Distributed Training of Recurrent Neural Networks by FGM Protocol

Ilias Balampanis

Supervisor: Vasilis Samoladas
Antonios Deligiannakis
Michail G. Lagoudakis

A thesis submitted in fulfilment of
the requirements for the degree of
Diploma in Electrical and Computer Engineering

School of Electrical and Computer Engineering
Technical University of Crete

October 2020

Acknowledgments

First and foremost, I would like to thank my mentor Vasilis Samoladas for his trust and guidance. Besides, I am grateful for being part of this team. Sofia, Edward, and Eftychia supported and helped me to solve my problems during this work the last year. Furthermore, I would like to thank my friends from Chania, Stefanos, Yiorgos and Spyros, and especially Ioanna. Together we had some amazing moments during my student life. Last but not least, I would like to thank my family for their love, support, and constant encouragement.

Abstract

Artificial Neural Networks are appealing because they learn by example and are strongly supported by statistical and optimization theories. The usage of recurrent neural networks as identifiers and predictors in nonlinear dynamic systems has increased significantly. They can present a wide range of dynamics, due to feedback and are also flexible nonlinear maps. Based on this, there is a need for distributed training on these networks, because of the enormous datasets. One of the most known protocols for distributed training is the Geometric Monitoring protocol. Our conviction is that this is a very expensive protocol regarding the communication of nodes. Recently, the Functional Geometric Protocol has tested training on Convolutional Neural Networks and has had encouraging results. The goal of this work is to test and compare these two protocols on Recurrent Neural Networks.

Contents

List of Tables

Chapter 1

Introduction

Nowadays, deep neural networks are trained on ever-growing data corpora. As a result, distributed training schemes are becoming increasingly important. A major issue in distributed training is the limited communication bandwidth between contributing nodes or prohibitive communication costs in general. Many pieces of research have made a try on distributed deep learning, but very few have considered the enormous network traffic costs that such a style requires. Deep learning methods have proved to have strong predictive performance but on the other hand, a complex learning process. Opportunely, the training of artificial neural networks uses algorithms like Gradient Descent decreasing their loss, a fact that leads to convenience to distribute their learning procedure. In this work, we focus on distributing the learning process of Recurrent Neural Networks, while minimizing the communication of the remote sites.

1.1 Related Work and Motivation

The first tries for Distributed Machine Learning (DML) or Deep Learning were done by the parameter server method [13]. This structure of this method has nodes and a parameter server. The central idea is when some batches of data or some real training time have passed, nodes synchronize with the server sending their parameters. Then, the server aggregates all these model parameters and send back to nodes the fresh one to continue the learning process.

In 2019, Konidaris [11] used the GM [20] [21] and the FGM [4] [5] [19] to train one other architecture of Artificial Neural Networks, the Convolutional Neural Networks. The work had unbelievable results regarding the gap of the network cost between the two methods. So, there I found the motivation to use these two methods, this time to train an architecture that comprises Recurrent Neural Networks. Besides, while searching for my diploma thesis subject, I do not found a lot of works on distributed training of RNNs. Making work with successful results translates to a valuable source for other people in the Machine Learning community.

1.2 Thesis Goal

This work aims to provide a comparison of two methods for a distributed training process of Recurrent Neural Networks. The comparison is about the network cost of these two methods. I focus on two supervised learning problems, Classification and Natural Language Processing, using a subset of the RNN architecture, the Long-Short Term Memory Networks as learning models. The distributed learning process will be achieved by two geometric monitoring methods, the GM and the FGM. Furthermore, I am going to compare two safe function, to decide which is better for distributed Deep Learning purposes. I will refer to these functions in later chapters.

1.3 Thesis Overview

This section summarizes the structure of this diploma thesis.

Chapter 2 presents the background you need to understand the meanings of this work. At first, section **2.1** refers to Machine Learning Paradigms and Deep Learning. It makes a further reference to Recurrent Neural Networks because my Deep Learning model is based on there. Finally, section **2.2** introduces the two Geometric Monitoring methods which I have implemented and compared each other.

Chapter 3 presents the tools that helped me to implement the above algorithms. It also presents the structure and setup of the Deep Learning model. Lastly, explains in detail the implementation by giving some pseudocodes to make it easier to understand.

Chapter 4, explains the results that came off from the experimental phase.

Chapter 5, concludes this work as well as proposes some ideas for further research in the future.

In the end, **Appendix A** you can find the abbreviations I have used in my text, while **Appendix B** provides the tables with the numerical data that resulted from the experimental phase.

Chapter 2

Theoretical Background

This chapter is providing the necessary background for the two main pylons of this work, Deep (Machine) Learning and the protocols that used for the distributed training of the neural networks.

2.1 Machine Learning

Machine learning (ML) is a subset of Artificial Intelligence (AI) algorithms that gives systems the ability to learn in an automatic way and improve from experience without being explicitly programmed. Machine Learning concentrates on the development of computer programs that can reach data and learn from them. The learning process works with observations or data, such as examples, direct experience, or instruction, to explore patterns in data and get better decisions based on the samples we provide. The primary purpose is to enable the computers to learn without human intervention or assistance and modify actions accordingly.

2.1.1 Learning Paradigms

The three major learning paradigms are supervised learning, unsupervised learning and reinforcement learning. They each correspond to a particular learning task. Below, I will try to introduce them.

Supervised Learning

Supervised learning algorithms construct a model of data that have both the inputs and the correct outputs. The training data and comprises a set of examples. Each training example has at least one input and the correct output. Supervised learning algorithms construct an optimized function that predicts the output correlated with new inputs. The percentage of the prediction accuracy determines how successful the model is.

The types of ML problems that fits in this type of learning are the regression and the classification. Classification algorithms are used when the outputs are a set of values, and in regression are a range of arithmetical values. For example, a classification algorithm that separates tumors, would have an image of a medical instrument as input, and the type of the tumor as output, namely benign or malignant.

Unsupervised Learning

In contrast to supervised learning, unsupervised learning algorithms need a collection of data that carries only inputs and aims to discover any pattern in the data, such as groups or clusters of data points. Consequently, these algorithms use data that have not been labeled to learn from. Rather than evaluating, unsupervised learning algorithms recognize data patterns and respond based on the presence or absence of such similarities in each fresh bunch of data. Unsupervised learning algorithms are mainly applied in Statistics for density estimation. Yet unsupervised learning encircles more domains such as data feature summary and explain.

In cluster analysis a set of observations is divided into batches subgroups named clusters. Thus, according to one or more predefined criteria, observations within the same cluster are similar, while observations extracted from other clusters are divergent. Various clustering methods generate various assumptions on the structure of the data. The similarity between members of the corresponding cluster define the success of the method. Other methods are based on calculated density and graph connectivity.

Reinforcement learning

Reinforcement learning (RL) is a field of machine learning concerned with how software agents should perform actions in an environment to maximize some concept of aggregate reward. RL differs from supervised learning in not needing labeled input/output pairs to be defined, and in not needing sub-optimal actions to be explicitly corrected. Instead, the focus is on balancing between exploration (of an unknown area) and exploitation (of current knowledge). As a result of its abstraction, reinforcement learning is used in many AI fields. In ML, the environment is modeled as a Markov Decision Process (MDP). Dynamic programming techniques are used in many RL algorithms. RL algorithms do not presume the information of a specific algebraic model of the MDP and are used when exact models are infeasible. RL is a good stuff for Robotics, Bidding and Advertising and building bots for games.

2.1.2 Deep Learning

Deep learning (DL) is a specific subfield of machine learning. The 'deep' part in deep learning is not a reference to any kind of deeper understanding achieved by the approach. Rather, it stands for this idea of consecutive layers of representations. The number of layers that contribute to a model of the data is called the depth of the model. Modern deep learning often involves tens or even hundreds of successive layers of representations and they have all learned automatically from exposure to training data. Meanwhile, other approaches to machine learning tend to concentrate on learning only one or two layers of representations of the data.

In DL, the representations of these layered are trained on models called neural networks. The fundamental part of these networks is the neurons (see figure 2.1). These are stacked in layers. The phrase neural network is a reference to neurobiology, but even though basic thoughts in deep learning were developed influenced by the brain's neural system, DL models differ. For our purposes, deep learning is a mathematical framework for learning representations from data.

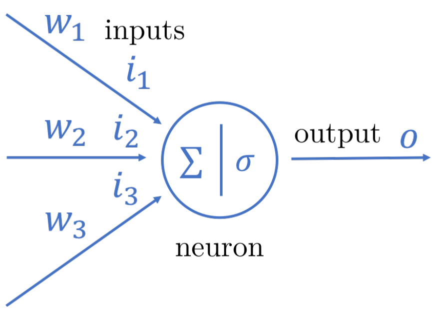


Figure 2.1. An artificial neuron

The simplest architecture of an Artificial Neural Network (ANN) is shown in Figure 2.2. This architecture is known as Feed forward Neural Network (FFNN) and is the most basic and widely used artificial neural network. Consider dealing with an image classification problem. The input of the network, consists of the distinct dataset sample. For this instance, this is an image pixels. It propagates the values over the hidden layers via the edges which are weighted until the last layer is reached. This is the output layer. The output represents the odds of the sample to belong to each one of the different classes, for example, the probabilities that the input image represents a benign tumor or a malignant one. At every neuron, a non-linear function can be triggered. Although this type of neural network has been successfully tested in many tasks, the temporal condition that it describes sequential data is not taken into account. Each sample has the fate to be distinct.

