_channel)
      j =request_from_worker(server_par_channel)
      send_params_to_worker(ps, j)
      j, Δθ_{t,j} =delta_from_worker(server_channel)
      update!(opt, ps, -Δθ_{t,j})
      t +=τ # iteration + τ
   end
end
```

and updating the local parameters. During communication period iteration worker i requests the global parameters from server $\widetilde{\theta}_s$, after receiving them it calculates elastic difference and in Formula 4.5 and sends it to server. Server after receiving the elastic difference from worker i subtracts it from the global parameters creating next server iteration $\widetilde{\theta}_{t+\tau}$ (in Formula 4.10).

We classify this algorithm as Semi-asynchronous centralized SGD algorithm.

4.6 Stale Synchronous Parallel SGD

The Stale Synchronous Parallel (SSP) model is a partially asynchronous centralized system. It is partially asynchronous as it has staleness threshold. By bounding the staleness threshold, limiting maximum iteration difference of the parameters on all the workers, the algorithm achieves faster processing. The amount of communication decreases between the server unit and the worker units proportionally. While using older, stale, versions of parameters from local cache instead of communicating the latest data from central unit the workers spend less time waiting and transferring the data and more time calculating.

The Stale Synchronous Parallel SGD method is generalizing the Bulk Synchronous Parallel SGD method, which is strongly synchronous. Partially removing the staleness of synchronization introduces less waiting and therefore allow for more processing during the same time period. Instead of removing

the staleness completely the Stale Synchronous Parallel method only increases the staleness difference from 0, meaning completely synchronous as all the workers are on the same iteration, to some number of iterations m.

This m-iteration staleness in turn has implication that the parameters used are not in complete disarray. It allows for more fluent run on multiple workers as the slower workers can catch-up, but faster workers don't have to stop and wait. The wait may be eventually introduced if some worker gets stuck or the staleness factor is too small, like in the Bulk Synchronous m equal to 0.

The SSP computation method is targeting the processing with heavy memory usage. It works with thread and process caches. This method is very hard to implement correctly, with tracking of the iterations and two local caches based on iterations. It was not selected to be implemented.

4.7 Synchronous All-Reduce SGD

The initial implementation of the synchronous decentralized stochastic gradient descent method is called Synchronous All-Reduce SGD. It is the Bulk Synchronous decentralized counterpart. The name is self-explanatory as it is based on the type of synchronization of SGD and it's communication method between the workers, called All-Reduce.

The All-Reduce communication is presumed to be $O(n)$, which it is on some networks like ring, but when using different network it may be better. The best case asymptotically is hypercube. On hypercube All-Reduce method has asymptotic complexity equal to $O(log(n)$.

In the Figure 4.5 we show random single iteration on all workers in the hypercube network. They gradually collect the gradients from other workers and update the local parameters. The algorithm starts with loading the parameters to the workers, shown as blue worker nodes. Then the workers calculate the gradients and change the color to white. Finished worker nodes are then sending (green), or receiving (blue) the newly calculated gradients to the first-bit neighbours (written as $g(i)$). There may happen the waiting period, as the neighbouring worker may not finish and rest of the workers must wait[3]. When worker aggregated the $g(i)$ they send the new $g(i, j)$ to second-bit neighbours. The situation is repeated for the $log(n)$ step, if there is

[3]This situation with waiting may happen but is not shown in the Figure 4.5

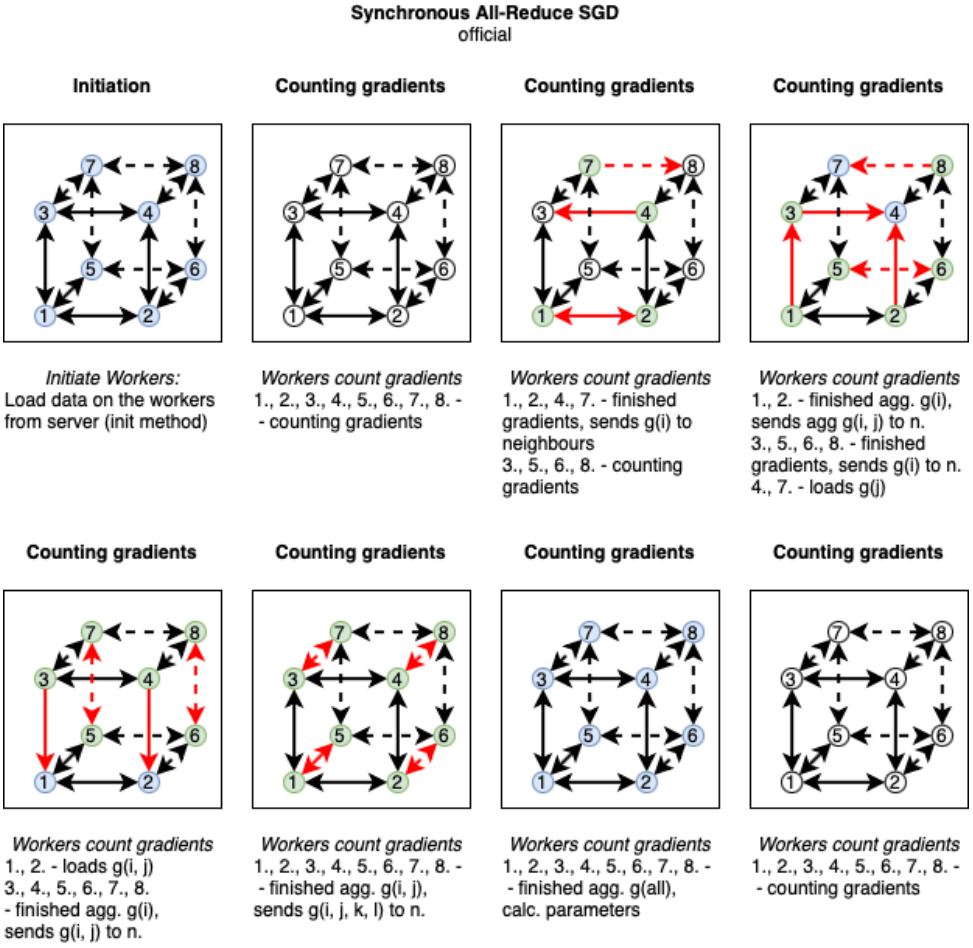


Figure 4.5: Synchronized All-Reduce algorithm displayed on minimal run.

n workers. When all the gradients are received on the worker it then updates the local parameters and starts new iteration.

We classify this algorithm as Synchronous decentralized SGD algorithm. This method was selected to be implemented and analysed. During the analysis of the smaller dataset the method displayed incorrect behaviour and was not added to the methods to be analysed on the big dataset.

4.8 Gossiping SGD[JYIK16]

The centralized network had problem with the over-usage of the central unit which is solved by switching to decentralized network having no central unit.

But all-reduce step distributing its gradient to all workers and collecting gradients from every worker in every iteration is even more costly with regard to network overhead[4]. This solution does not overwhelm the server unit, but the amount of communication (even though optimised from n to $log(n)$ in single iteration for one worker) is too massive. All-Reduce solution generates slowdowns with the amount of data communication and additionally by waiting on slow workers synchronization.

To make the decentralized network solution usable it is necessary to decrease the amount of communication with the other workers. This means using some simplification of communication. The authors used *gossip aggregation algorithm*[KDG03] as fundamental building block. They describe the previous gossiping algorithm as synchronous and introduce the asynchronous version.

The authors conceptually link gossip to elastic averaging algorithm and therefore the gossiping can decrease the amount of communication of the workers by communicating only in τ iterations. Further more the authors describe two Gossiping techniques: one is Pull and the other is Push technique. Both techniques are heavily linked with the Elastic Averaging.

Pull Gossiping SGD. The authors describe Pull Gossiping as an equivalent to "pull gossip" method. There every worker node receives only one parameter per iteration. The general idea is that in communication iteration every worker can "pull" or request exactly one other workers data. Therefore each worker has exactly 2 averaged parameter data (itself and random other worker). This is called "one-node estimator".

Push Gossiping SGD. The authors introduced the "Push" version of Gossiping Stochastic Gradient Descent. Idea behind this method is that every worker node "pushes" or sends its parameter data only once instead of being queried any number of times. The worker sends their solution and receives any number of parameter data from other workers. The minimum is 1 and maximum is p parameter data to be averaged.

We classify both of these algorithms as Semi-asynchronous decentralized SGD algorithms. Push Gossiping method was selected to be implemented and analysed. Unfortunately during the analysis of the smaller dataset the method displayed incorrect behaviour and was not added to the methods to be analysed on the big dataset.