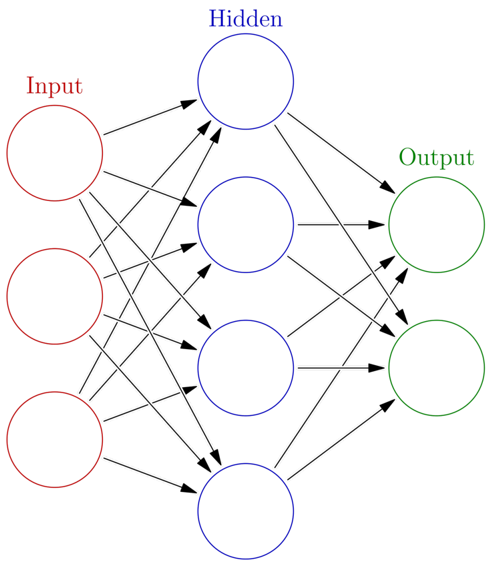


Figure 2.2. A Feedforward Neural Network with only one hidden layer

The specification of what a layer does to its input data is stored in the layer's weights, which in essence are a collection of numbers. In technical terms, we would assume that the transformation implemented by a layer is parameterized by its weights. Weights are also sometimes called the parameters of a layer. If we want to achieve a successful learning process, we must find a set of values for the weights of all layers in a network and this will correctly map example inputs to their correlated targets.

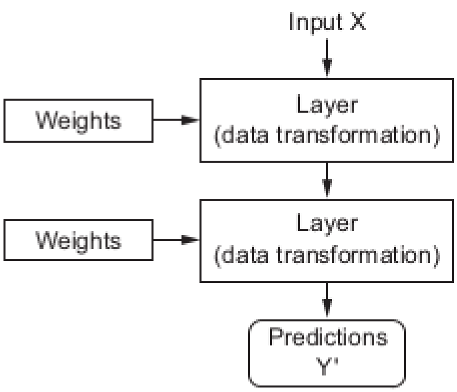


Figure 2.3. A neural network is parameterized by its weights

To succeed in learning, we need to be able to measure how far this output is from what you expected. This is the job of the loss function of the network. The loss function grabs the predictions of the network and the actual target which is the desired output and computes a distance score, capturing how well the network has done on this specific example.

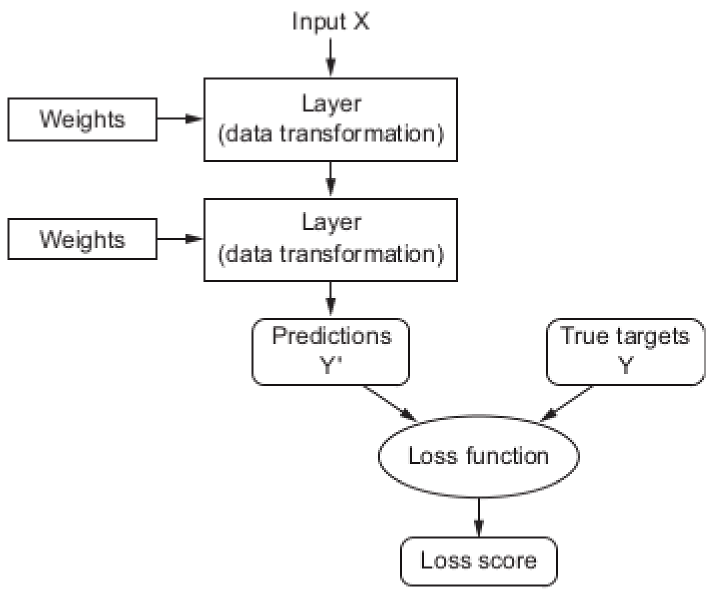


Figure 2.4. A loss function measures the quality of the network's output

The most elemental function is the Mean Squared Error (MSE). The formula is

$$MSE(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2, \tag{2.1}$$

where y stands for the target output and $\hat{y}$ is the output that we got from the network.

Training Process

At first, the weights of the network are initialized with random values, so the network merely performs a set of random transformations. Normally, its output is far away from what it should ideally be, and the accuracy is respectively very low. As the network processes the rest of the examples, it adjusts the weights a little in the correct direction, and the accuracy increases. This is the training loop, which, in a sufficient number of iterations, produces weight values that minimize the loss function. This process is called Backpropagation (BP). Actually, this is the backward propagation of the error.

Backpropagation is an algorithm that computes the chain rule, with a precise order of operations that is highly useful. Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Assume that y = g(x) and z = f(g(x)) = f(y). Then the chain rule asserts that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{2.2}$$

We can generalize this regarding a scalar case. Assume that $x \in \mathbb{R}^m, y \in \mathbb{R}^n, g$ maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $y = g(x)$ and $z = f(y)$, then

$$\frac{\partial z}{\partial x_i} =_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} \tag{2.3}$$

If I write with vectors, then we get

$$\nabla_x z = \frac{\partial y}{\partial x}^\top \nabla_y z \tag{2.4}$$

where $\frac{\partial y}{\partial x}$ is the $n \times m$ Jacobian matrix of g .

From this we understand that the gradient of a variable x can be reached by multiplying a Jacobian matrix $\frac{\partial y}{\partial x}$ by a gradient $\nabla_y z$. The backpropagation algorithm comprises performing such a Jacobian-gradient product for each operation in the graph. Let us see BP in more detail with two algorithms.

Forward pass

Below I have a forward pass through a standard Deep Neural Network (DNN) and the calculation of the cost function. The loss $L(\hat{y}, y)$ depends on the output $\hat{y}$ and on the target y. For integrity, this approach uses only a single input example x. Practical applications should work with mini-batches.

Algorithm 1 Forward pass in a standard DNN

Require: net depth l
Require: $W^{(i)}, i \in 1, \cdots, l$, the weights
Require: $b^{(i)}, i \in 1, \cdots, l$, the biases
Require: input x
Require: target output y
  $h^{(0)} = x$
  for $k \leftarrow 1$ to l **do**
    $a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$
    $h^{(k)} = f(a^{(k)})$
  end for
  $\hat{y} = h^{(l)}$
  $J = L(\hat{y}, y)$

Backward pass

Momentarily, I introduce the backward pass for the DNN of Algorithm 1. This computation produces the gradients on the activations $a^{(k)}$ for each layer k, starting from the output layer and moving backward to the first hidden layer. From these gradients, which can be described as evidence of how each layer's output should adjust to reduce error, one can get the gradient on the parameters of each layer. Generally, the main purpose is to minimize these gradients, following several iterations. This process is also called as Gradient Descent (GD).

Algorithm 2 Backward pass in a standard DNN

  $g \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} L(\hat{y}, y)$
  for $k \leftarrow l$ to 1 **do**
    $g \leftarrow \nabla_{a^{(k)}} J = g \odot f'(a^{(k)})$
    $\nabla_{b^{(k)}} J = g + \lambda \nabla_{b^{(k)}}$
    $\nabla_{W^{(k)}} J = g h^{(k-1)\top} + \lambda \nabla_{W^{(k)}}$
    $g \leftarrow \nabla_{h^{(k-1)}} J = W^{(k)\top} g$
  end for

A network with the smallest loss is one for which the outputs are as close as they can be to the targets. This is a trained network.

There are many different architectures of neural networks each with their unique strengths. For instance, Convolutional Neural Networks (CNN) show very effective results in image and video recognition. In this work, I will focus on Recurrent Neural Networks.

2.1.3 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is one powerful model from the deep learning family that has shown incredible results in the last years. It proposes to produce predictions on sequential data by utilizing a powerful memory-based architecture. But how is it differs from a feed-forward neural network? An FFNN works as a mapping function, where a single input is associated with a single output. In this type, no two inputs share knowledge and each moves in only one direction beginning from the input nodes, accessing hidden nodes and closing at the output nodes.

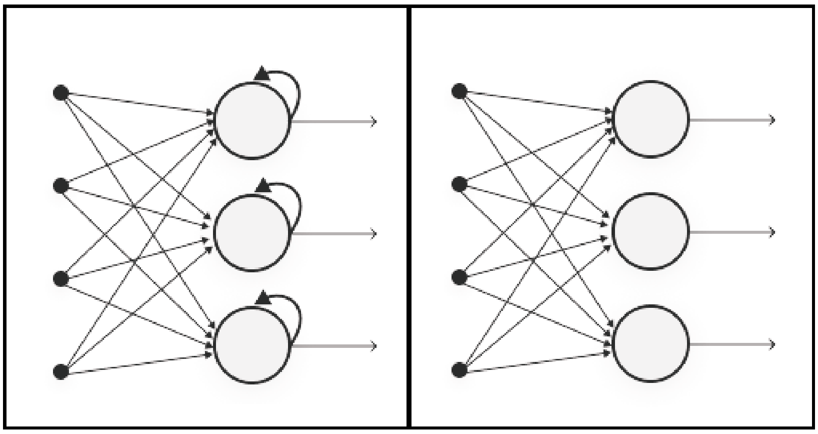


Figure 2.5. At left, this is a Recurrent Neural Network, and at right a Feedforward one.

Since a CNN is functional for dealing with a grid of values X such as a photo, a RNN is a network that is specially designed for dealing with a sequence of values $x^{(1)}, \ldots, x^{(\tau)}$.

To jump from a FFNN to a RNN, we need to consider the idea of splitting parameters through other parts of a model. Such splitting is especially significant when an exact chunk of information can happen at various positions within the sequence. For instance, look at the two sentences “I visited Italy in 2019” and “In 2019, I visited Italy.” If we request a ML model to read each words and obtain the year in which the reciter visited Italy, we would like it to see the year 2019 as the important piece of information, either it appears in the sixth word or the second word of the sentence. Assume that we have trained an FFNN that processes sentences of fixed length. A conventional fully connected FFNN would have separate parameters for each input feature, so it would need to learn all of the rules of the language separately at each position in the sentence. By comparison, an RNN shares the same parameters across a lot of time steps.

Basic Structure

Now, I will introduce the forward pass equations of an RNN represented in Figure 2.6. The figure does not define either the type of activation function for the hidden units or the loss function. Suppose we have the hyperbolic tangent (tanh) activation function and the MSE as the loss function. Additionally, we suppose that the output is distinct as if the RNN is used to forecast characters or words.

A straightforward manner to describe discrete variables is to rate the output **o** as providing the probabilities of each possible value. Now, it can be applied the softmax function to get a vector $\hat{y}$ across the output. Forward pass starts with a definition of the initial state $h^{(0)}$. Next, for each time step from $t = 1$ to $t = \tau$, we use the latter equations for updating:

$$a^{(t)} = b + W \cdot h^{(t-1)} + U \cdot x^{(t)} \tag{2.5}$$

$$h^{(t)} = tanh(a^{(t)}) \tag{2.6}$$

$$o^{(t)} = c + V \cdot h^{(t)} \tag{2.7}$$

$$\hat{y}^{(t)} = softmax(o^{(t)}) \tag{2.8}$$

where **b** and **c** are the biases along with the weight matrices **U**, **V**, and **W**, for input-to-hidden, hidden-to-output, and hidden-to-hidden connections, respectively. This is an example of a RNN that connects an input sequence to an output one. Both sequences have the same length. The entire loss for a given sequence matched with a sequence of output values would then be the summation of the losses over all the steps.

Calculating the loss function gradient concerning the weights has a lot of cost. The gradient calculation requires making a forward pass moving left to right, tailgated by a backward pass. The runtime is $O(\tau)$ cannot be cut by parallel execution because the forward pass graph is essentially sequential since each time step needs to be calculated after the previous one. The backpropagation algorithm utilized to the unfolded graph is called backpropagation through time (BPTT).

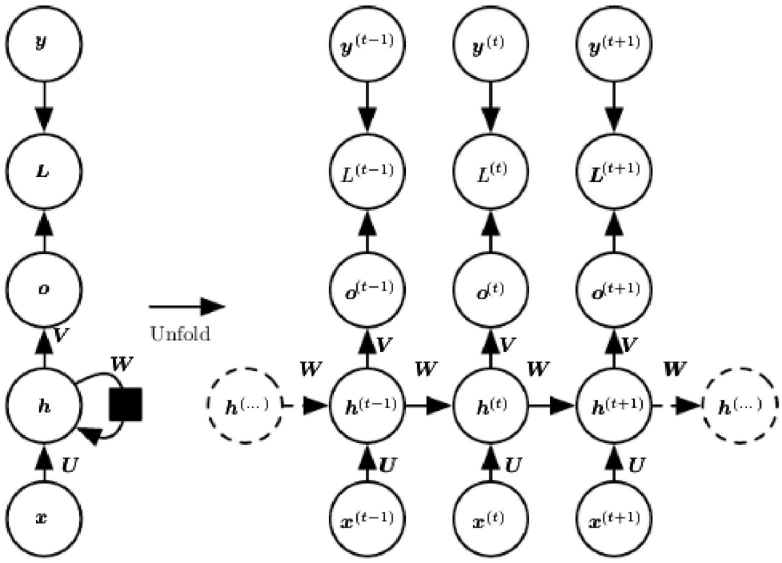


Figure 2.6. Unfolding an RNN

In Figure 2.6, at left we have the RNN and its loss have described with recurrent connections. At right, it is the same scene as a time-unrolled graph, where each node is now correlated with a specific time step.

Training and Evaluation

Calculating the gradient through a recurrent neural network is straightforward. One simply applies the generalized backpropagation Algorithm 2 to the unfolded graph.

To obtain some feeling for how the BPTT acts, we present an example of how to compute gradients by BPTT for the RNN equation 2.4 above. The nodes of our graph combine the weights U, V, W, b, and c as well as the sequence of nodes indexed by t for $x^{(t)}, h^{(t)}, o^{(t)}$, and $L^{(t)}$. For each node N, we ought to calculate the gradient $\nabla_N L$ recursively, based on the gradient calculated at nodes that follow it in the graph. We begin the recursion with the nodes directly preceding the terminal loss

$$\frac{\partial L}{\partial L^{(t)}} = 1 \tag{2.9}$$

In this derivation, we suppose that the outputs $o^{(t)}$ are managed as the argument to the softmax function to get the vector $\hat{y}$ of probabilities over the output. Besides, we suppose that the loss is the negative log-likelihood of the true target $y^{(t)}$. The gradient $\nabla_{o^{(t)}} L$ on the outputs at time step t, for all i, t, is as arises:

$$(\nabla_{o^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - 1_{i,y^{(t)}} \tag{2.10}$$

We work our way from the opposite direction, beginning from the tail of the sequence. At the last time step $\tau, h^{(\tau)}$ just has $o^{(\tau)}$ as a descendent. Therefore, we calculate its gradient as,

$$\nabla_{h^{(t)}} L = V^\top \nabla_{o^{(t)}} L \tag{2.11}$$

Now we are able to loop back in time to backpropagate gradients through time, from $t = \tau - 1$ down to $t = 1$, regarding that $h^{(t)}$ (for $t < \tau$) has as descendants both $o^{(t)}$ and $h^{(t+1)}$. Consequently, its gradient is provided by,

$$\nabla_{h^{(t)}} L = (\frac{\partial h^{(t+1)}}{\partial h^{(t)}})^\top (\nabla_{h^{(t+1)}} L)(\frac{\partial o^{(t)}}{\partial h^{(t)}})^\top (\nabla_{o^{(t)}} L) \Rightarrow \tag{2.12}$$

$$\nabla_{h^{(t)}} L = W^\top (\nabla_{h^{(t+1)}} L) diag(1 - (h^{(t+1)})^2) + V^\top (\nabla_{o^{(t)}} L) \tag{2.13}$$

where $diag(1 - (h^{(t+1)})^2)$ registers to the diagonal matrix containing the elements.

Once the gradients on the inner nodes of the graph are collected, we can get the gradients on the parameter nodes. Due to the weights are shared through many steps, we should get some concern when expressing calculus processes including these variables. The types we hope to complete working with the backpropagation method of the previous subsection, which calculates the addition of a single edge in the graph to the gradient. Nevertheless, the $\nabla_W f$ operator applied in calculus considers the contribution of W to the value of f because of the all edges in the graph. To fix this doubt, we import fool variables $W^{(t)}$ that are set to be clones of W but with each $W^{(t)}$ used hardly at time step t. Using these symbols, the gradient on the parameters that left is provided by:

$$\nabla_c L = \sum_t (\frac{\partial o^{(t)}}{\partial c})^\top \nabla_{o^{(t)}} L = \sum_t \nabla_{o^{(t)}} L \tag{2.14}$$

$$\nabla_b L = \sum_t (\frac{\partial h^{(t)}}{\partial b^{(t)}})^\top \nabla_{h^{(t)}} L = \sum_t diag(1 - (h^{(t)}))^2 \nabla_{h^{(t)}} L \tag{2.15}$$

$$\nabla_V L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}}\right) \nabla_V o_i^{(t)} = \sum_t (\nabla_{o^{(t)}} L) h^{(t)^\top} \tag{2.16}$$

$$\nabla_W L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{W^{(t)}} h_i^{(t)} \Rightarrow \tag{2.17}$$

$$\nabla_W L = \sum_t diag(1 - (h^{(t)}))^2 (\nabla_{h^{(t)}} L) h^{(t-1)^\top} \tag{2.18}$$

$$\nabla_U L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{U^{(t)}} h_i^{(t)} \Rightarrow \tag{2.19}$$

$$\nabla_U L = \sum_t diag(1 - (h^{(t)}))^2 (\nabla_{h^{(t)}} L) x^{(t-1)^\top} \tag{2.20}$$

Possible issues and Optimization

These architectures regularly face long-term dependencies. The primary problem is that gradients propagated over many stages and conduce to either vanish or explode resulting in much contamination to the optimization. Even if we suppose that the weights make a reliable network, the difficulty with long-term dependencies appears from the hugely smaller parameters given to long-term interactions associated with short-term ones.

This problem is particular to RNN. Figure multiplying a weight w by itself a lot of times. The product w^t will either vanish or explode depending on the magnitude of w. The vanishing and exploding gradient problem for RNNs were separately discovered by many researchers. Unluckily, to save memories in a way that is robust to small perturbations, the RNN must get in an area of weight range where gradients vanish. It does not mean that it is difficult to train, but that it may cost much time to learn long-term dependencies because the signal about these dependencies will tend to be hidden by the smallest changes resulting from short-term dependencies.

Long Short Term Memory Networks

Long Short Term Memory (LSTM) networks are a special set of RNN, capable of learning long-term dependencies. They introduced by Hochreiter and Schmidhuber in 1997. They operate remarkably well on a wide variety of problems. LSTM networks are explicitly created to bypass the long-term dependency obstacle. Memorizing information for long periods is reasonably their default behavior. All recurrent neural networks have the form of a chain of recurring modules of neural networks. In a traditional RNN, this repeating module will have a very simple composition, such as a single tanh layer.