[4]Based on amounts of gradients transferred in the network

Part II

Implementation

Chapter 5

Experimental results

In this chapter we will first describe the setup for the experiments. It will consist of performance metrics, by which the algorithms will be compared, then experimental details, where will be more information about algorithm setup and about selected datasets for this experiment.

In second part of this chapter we will run the algorithms and use the performance metrics for the comparison. Details of the runs will display the results in form of graphs.

5.1 Performance metrics

Measuring performance of algorithms in distributed systems is not trivial. This is even more true for algorithms for analysis, like stochastic gradient descent in this thesis.

Problems are: selection of algorithm parameters (like learning rate, exploration rate, or communication period), loading datasets onto the workers (as the amounts of data are huge and it means transferring these over the network), setting basic definitions for algorithms with different types of communication and different amounts of communication periods (e.g. definition of iteration for all algorithms) and measurements of time (as the logging measurements take time, saving logs takes time), etc.

We have selected metrics that we think are the most valid in the context.

Measuring the loss is closest to the accuracy without huge hit to the performance of the algorithm. This performance metric is efficient in case of comparison as all the algorithms will be working on same problem with same resources and very similar setup. It displays how fast is algorithm progressing in term of iterations as well as time.

This metric even allows for closer comparison of the algorithm itself, in case it has multiple parameters to set up. Therefore we will first compare the algorithm with itself and display how a choice of different parameters is changing the progress and final convergence and speed of processing.

5.2 Experimental datasets

Selection of datasets for the experiment was very restrictive as the datasets had to be very big, very complex and having very big amount of model parameters to allow for more space to show the differences in this experiment.

All of the selected datasets were from security domain. These datasets were recorded from data before 2018 and are being used in the research of malware. The selected datasets had to be pre-processed into a unifying hierarchical data structure to be later used for learning using *Hierarchical Multiple Instance Learning* (HMIL). Hierarchical Multiple Instance Learning uses the structure of a samples in dataset and reflects the structure in the models. Using this type of learning from instances has added value in

5.2.1 Elastic Malware Benchmark for Empowering Researchers (EMBER)

First dataset to be selected was Elastic Malware Benchmark for Empowering Researchers, called EMBER[AR18]. It is huge dataset that is very well known. The 2017 and 2018 EMBER datasets contain features from more than 1 million PE files scanned in or before their respective years.

Thanks to my advisors doc. Ing. Tomáš Pevný, Ph.D. contribution of his

prepared Ember dataset in *json* format and the serialized data in Julia *jls* format we were able to use the EMBER dataset using HMIL model.

This dataset was selected as main experimental dataset having all the parameters we required.

5.2.2 Android Malware Dataset (CIC-AndMal2017)

We have selected the second dataset to be Android Malware Dataset, called AndMal2017[LKTG18] or CIC-AndMal2017. This dataset was proven to be hard problem. This dataset is in form of *pcap* files containing network packet data. These types of data are used for network analysis. This dataset contains 4 main categories - Adware, Ransomware, Scareware, SMS Malware. Every category contains more than 5 subcategories, with different families of malware in respective category.

Thanks to Bc. Vojtěch Kozel[Voj21] who was very kind and contributed the AndMal2017 data in *json* format, we were prepared to use them in our experiment. This dataset was selected as secondary experimental dataset and was prepared to be analysed.

Due to problems while processing the primary dataset the secondary experiment was postponed and will not be part of this thesis.

5.3 Experimental details

In this section we first describe the setup of the experiment, the code used to run the implementation and the server it is run on. Later we run all of the selected algorithms with different parameters. We compare the progress and results and discuss the details of the experiment preparation for each of the algorithms used.

The algorithms will be run on smaller dataset to allow for faster processing and more analysis. This test dataset is small but still very complex and will display the differences in very good detail for comparison after few iterations.

5.3.1 Hardware setup of experiment

Computer running the experiment had the configuration:

- AMD EPYC 7282 2.8 GHz (8 Physical Cores with Hyper-threading)
- 64 GB RAM
- 480 GB NVMe SSD
- 1 Gbit/s Port

Server was hosted by German company called Contabo, using the best virtual dedicated server company has on offer VDS-XL.

5.3.2 Bulk Synchronous SGD

Bulk Synchronous algorithm is simple and has few parameters that may change the performance.

The first parameter setting the speed of processing is a minibatch size. In this algorithm the minibatch will set the communication performance as it may greatly increase or decrease waiting on workers based on how hard is problem, in terms of computation, compared to other workers problems. This method especially struggles with slow workers.