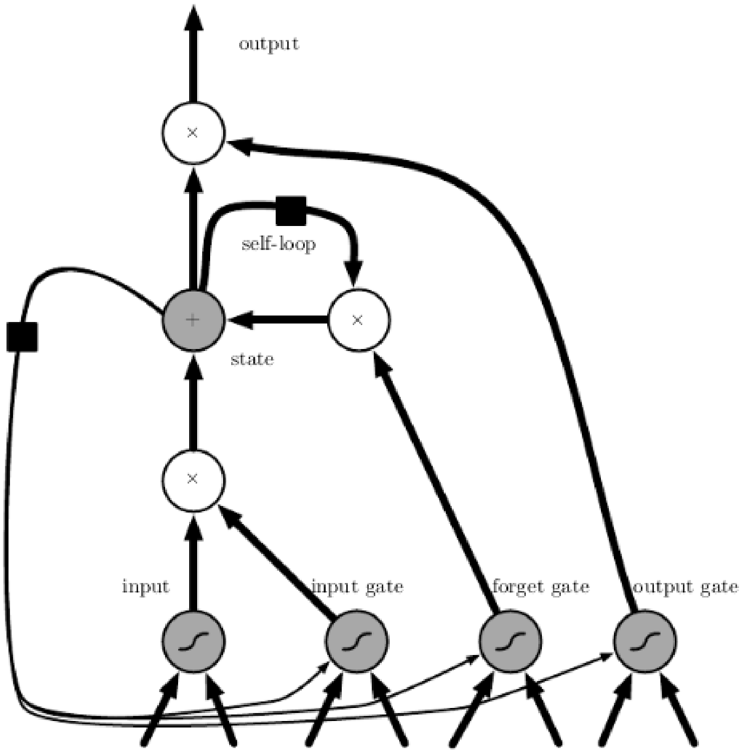


Figure 2.7. The structure of an LSTM cell

The smart idea of entering self-loops to build paths where the gradient can remain for long durations is the core of the initial LSTM model. By securing the parameters of this self-loop gated (controlled by another hidden unit), the time scale of integration can be adjusted dynamically. The LSTM has been observed notably successful in many applications, such as unconstrained handwriting recognition, speech recognition, and handwriting generation.

The LSTM block diagram is demonstrated in figure 2.7. The corresponding forward pass equations are given below. Instead of a unit that only applies an element-wise nonlinearity to the affine transformation of inputs and recurrent units, LSTM networks have cells that have an internal self-loop, except that the outer recurrence of the RNN. Each cell has the same inputs and outputs as a common RNN. However, it has more parameters and a system of gating units that control the flow of information. The most significant part is the state unit $s_i^{(t)}$ that has a linear self-loop. This self-loop weight is controlled by a forget gate unit $f_i^{(t)}$ (for time step t and cell i), that defines this parameter between 0 and 1 using a sigmoid function:

$$f_i^{(t)} = \sigma(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}), \tag{2.21}$$

where $x^{(t)}$ is the current input vector and $h^{(t)}$ is the current hidden layer vector, including the outputs of all the LSTM cells, and b^f, U^f, W^f are respectively biases, input weights and recurrent weights for the forget gates. Consequently, the LSTM cell internal state is updated as follows, but with a conditional self-loop weight $f_i^{(t)}$,

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}), \tag{2.22}$$

where b, U and W respectively declare the biases, input weights and recurrent weights into the LSTM cell. The **external input gate** unit $g_i^{(t)}$ is calculated similarly to the forget gate, but with its own parameters,

$$g_i^{(t)} = \sigma(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}), \tag{2.23}$$

The output $h_i^{(t)}$ of the LSTM cell can also be shut off by the output gate $q_i^{(t)}$, which also uses a sigmoid unit for gating,

$$h_i^{(t)} = tanh(s_i^{(t)}) q_i^{(t)} \tag{2.24}$$

$$q_i^{(t)} = \sigma(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}), \tag{2.25}$$

which has parameters b^o, U^o, W^o for its biases, input weights and recurrent.

LSTM networks have been proved to learn long-term dependencies more easily than conventional RNN. At first, they tested on artificial data sets designed for examining the ability to learn long-term dependencies. After that, this time had tested on challenging sequence processing tasks where state-of-the-art performance was achieved.

2.2 Geometric Monitoring Methods

Monitoring complex and continuous queries on distributed streams are by default a complicating problem. Sharfman, Schuster, and Keren in [20] [21], initially presented the Geometric Monitoring (GM) method for monitoring non-linear functions. This method is a communication protocol that resolves this problem efficiently by using convex analysis theory. Vasilis Samoladas and Minos Garofalakis in [4] [5] [19], generalized and improved this method with the Functional Geometric Monitoring (FGM) method by decreasing dramatically the network cost communication. In this section, we will introduce the theoretical background of these two geometric monitoring methods.

In this point, it is very important to define some stuff. The following algorithms fit in star network topologies. Thus, there are two fundamental entities, the local nodes/sites and the hub/coordinator. Consider we have k nodes/sites.

2.2.1 Geometric Monitoring

At each site, a local data stream is received and stands for a high dimensional vector $V \in \mathbb{R}^D$. Let define $S_i(t)$, $i \in [0, k]$ the local state vector. Assume w.l.o.g that in a random time step t, the coordinator holds the true global stream and this is the average vector $S(t)$ of local state vectors of all sites. So,

$$S(t) = \frac{1}{k}\sum_{i=1}^{k} S_i(t) \tag{2.26}$$

Consistently, a continuous query on the global state $Q(S(t))$ is a complicated non-linear function of S. To decrease communication costs among the hub and the local sites, the user can permit a small bounded error ϵ to the query answer. Specifically, the coordinator does not hold the actual global stream state $S(t)$, but a enough close estimation of it, $E(t)$, providing an approximate query response $Q(E(t))$, with a warranty that at any time step t will be

$$Q(S(t)) \in (1 \pm \epsilon) Q(E(t)) \tag{2.27}$$

In a same way, we define E_i as the last sent vector to the coordinator by the site i. Therefore,

$$E(t) = \frac{1}{k}\sum_{i=1}^{k} E_i(t) \tag{2.28}$$

Additionally, until the global estimate E is updated, and as long as the true global stream state $S(t)$ is inside the admissible region

$$A = \{x \in V | Q(x) \in (1 \pm \epsilon)Q(E)\}, \tag{2.29}$$

there is no need a site to communicate with the hub. When this condition has been violated, it is necessary to update the estimate E to compromise in the Eq. 2.27.

The GM protocol works in rounds. Every round starts when a new estimate E is aggregated in the coordinator and lasts until it will be updated again.

At the beginning of each round the hub the initial global state vector is equal with

$$S(t_0) = \frac{1}{k}\sum_{i=1}^{k} E_i = E. \tag{2.30}$$

The next step for the hub is to adopt and send to sites a 'good' safe zone $Z \subseteq A$ where Z is a convex subset of A and $E \in Z$.

Each node, at any time t maintains a drift vector $X_i(t)$. All nodes drift vectors compose the current global state. Thus,

$$S(t) = \frac{1}{k}\sum_{i=1}^{k} X_i(t) \tag{2.31}$$

Basic structure of protocol

At the beginning of each round, when $t = t_0, X_i(t_0) = E$ for all sites $i = 1, \ldots, k$. As a local stream update appears at a site i at time t, the invariant is managed by adding to Xi the vector $S_i(t) - S_i(t-1)$. Therefore, the actual difference of the local stream state E of each site is equal to $X_i(t) - E$. Obviously, at the kickoff of each round the drift vector is equal to the zero vector for all $i = 1, \ldots, k$.

Each local site observes the local condition $X_i(t) \in Z$. If this condition continues at each site, then by the convexity of Z and the drift invariant, it will be true that $S(t) \in Z$ too, and consequently, $S(t) \in A$. When a violation of the local condition $X_i \in Z$ happens, the site i wave the hub and the round ends. The coordinator says to the sites to broadcast the updates that occurred during the round. This can be arranged by shipping each local state vector Xi to the coordinator. Then, the coordinator refreshes E and starts a new round.

To be more specific, the tool that helps protocol monitor the local conditions is the variance of the current local stream. So,

$$Var[S(t)] = \frac{1}{k}\sum_{i=1}^{k} ||S(t) - E||_2^2 \tag{2.32}$$

The protocol receives a positive and real user-defined number as threshold T. While the variance of local streams described by the above equation is below the threshold T, the nodes continue to monitor this condition without communication with the hub. Differently, there is a need for communication with the hub. The local violation for a node i is described by the below condition.

$$||X_i(t) - E||_2^2 > T \tag{2.33}$$

Rebalancing the GM protocol

When you inspect the basic algorithm of the GM protocol you may mention that when a local violation occurs at a site i, it is not undoubtedly the fact that $S \in Z$. This might be true, in all other remote sites $j \neq i$, is still the case that $X_j \in Z$. To be more accurate, consider that after the beginning of a round, all the stream updates have been sent to the remote site i, whose drift vector X_i does not belong in the convex set Z but it's close enough. Next, it is true that all the other drift vectors X_j are yet equal to E, for $j \neq i$. Now, it is more obvious to realize that it is probably liberal to end the round at the first local violation.

These methods are principally heuristic and their goal is to reset some of the drift vectors so as to restore the local conditions at all sites with the dream of additional reduction of the communication cost. Firstly, we define the set $B = \{i\}$, where i is the site that yielded a local violation. Iteratively, we add each new local site index to B, and at each step, we compute the mean state vector X_B for all nodes $\in B$. If $X_B \in Z$, we reset the drift vector that for all the nodes $\in B$ to X_B and the round resumes regularly. Else, if $|B| = k$, the round comes to an end.

The goal of the rebalancing methods is to prolong the round life. There is not any mathematical proof that it is causal, but many experimental researches have confirmed that such heuristic methods can achieve better performance.

2.2.2 Functional Geometric Monitoring

Let's explore the idea of a safe state in a monitoring algorithm. The system is in a safe state while $\frac{1}{k}\sum_{i=1}^{k} X_i = S \in A$. In the meaning of the GM protocol, system safety is only monitored by observing all local conditions $\wedge_{i=1}^{k} X_i \in Z$, where Z is a convex subset of the admissible region A. When this becomes false, the system restores it, either by starting a new round or by rebalancing.

On the contrary, FGM uses a real function $\phi : \mathbb{R}^D \to \mathbb{R}$. Each remote site i, for $i = 1, \ldots, k$, holds its ϕ-value, or the value of function ϕ on their state vector X_i as it becomes updated. Thus, system safety is ensured while the global summation of those one-dimensional projections sum $\psi = \sum_{i=1}^{k} \phi(X_i)$ is non-negative.

Mathematical definitions and theorems

Underneath, we introduce some significant definitions and theorems that are needed to comprehend the FGM protocol.

Definition 1 (Safe function). *A function $\phi : \mathbb{R}^D \to \mathbb{R}$ is safe for admissible region A, if, for all $X_i \in \mathbb{R}^D, i = 1, \ldots, k$,*

$$\sum_{i=1}^{k} \phi(X_i) \geq 0 \Rightarrow \frac{\sum_{i=1}^{k} X_i}{k} \in A$$

Theorem 1. *For any set A, if ϕ is safe for A, then exists a* ***concave*** *function $\zeta \geq \phi$ which is also safe for A.*

Based on the above theorem, the FGM protocol limits its application exclusively to concave safe functions. For any function ϕ, define the level set of ϕ, as

$$L(\phi) = \{x \in \mathbb{R}^D | \phi(x) \geq 0\}$$

For ϕ to be safe for some A, it is essential that $L(\phi) \subseteq A$. This is also satisfactory for a concave function ζ.

Proposition 1. *A concave function ζ is safe for A, if and only if, $L(\zeta) \subseteq A$.*

Proof. To prove sufficiency, consider $L(\phi) \subseteq A$. By the definition of a concave function, for any $k \geq 1$,

$$\zeta\left(\frac{\sum_{i=1}^{k} X_i}{k}\right) \geq \frac{1}{k}\sum_{i=1}^{k}\zeta(X_i)$$

Next, $\frac{1}{k}\sum_{i=1}^{k}\zeta(X_i) \geq 0$ implies $\zeta(S) \geq 0$, and thus $S \in L(\zeta) \subseteq A$ □

Moreover, if ζ is concave, then the set $Z = L(\zeta)$ is either convex and closed. Hence, a concave safe function ζ for an admissible region A can be described as a functional representation for a convex safe zone $L(\zeta) \subseteq A$. In this way, FGM is conceptually a generalization of GM. Below we define a safe zone function.

Definition 2 (Safe zone function). *Given an admissible region A and a reference point E, a safe zone function ζ is a concave function which is safe for A, and $\zeta(A) > 0$.*

A 'good' safe zone massively depends on the quality of the safe zone function. Here in [5] are described the principles for the quality of a safe zone function where safe zone functions are used in the compositional design of high-end safe zones for complicated queries. The problem of defining safe zone functions for specific queries can be beneficial, particularly in ML problems.

Basic structure of protocol

The FGM protocol also works in rounds. Monitoring the threshold condition

$$\sum_{i=1}^{k}\phi(X_i) \leq 0 \tag{2.34}$$

over the duration of the round. At the beginning of a round, the coordinator has a perfect knowledge of the current state of the system $E = S$. It selects an (A, E, k)-safe function ϕ, where A is the admissible region, E is the current estimate and k is the number of local nodes. At any point in time, assume $\psi = \sum_{i=1}^{k}\phi(X_i)$.

The **round's** steps are:

(1) At the beginning of a round, the coordinator ships ϕ to every site (it is sufficient to ship vector E). Local sites initialize their drift vectors to 0. With these settings, initially it is $\psi = k\phi(0)$.
(2) Next, the hub defines a number of subrounds, which will be described in detail below. At the end of all subrounds, we 'll have $\psi > \epsilon_\psi k\phi(0)$, for some small ϵ_ψ, which usually is set to 0.01.
(3) At the end, the hub ends the round by collecting all drift vectors and updating E.

The goal of each subround is to check the condition $\psi \leq 0$ coarsely, with a precision of θ, achieving this with as little communication as possible. The **subround's** steps are:

(1) At the beginning of a subround, the coordinator knows the value of ψ. It calculates the subround's quantum $\theta = -\psi/(2k)$, and sends θ to each local site. In addition, the hub initializes a counter $c = 0$. Each local site holds its initial value $z_i = \phi(X_i)$, where $2k\theta = \sum_{i=1}^{k} z_i$. Moreover, each local site initializes a counter $c_i = 0$.
(2) Each local site i keeps its local drift vector X_i , as it makes stream updates. When X_i is updated, site i updates its counter,

$$c_i = \max\{c_i, \lfloor \frac{\phi(X_i) - z_i}{\theta} \rfloor\} \tag{2.35}$$

If this update increases the counter, the local site sends a message to the coordinator, with the increase to c_i.
(3) When the coordinator receives a message with a counter increment from a site, it adds the increment to its global counter c. If the global counter c is bigger than k, the hub ends the subround by collecting all $\phi(X_i)$ from all local sites, recomputing ψ. If $\psi \geq \epsilon_\psi k\phi(0)$, the subrounds end, else another subround begins.

Chapter 3

Implementation

In this chapter, I am going to analyze my implementation process. Initially, I will refer to the libraries and the tools that helped me to implement this work. Next, I will define the DL model. Finally, I will explain how this model attached to the stream monitoring protocols.

As I said in section 1.1, both GM and FGM do not follow the classic parameter server model. At first glance, the coordinator seems to have this functionality. Although, the way that decides to make the synchronizations differs from the classic model. The parameter server synchronizes the workers after some discrete steps while the stream monitoring methods synchronize when a condition is violated. It leads to a more efficient training process regarding the communication between workers and the coordinator.

At this point, it is essential to note that this work is a simulation of a real system. The goal of this thesis is to prove that FGM is more efficient than GM in respect of the network cost without losing the accuracy of the ML model. I achieved this by tracking the communication cost (bytes) between the network workers.

3.1 Libraries and Tools

This project is developed mostly in *C++* and rarely in *Python*. The library that I handled to track the communication over the workers is called *ddssim* [18] and was developed in C++ by my mentor, Vasilis Samoladas. I used the Python and especially the *Pandas* [22] and *scikit-learn* [15] libraries to preprocess the NLP data, and I utilized the *matplotlib* [7] library to visualize both data and results. I used the *mlpack* library [1] for ML purposes, which is written in C++.

3.2 Distributed learning using GM protocol

I introduced the basic structure of GM protocol in 2.2.1. I used the Kamp's rebalanced method Kamp to make distributed training. Both GM and FGM protocols monitor data streams. For this reason, I should adjust some meanings to the distributed DL concept. For the distributed training, $S_i(t)$ is the local model parameters (weights) of a site (worker) while E is the average of the local model parameters or the global model. Besides, the drift vector $X_i(t)$ is the difference between the previous and current model weights. The threshold T has the same meaning as in the stream monitoring.

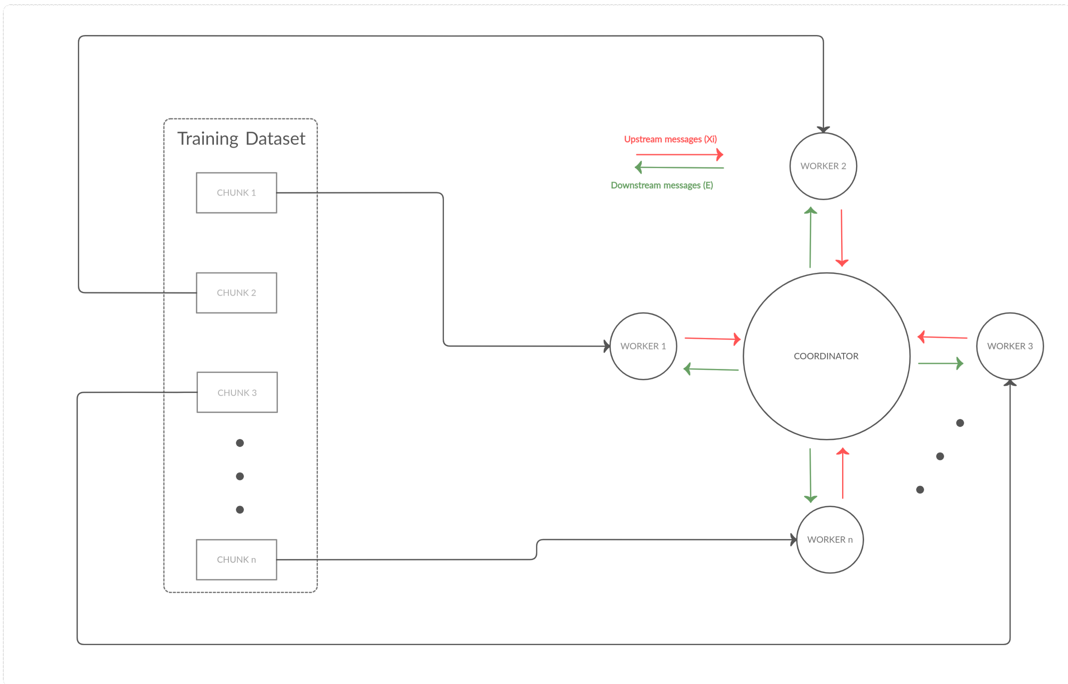


Figure 3.1. Coordinator - Worker model for training via GM

The learning process by GM is organized in three (3) phases.
In the first phase, the coordinator warms up it's ML model with a small bunch of data. In this way, the system becomes stable earlier since the other option was to define the global model as a zero vector. A system is called stable when the pace of new rounds is solid. After that, the global violation counter u is defined to zero and the global estimate E has as value the warmed up vector. The global estimate E will be the final trained model. The first round has already started.