Second parameter is the number of workers. The number of workers has to be regulated to not overwhelm server, in our experiment environment the maximum is 8 workers which is not problematic. Second parameter has two subproblems: Are all workers stable? Do all workers have same hardware? The response to both questions is in our environment yes.

Third parameter is learning rate. Change in learning rate affects how fast the algorithm converges as well as in basic Gradient Descent method.

5.3.3 Downpour SGD

Downpour algorithm is simple and has the same parameters as Bulk Synchronous SGD. This method however does not need stable[1] workers and does not need the same hardware for workers. Even the first parameter, the size of minibatch, is a smaller problem compared to the Bulk Synchronous. If the minibatch is huge then the server will wait for the first finished worker. It does not suffer from struggling workers.

Change in the learning rate does affect how fast the algorithm converges the same as in basic Gradient Descent method.

5.3.4 Elastic Averaging SGD

This method has a lot of parameters and therefore has much harder setup preparations. The previously mentioned parameters: size of minibatch and learning rate affects this method. The slow workers have big effect on Synchronous version of the Elastic Averaging with Asynchronous version having similar solution as Downpour for this problem[2].

Additional parameters are exploration and the communication period. For these parameters it is necessary to run few[3] iterations to find the best starting. In this case the amount was 100 iterations on the smaller dataset (hundreds of Megabytes).

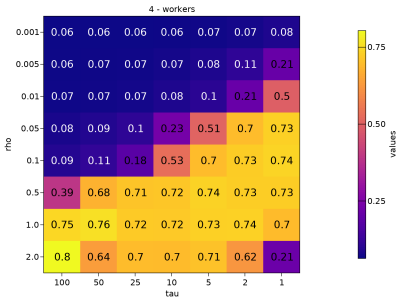


Figure 5.1: Gradient Descent $\eta = 0.5$

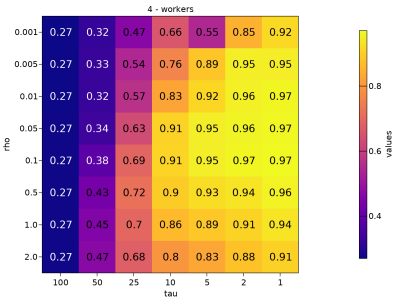


Figure 5.2: ADAM $\eta = 0.05$

[1] It will recover from failing workers
[2] Solution is the communication type - the communication with single worker
[3] Amount required is determined based on the complexity of the problem

In the Figures 5.1 and 5.2 we search for best combination of parameters τ and ρ in Elastic Averaging algorithm. The algorithm is run on eight workers with optimization method the Gradient Descent. The final values in the heatmaps are the accuracy. In just 100 iterations we can see that the Gradient Descent progresses fast enough on this problem with defined learning rate.

5.3.5 Synchronous All-Reduce SGD

This Stochastic Gradient Descent method is very similar in requirements to Bulk Synchronous SGD. As mentioned in the previous chapter this method was implemented but due to the problems with convergence but was not tested on experimental data.

5.3.6 Gossiping SGD

The Gossiping method is very similar in requirements to Elastic Averaging SGD in respective communication types. As mentioned in the previous chapter this method was implemented but due to the problems with convergence but was not tested on experimental data.

5.4 Experiment preparation

As the results of computing the stochastic gradient descent are dependent on randomness of samples we wanted to have the results reproducible and comparable. For better comparability we have set the same parameters to all algorithms. The initial Gradient Descent method didn't show much progress and therefore we have selected better performing ADAM for all of the runs.

In the Figures 5.3 and 5.4 we have analysed both Elastic Averaging algorithms on EMBER dataset to find the best possible additional parameters. After 500 iterations we can see that the Gradient Descent does not progress fast enough on this problem with defined learning rate.