In the second phase, each worker fits a batch of samples to its model. In other words, at this moment each worker calculates the drift vector which is the difference from the past and the current model parameters vector. Next, checks if the fresh model violates the local condition (Eq. 2.33). If it is true, this worker sends a violation message to the coordinator. Else, it keeps going by fitting the next batch of samples.

The third and the last phase concerns the coordinator. In this phase, the coordinator should handle the situation that resulted from the previous phase. Initially, the hub checks if the workers that have violated the condition are equal to the total number of the system workers. If it's true, then the hub sets a new global estimate averaging all the workers' model parameters. Then it sends the new model to all workers. Otherwise, it aggregates and sends only the model parameters from the workers who have violated the local condition.

Following, I present the algorithm that describes the process of distributed training using the GM protocol which is based on 2.2.1

Algorithm 3 Learning process via GM

Require: T (threshold), k (# workers)

Phase 1 – Initialization

warm up the global learner (coordinator) and **take** the initial weights w_0
violation counter $v \leftarrow 0$
global estimate $E \leftarrow w_{init}$

Phase 2 – Round n ***at worker*** i

fit a batch of samples to the local model and calculate the new drift vector X_i
if $||X_i(n) - E||_2^2 > T$ **then**
  send $w_{n,i}$ to coordinator {**violation**}
end if

Phase 3 – The coordinator deals with a local violation

let B the set of workers that have violated the condition
$v \leftarrow v + |B|$
if $u = k$ **then**
  $B \leftarrow [k]$
end if
while $B \neq [k]$ **and** $\frac{1}{k} \sum_{j \in B} ||X_j(n) - E||_2^2 > T$ **do**
  receive model parameters from workers $\in B$
end while
send model parameters $S(n) = \frac{1}{|B|} \sum_{j \in B} X_j(n)$ to workers $\in B$
if $B = [k]$ **then**
  set a new global estimate $E \leftarrow S(n)$
end if

3.3 Distributed learning using FGM protocol

To describe the learning process using the FGM protocol, I based on the basic structure that I have quoted in 2.2.2.

Safe function

The GM protocol handles the safe zone condition of Eq 2.33 to synchronize the learners. This time, I adopted and adjusted [19] this safe zone to a real safe zone function $\phi : V \to R$, to match with FGM protocol. The admissible region A is the convex level set

$$A = \{x |\, \|x\|_2 \geq (1 - T)\, \|E\|\} \tag{3.1}$$

Next, the safe function can be combined via the point-wise max operation:

$$\phi(X, E) = max\{-T \|E\| - X\frac{E}{\|E\|}, \|X + E\| - (1 + T) \|E\|\} \tag{3.2}$$

where X is the drift vector of a worker.

Learning by FGM

In the beginning of 3.2 I mentioned to the variables that both algorithms have. Not only GM but also FGM has the same meanings to these variables (e.g E, X_i, T). The FGM has an additional variable, the ϵ_ψ which is a small number that is not related to the desired accuracy query ϵ (or T), but only to the desired quantization for monitoring ψ.

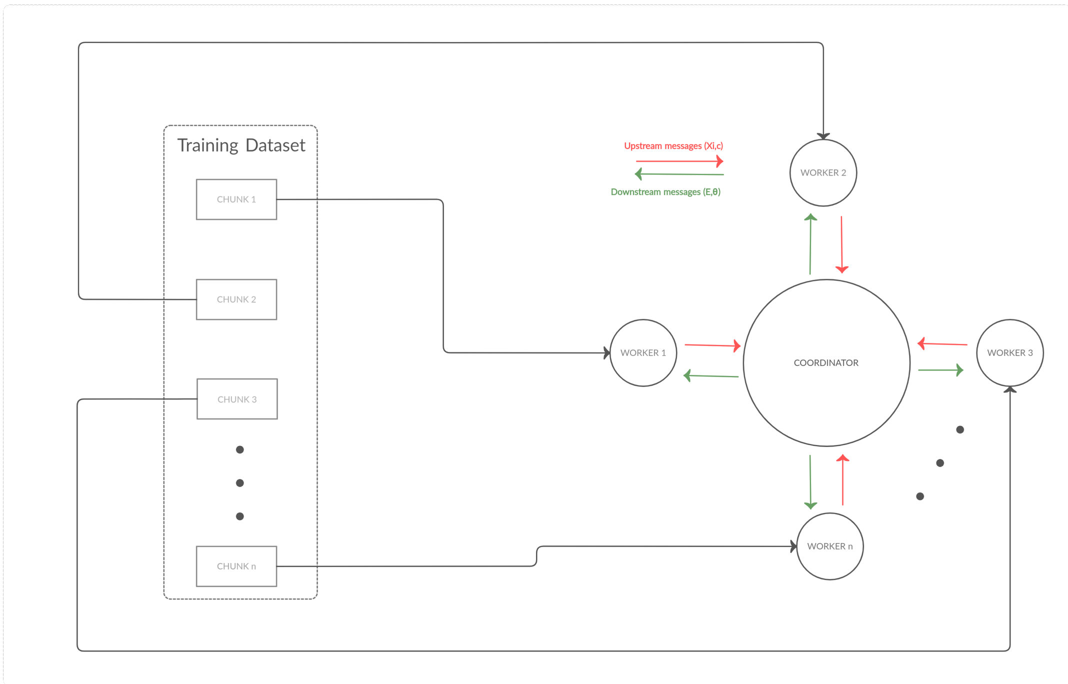


Figure 3.2. Coordinator - Worker model for training via FGM

The learning process by FGM is organized in four (4) phases.
In the first phase, as in GM, the coordinator warms up it's ML model with a small bunch of data. After that, the global counter c is defined to zero and the global estimate E has as value the warmed up vector. Additionally, ψ and quantum θ takes the initial value as it defined in subsection 2.2.2. The first round has already started.

In the second phase, the worker has received the global estimate E and the quantum θ and updates its corresponding variable. The difference between the second and third phases is the update of E, which is not needed for the start of a new sub-round in phase 3.

In the fourth phase, each worker fits a batch of samples to its model as the third one in GM. Next, checks if violates the local condition (Eq. 2.35). If it is true, this worker sends the local increment to the coordinator. Else, it keeps going by fitting the next batch of samples.

In the fifth phase, the coordinator should check if the increment that has received leads to a violation. Initially, the hub sums the received increment to the global counter. If the global counter is greater than the total number of the system workers, then the hub calculates the ψ. If ψ is greater or equal to $\epsilon_\psi k\phi(\overrightarrow{0})$ then the coordinator sets the variables and routes a new round. Otherwise, a new sub-round starts.

Below, I present the algorithm that describes the process of distributed training using the FGM protocol.

Algorithm 4 Learning process via FGM

Require: ϕ (safe function), ϵ_ψ, k (# workers)

Phase 1 – Initialization

warm up the global learner (coordinator) and **take** the initial weights w_0
global estimate $E \leftarrow w_0$
global counter $c \leftarrow 0$
$\psi \leftarrow k\phi(\overrightarrow{0}, E)$
quantum $\theta \leftarrow -\frac{\psi}{2k}$

Phase 2 – Starting a new round: Worker i receives E and θ

update the drift vector $X_i \leftarrow E$
quantum $\leftarrow \theta$
local counter $c_i \leftarrow 0$
$z_i \leftarrow \phi(X_i, E)$

Phase 3 – Starting a new subround: Worker i receives θ

$c_i \leftarrow 0$
quantum $\leftarrow \theta$
$z_i \leftarrow \phi(X_i, E)$

Phase 4 – Worker i fits a batch of samples

fit a batch of samples to the local model and calculate the new drift vector X_i
quantum $\leftarrow \theta$
$z_i \leftarrow \phi(X_i, E)$
$currentC_i = \lfloor \frac{\phi(X_i,E)-z_i}{\theta} \rfloor$
$maxC_i = \max(currentC_i, c_i)$
if $maxC_i \neq c_i$ **then**
    $incr_i = maxC_i - c_i$
    $c_i = currentC_i$
    send increment $incr_i$ to the coordinator
end if

Phase 5 – Coordinator receives an increment $incr_i$

$c \leftarrow c + incr_i$
if $c > k$ **then**
    collect all $\phi(X_i, E)$ from all workers
    $\psi \leftarrow \sum_1^k \phi(X_i, E)$
    if $\psi \geq \epsilon_\psi k\phi(\overrightarrow{0})$ **then**
        send E and θ to all workers and **goto** *Phase 2* {**round condition violation**}
    else
        $c \leftarrow 0$
        $\theta \leftarrow -\frac{\psi}{2k}$
        send θ to all workers and **goto** *Phase 3* {**subround condition violation**}
    end if
end if

Chapter 4

Experimental Results

At this point, I would like to note that all these executes had run in the computing grid of my university, the Technical University of Crete. This grid consists of forty-four (44) nodes, each with four (4) CPUs.

At the beginning of each run, I was shuffling the input data. I made this to create some randomness in my experiments. For this reason, to have safe results and a sufficient statistical sample, I run thirty (30) times each experimental setup. The results that came off from each case are the average of these runs.

The following plots show the final results. There were a lot of failed attempts to reach them. Each attempt cost four (4) days of continuous usage of the computing grid.

The numerical data that all the below plots are based are in Appendix B.

4.1 Models and Datasets

4.1.1 Classification problem

Looking into this GitHub repository [23] without the usage of CNNs, I concluded that the best setup is with one LSTM layer with 19 units. Finally, we add a Dense layer with 39 units (as the number of classes) to predict the category of the crime.

I selected the *MSE* function for calculating the error and the *Adam SGD* optimizer. I also selected the *ReLU* activation function for the LSTM layers and the *Softmax* for the Dense.

Regarding the hyperparameters, again based on this repository [23], I have chose,

- **Points look back:** 5
- **Learning rate:** 0.005
- **Maximum optimizer iterations:** 500
- **Sample shuffling:** Yes
- **Adam's tolerance:** 1e−08
- **Adam's epsilon:** 1e−08
- **Adam's beta1:** 0.9
- **Adam's beta2:** 0.99

Figure 4.1. The structure of the DL Model for Classification purposes

For this work, I selected the **San Francisco Crime Classification (SFCC)** dataset [14].

This dataset contains incidents derived from SFPD Crime Incident Reporting system. The data ranges from 1/1/2003 to 5/13/2015.

There are 9 variables:

- Dates - timestamp of the crime incident
- Category - category of the crime incident.
- Descript - detailed description of the crime incident.
- DayOfWeek - the day of the week
- PdDistrict - name of the Police Department District
- Resolution - how the crime incident was resolved.
- Address - the approximate street address of the crime incident.
- X - Longitude.
- Y - Latitude

The dataset had totally 878,049 samples. I split the data into training and testing samples with a ratio equivalent to 0.85. In other words, the training samples were approximately 746,340 while the testing was approximately 131,700 samples. The load that got each worker was equal.

The goal is to predict the category of crime that occurred, given the time and location and the rest of variables. The number of crime categories is 39 which is the number of classes.

4.1.2 Natural Language Processing problem

The model was built to make sentiment analysis for as much as we have an NLP problem. The input is sequences of words and the output is the case that this sequence is either positive or negative. The structure of the model is represented below.

Figure 4.2. The structure of the DL Model for NLP purposes

Here, we see that the first is the input layer. Inside the layer, there is a built-in word embedding layer, which I implemented through the Python and the sklearn library. The method of learning word embeddings from the text I chose was the *word2vec* algorithm.

Based on this article [17] and after a bunch of runs of trying not to reduce the accuracy, I concluded that the best setup is a stacked LSTM with three (3) layers. Both the first and the second layer has 256 units. The third one has 128. Finally, we add a Dense layer with 1 unit to help decide if the sequence has a positive or negative sentiment.

I selected the *MSE* function for calculating the error and the *Adam SGD* optimizer. I also selected the *ReLU* activation function for the LSTM layers and the *Sigmoid* for the Dense.

Regarding the hyperparameters, again based on this article [17], I have chose,

- **Words look back:** 30
- **Learning rate:** 0.005
- **Maximum optimizer iterations:** 1000
- **Dropout rate:** 0.2
- **Sample shuffling:** Yes
- **Adam's tolerance:** 1e−08
- **Adam's epsilon:** 1e−08
- **Adam's beta1:** 0.9
- **Adam's beta2:** 0.99

For this work, I selected the **Amazon Fine Food Reviews (AFFR)** dataset [16].

This dataset consists of reviews of fine foods from Amazon. The data span a period of more than ten (10) years, including all $500,000$ reviews up to October 2012. Reviews include product and user information, ratings, and a plain text review. It also includes reviews from all other Amazon categories.

In detail, the dataset includes:

- Reviews from October 1999 to October 2012
- 568,454 reviews
- 256,059 users
- 74,258 products
- 260 users with $>$ 50 reviews

From all the above details, I cared only for the reviews. Hence, my dataset had 568,545 samples. I split the data into training and testing samples with a ratio equivalent to 0.85. In other words, the training samples were approximately 483,000 while the testing was approximately 85,200 samples. The load that got each worker was equal.

4.2 Results

In this section, I will present the results of this work. I separated the results in two parts.

In the first part, I compare the two protocols using the same safe function. This function is called **spherical cap** (Eq. 3.2) and is defined in section 3.3. I chose this function because in the next subsection I will prove that is better than the other (simple norm).

In the second part, I compare the two safe functions using -this time- the same protocol. I chose the FGM protocol.

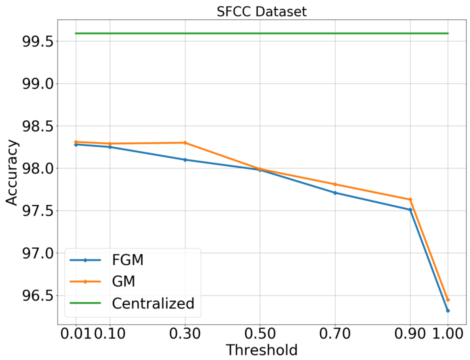
I have divided the experiments into three (3) subgroups. In each case, the accuracy, the number of rounds, and the network traffic are compared each time with a different variable. The first group is regarding the **threshold** (T), the second is regarding the **batch size** (b), while the third one is regarding the **number of workers** (k).

4.2.1 Protocols comparison

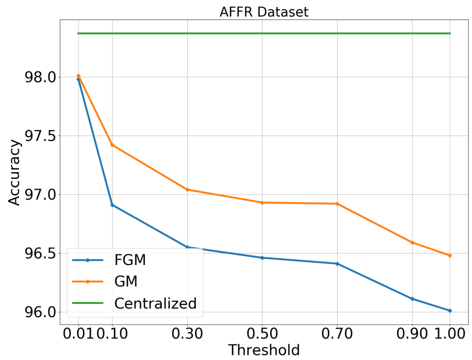
Firstly, I would like to note the accuracy that the model had when I trained it in a centralized way. A centralized way is when only one worker undertakes the learning process. To attain a logical comparison of the accuracy, I consider that the maximum value that the model can achieve is via centralized training.

The model for classification purposes had 99.59% while the NLP one had 98.37% accuracy.

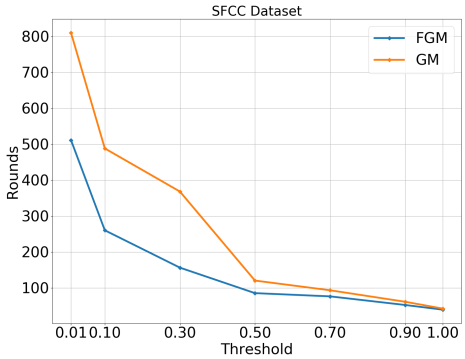
In Figure 4.3 A,B, we see the accuracy, the number of rounds and the traffic when the threshold changes. It is reasonable that all of them decrease over the threshold increase because of the lack of communication. It happens because the violation condition has relaxed. This reduction of accuracy is not so notable as to make our model tolerant of less communication. Besides, we observe that the absolute number of rounds is greater in the AFFR dataset than the other (Fig. 4.3 C,D). It arises not only from the nature of the problem but also the structure of the model. The model for the AFFR dataset has two additional LSTM layers. For this reason, the traffic is much more in NLP model training (Fig. 4.3 E,F). It is also notable that both curves about the number of rounds and traffic follow the same direction. In this case, the **batch size** equals to 16 and the number of **workers** is 8.



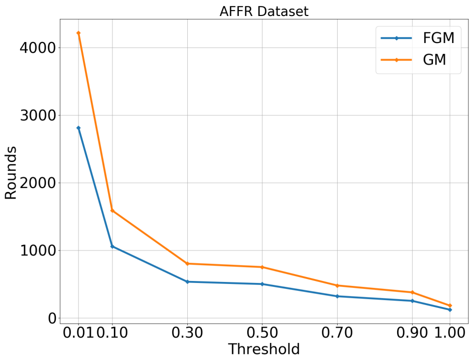
(A)



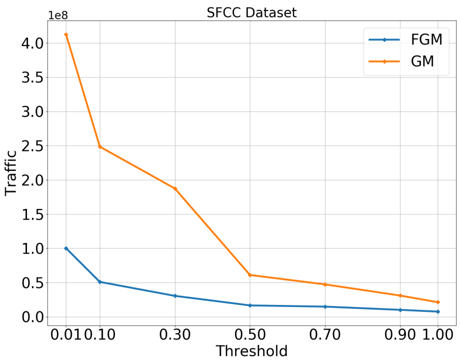
(B)



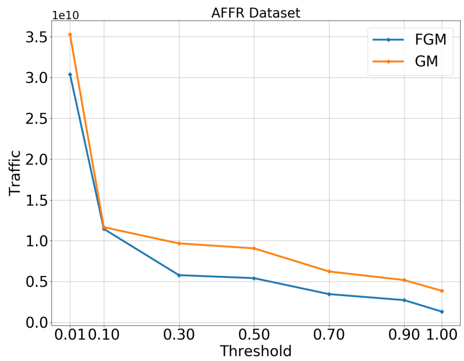
(C)



(D)



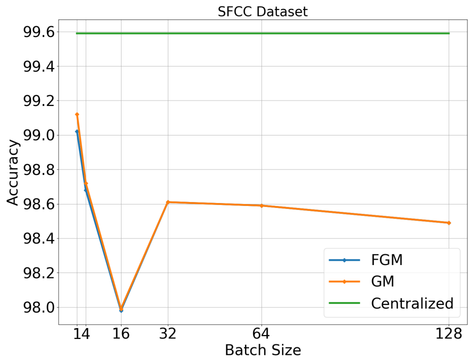
(E)



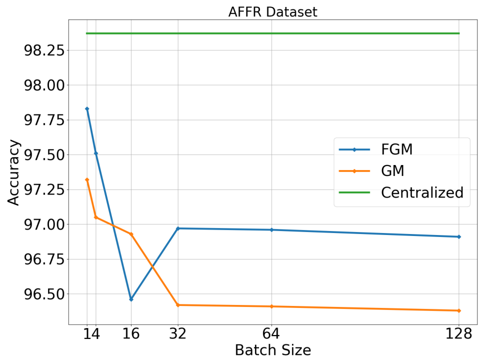
(F)

FIGURE 4.3

Before a worker calculates its new drift vector, it fits to the local model a batch of samples. In this subgroup, the variable that changes is the **batch size**. So, the **threshold** equals to 0.5 and the number of **workers** is 8. It is very interesting that the accuracy in SFCC is precisely the same (Fig. 4.4 A). In AFFR (Fig. 4.4 B), the FGM is better than GM, but the difference is just 0.5% which is a negligible amount.