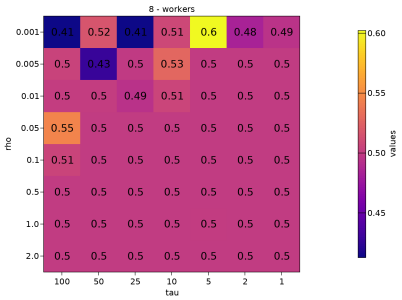


Figure 5.3: Gradient Descent with 0.5 - Asynchronous Elastic Averaging

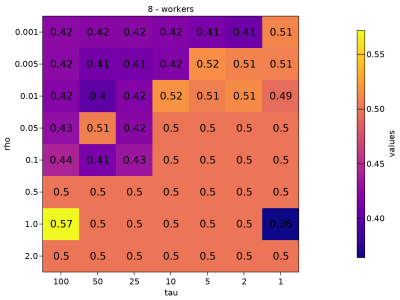


Figure 5.4: Gradient Descent with 0.5 - Synchronous Elastic Averaging

Based on the results we have selected the learning rate to be ADAM instead of basic Gradient Descent. The individual experiment runs were seeded based on worker ids. Parameters were set to:

sample size 200

learning rate η based on ADAM with $\eta = 0.001$ and decay of momentums $\beta = (0.9, 0.999)$[4]

amount of workers p 8

communication period τ 10 iterations

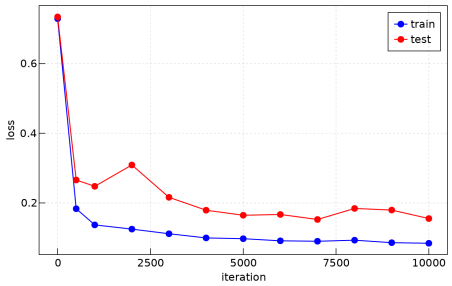
exploration amount ρ 0.05

5.5 Experiment results

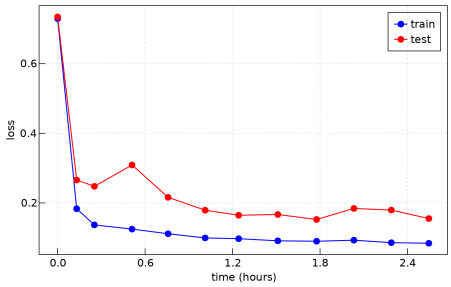
We are comparing selected implementations of algorithms to the state-of-the-art algorithm named Flux shown in Figure 5.5.

Selected algorithms are shown in their respective figures. The Bulk Synchronous is shown in Figure 5.6. The Downpour is shown in Figure 5.7. The Synchronous Elastic Averaging is shown in Figure 5.8 and the Asynchronous Elastic Averaging is shown in Figure 5.9.

[4]In the Flux the definition is $ADAM(\eta = 0.001, \beta :: Tuple = (0.9, 0.999))$



Loss in Iterations



Loss in Time

Figure 5.5: Line plots of 10.000 iterations of Flux algorithm on Ember

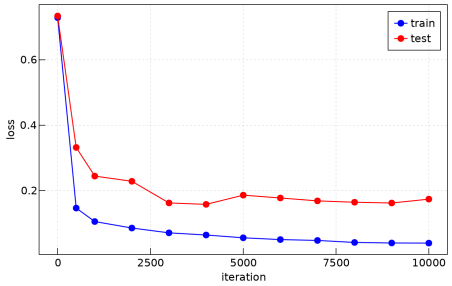


Figure 5.6: Bulk Synchronous: Loss/Iterations

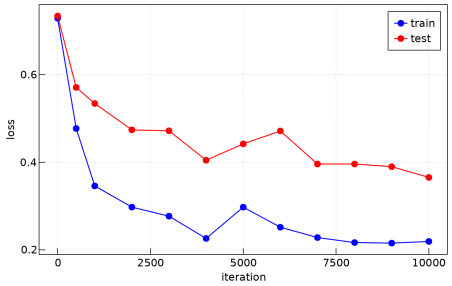


Figure 5.7: Downpour: Loss/Iterations

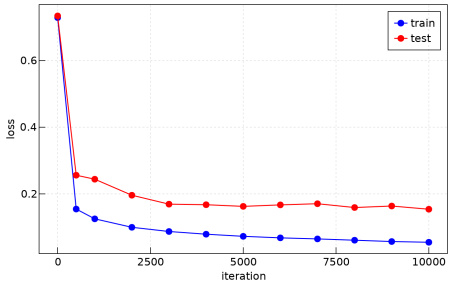


Figure 5.8: Synchronous Elastic Averaging: Loss/Iterations

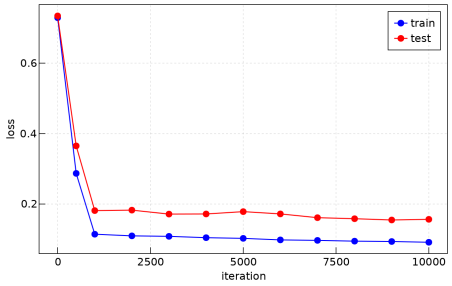


Figure 5.9: Asynchronous Elastic Averaging: Loss/Iterations

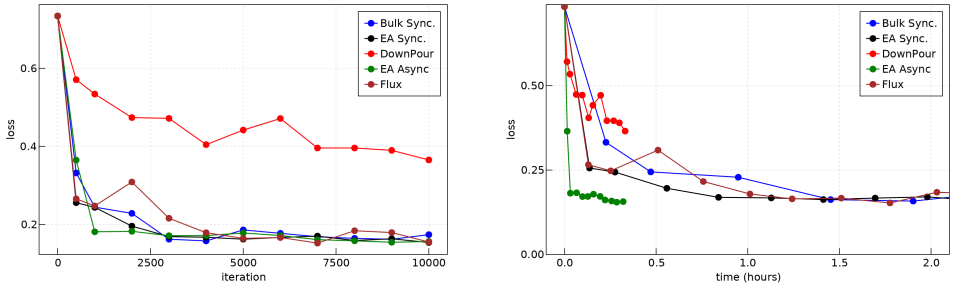
5.5.1 Results summary - 10.000 iterations

Summary of measurements is in the the Table 5.1 with the graphical version of losses shown in Figure 5.10 and the accuracy shown in Figure 5.11.

The best accuracy in 10.000 iterations was by state-of-the-art solution Flux.

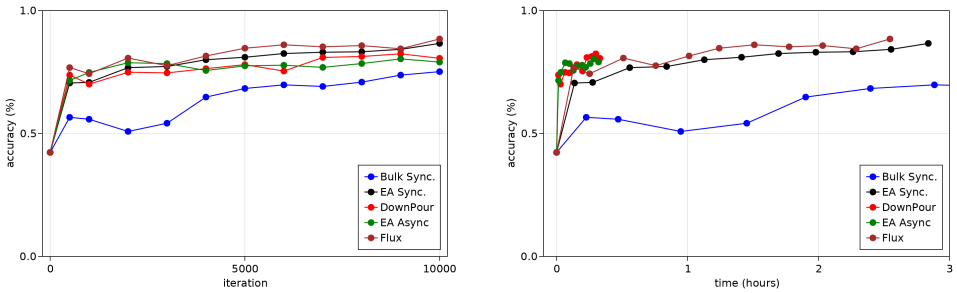
Algorithm	Train loss	Test loss	Accuracy (%)	Time (hod)	Speed-up
Flux	0.084	0.155	88.476	2.54	1
Bulk Sync.	0.039	0.174	75.183	4.85	0.52
Downpour	0.219	0.365	80.618	0.33	7.70
EA sync.	0.054	0.154	86.667	2.83	0.90
EA async.	0.091	0.156	79.077	0.32	7.94

Table 5.1: Comparison of final model after 10.000 iterations on Ember dataset



Loss in Iterations Loss in Time

Figure 5.10: Line plots of 10.000 iterations of all algorithms on Ember



Accuracy in Iterations Accuracy in Time

Figure 5.11: Line plots of 10.000 iterations of all algorithms on Ember

It did converge to almost 88.5% accuracy in around two and a half hours. Little smaller accuracy was achieved by Synchronous Elastic Averaging with almost two hours and fifty minutes. The Bulk Synchronous displayed the worst results both in terms of accuracy as well as processing time. The problem of Bulk Synchronous algorithm was the communication every iteration with batch of size 200, where some workers struggled. On the other side very impressive processing time with great accuracy was achieved by Downpour with the Asynchronous Elastic Averaging right behind in terms of accuracy.

In terms of loss the worst was for both cases of training and testing datasets the Downpour algorithm. The other algorithms were all very close with exception to the Bulk Synchronous. Bulk Synchronous had very small final training error which could potentially mean overfitting of the model and

it would justify the smaller accuracy in comparison to the other algorithms.

The synchronous algorithms displayed slowdowns, but the asynchronous algorithms were very fast. In just 20 minutes both impressed with speed-up almost 8 times the speed of processing. That is close to linear speed-up. This almost linear speed-up in centralized system was achieved by using asynchronous communication, that is a communication worker-to-server and server-to-worker instead of all workers-to-server and server-to-all workers. This decreases the number of workers participating in single iteration from all, in this experiment eight workers, to only one worker. Even better speed-up could be achieved on Elastic Averaging methods by selecting communication period $\tau > 10$.