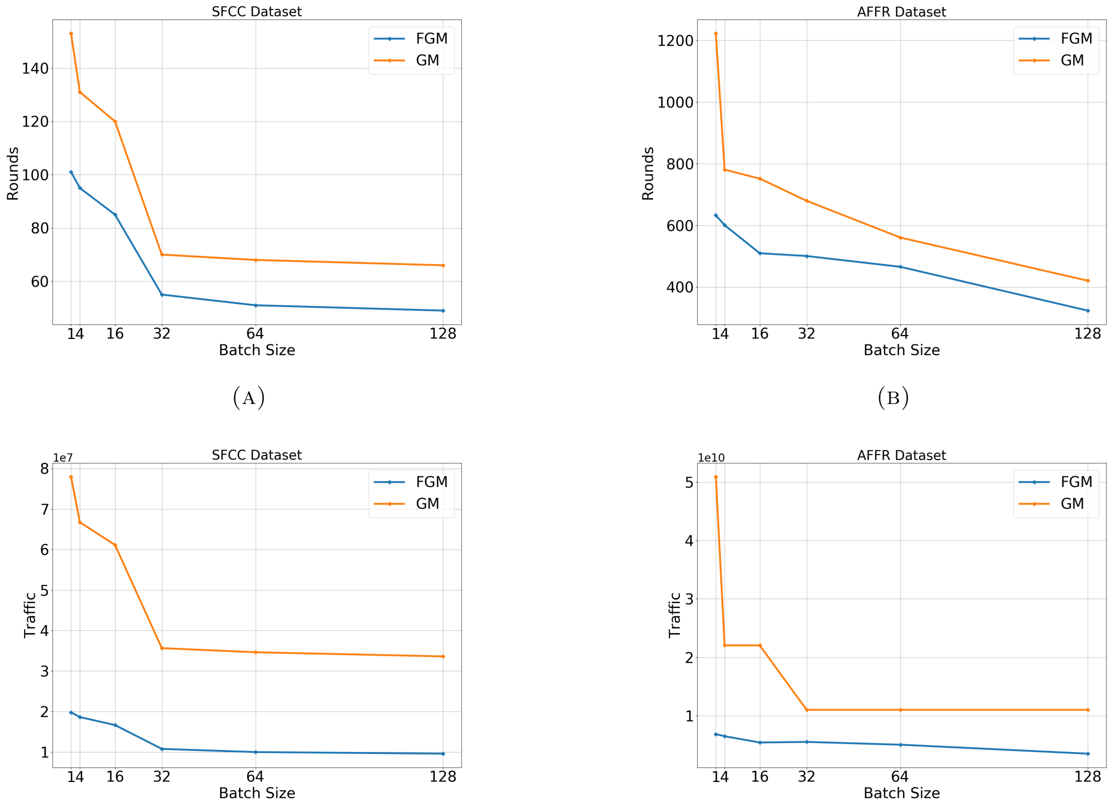
(A)



(B)

Figure 4.4

As in the graph above with the rounds when the threshold was changing, it is acceptable that the rounds decrease while the batch size increases (Fig. 4.5 A,B). Regarding the network traffic while batch size increasing (Fig. 4.5 C,D), note that the traffic is decreasing with a smaller rate due to subrounds of FGM. FGM is 3.5 times more efficient than GM in SFCC. In AFFR, the percentage change is 100%.

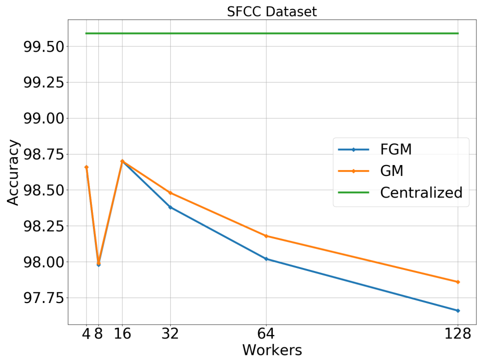


(A) (B)

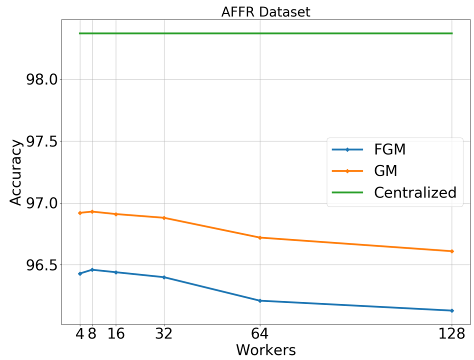
(C) (D)

Figure 4.5

In this subgroup, the variable that changes is the number of **workers**. Hence, the **threshold** equals to 0.5 and the **batch size** is 16. In SFCC and especially in the case of 8 workers, we see probably an outlier (Fig. 4.6 A,B). After that, the accuracy decreases over the increase of scalability, but not too much to make a problem. In AFFR, there is no outlier and the accuracy follows the expected direction. Here FGM is more precise than GM by 0.5%.



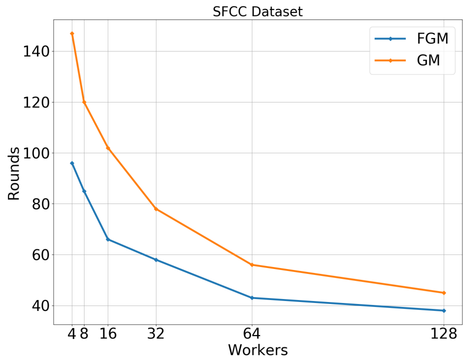
(A)



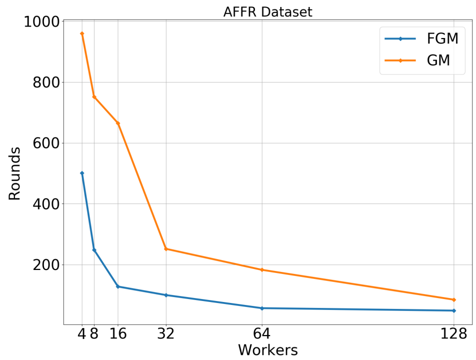
(B)

Figure 4.6

In Figure 4.7 A,B, it seems that both models follow the same direction. The rounds decrease over the increase of workers. Again the class model achieves fewer rounds than NLP.



(A)



(B)

Figure 4.7

These plots (Fig. 4.8 A,B) are typical GM-FGM plots on scalability analysis. FGM keeps the same levels of traffic while the workers are increasing. On the other hand, GM uses more network resources while the network getting bigger. FGM is up to 7 times more efficient than GM in SFCC and up to 15 in AFFR. NLP problems fit better to this architecture. Additionally, these plots are the most important of this work since the gap of the network cost between the two methods becomes enormous, while the accuracy remains at the same high levels in both algorithms. The number of rounds may not be very different by absolute numbers, although this is where the superiority of FGM is located.

At this point, note that for FGM a subround costs some bytes since the only thing that needs to be transferred is the increment from the workers to the coordinator and the quantum from the coordinator to the workers. On the other hand, a rebalance is equivalent to thousands of bytes since the workers send their entire model to the coordinator. Therefore, it seems that as the number of workers increases, the communication in GM increases linearly while in FGM it remains relatively stable.

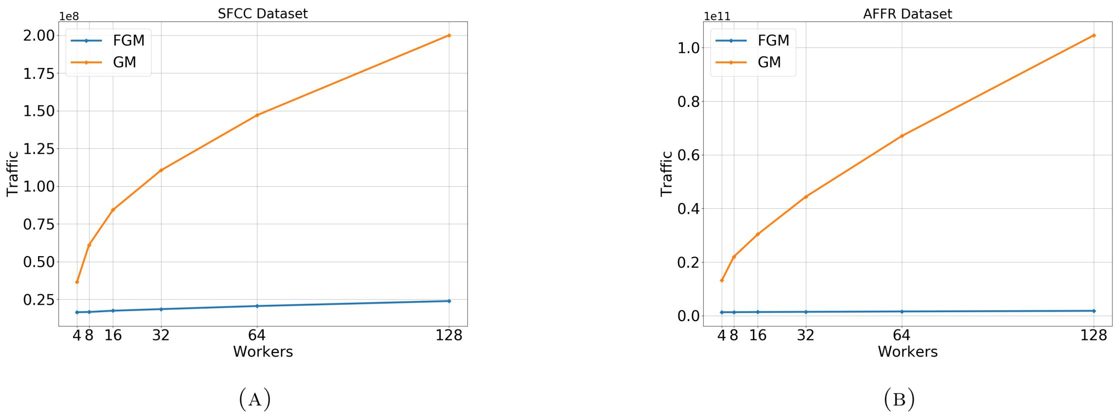