The parameters were set for all of the algorithms before the actual run on the EMBER dataset. Only tuning was the selection of the parameters for Elastic Averaging based on the heatmaps for the parameters that are algorithm specific. By tuning the parameters based on the requirements of algorithms instead of setting the universal values for the algorithms the results would be very different.

5.6 Results summary - time

The summary of measurements is in the the Table 5.2 with the graphical version of losses shown in Figure 5.12 and the accuracy shown in Figure 5.13.

Algorithm	Train loss	Test loss	Accuracy (%)	Time (hod)
Flux	0.084	0.155	88.476	2.54
Bulk Sync.	0.039	0.174	75.183	4.85
Downpour	0.148	0.355	84.754	2.73
EA sync.	0.054	0.154	86.667	2.83
EA async.	0.051	0.138	87.435	2.59

Table 5.2: Comparison of final model after 10.000 iterations on Ember dataset

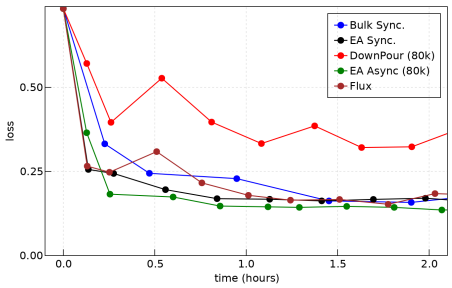


Figure 5.12: Loss in Time

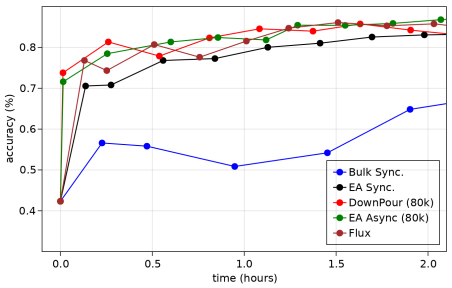


Figure 5.13: Accuracy in Time

When all algorithms are given the same time, e.g. 2 hours as in Figure 5.12, the asynchronous algorithms achieve similar results as state-of-the-art method Flux.

At the end of the algorithms training Downpour and both Elastic Averaging algorithms are comparable to Flux in regard to accuracy as well as in regard to processing time.

Chapter 6

Conclusion

The goal of this thesis was to implement state-of-the-art algorithms for distributed training of any machine learning algorithm with loss being expressed as in Formula 2.2, where y_i can be empty. The algorithms then were to be compared on training hierarchical multiple instance models. Emphasis was put on improving processing speed as well as convergence. As this may not be possible the resulting model should be with minimal decrease in convergence.

We have started by setting main principles of data analysis with emphasis on stochastic gradient descent and machine learning. Next we have described the concurrency with emphasis on the distributed networks and then pointed out the prior art in distributed stochastic gradient. The research in these chapters explains the problems. After carefully selecting the algorithms we have implemented the algorithms and tested them EMBER dataset.

The algorithms that were implemented and tested were all centralised with each being in the different communication type. The centralised synchronous algorithm - Bulk Synchronous, centralised asynchronous algorithm - Downpour, centralised semi-synchronous algorithm - Synchronous Elastic Averaging and centralised semi-asynchronous algorithm - Asynchronous Elastic Averaging.

Two algorithms, synchronous decentralised algorithm - All-Reduce SGD and semi-asynchronous decentralised algorithm - Push Gossiping, were implemented but omitted from analysis due to the problems with convergence.

All of the implemented algorithms shown promise, but need more extensive parameter tuning to perform better. Some implemented algorithms, Downpour and Asynchronous Elastic Averaging, have achieved almost linear speed-up with small impact on convergence in comparison to state-of-the-art implementation of Flux. Additional exploratory analysis shows that especially Asynchronous Elastic Averaging may converge even faster.

The algorithms are created as part of the library for Julia ecosystem and will be released soon after the release of the thesis.

6.1 Future work

By creating the open-source library this thesis opens possibility for speeding-up the process of research based on machine learning methods using Gradient Descent. This is not strictly limited to using only Julia programming language[1] but to any language.

In this thesis we have created the library improving state-of-the-art run times with room for improvements based on the standards of Julia projects. Additional improvements in the future may include adding more methods in this library, with algorithms using momentum being an example, or implementing the automatic research of algorithm parameters for debugging purposes. Other improvements for the library may be by creating more stable environment using automatic building with test cases and documentation with code base.

[1]The library may be used in any programming language as Julia programming language is embeddable into another languages, like Python.

Appendices

Appendix A

Bibliography

[AR18] H. S. Anderson and P. Roth, *EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models*, ArXiv e-prints (2018).

[CPM+16] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz, *Revisiting distributed synchronous sgd*, arXiv preprint arXiv:1604.00981 (2016).

[FBG+17] Lex Fridman, Daniel E Brown, Michael Glazer, William Angell, Spencer Dodd, Benedikt Jenik, Jack Terwilliger, Julia Kindelsberger, Li Ding, Sean Seaman, et al., *Mit autonomous vehicle technology study: Large-scale deep learning based analysis of driver behavior and interaction with automation*, arXiv preprint arXiv:1711.06976 **1** (2017).

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, MIT press, 2016.

[GKGK03] Ananth Grama, Vipin Kumar, Anshul Gupta, and George Karypis, *Introduction to parallel computing*, Pearson Education, 2003.

[Inn18a] Michael Innes, *Don't unroll adjoint: Differentiating ssa-form programs*, arXiv preprint arXiv:1810.07951 (2018).

[Inn18b] Mike Innes, *Flux: Elegant machine learning with julia*, Journal of Open Source Software (2018).

[ISF+18] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali,

Avik Pal, and Viral Shah, *Fashionable modelling with flux*, CoRR **abs/1811.01457** (2018).

[JYIK16] Peter H Jin, Qiaochu Yuan, Forrest Iandola, and Kurt Keutzer, *How to scale distributed deep learning?*, arXiv preprint arXiv:1611.04581 (2016).

[KDG03] David Kempe, Alin Dobra, and Johannes Gehrke, *Gossip-based computation of aggregate information*, 44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings., IEEE, 2003, pp. 482–491.

[Lem12] Claude Lemaréchal, *Cauchy and the gradient method*, Doc Math Extra **251** (2012), no. 254, 10.

[LKTG18] Arash Habibi Lashkari, Andi Fitriah A Kadir, Laya Taheri, and Ali A Ghorbani, *Toward developing a systematic approach to generate benchmark android malware datasets and classification*, 2018 International Carnahan Conference on Security Technology (ICCST), IEEE, 2018, pp. 1–7.

[LLPS93] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken, *Multilayer feedforward networks with a nonpolynomial activation function can approximate any function*, Neural Networks **6** (1993), no. 6, 861–867.

[OS13] Cathy O'Neil and Rachel Schutt, *Doing data science: Straight talk from the frontline*, " O'Reilly Media, Inc.", 2013.

[RRWN11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu, *Hogwild: A lock-free approach to parallelizing stochastic gradient descent*, Advances in neural information processing systems, 2011, pp. 693–701.

[RUL10] Anand Rajaraman, Jeffrey Ullman, and Jude Leskovec, *Mining of massive datasets*, Stanford University Press, 2010.

[Voj21] Kozel Vojtěch, *Hierarchické modely sít'ové komunikace*, B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2021.

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Tománek** Jméno: **Petr** Osobní číslo: **393221**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Distribuované učení neuronových sítí

Název diplomové práce anglicky:

Distributed training of neural networks

Pokyny pro vypracování:

Seznam doporučené literatury:

1. Recht, Benjamin, et al. "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent." Advances in neural information processing systems 24 (2011): 693-701.
2. Ho, Qirong, et al. "More effective distributed ml via a stale synchronous parallel parameter server." Advances in neural information processing systems 26 (2013): 1223-1231.
3. Chen, Jianmin, et al. "Revisiting distributed synchronous SGD." arXiv preprint arXiv:1604.00981 (2016).
4. Jin, Peter H., et al. "How to scale distributed deep learning?." arXiv preprint arXiv:1611.04581 (2016).
5. Lian, Xiangru, et al. "Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent." Advances in Neural Information Processing Systems 30 (2017): 5330-5340.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Tomáš Pevný, Ph.D., centrum umělé inteligence FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **10.02.2021** Termín odevzdání diplomové práce: **21.05.2021**

Platnost zadání diplomové práce: **30.09.2022**

doc. Ing. Tomáš Pevný, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. | Datum převzetí zadání | Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC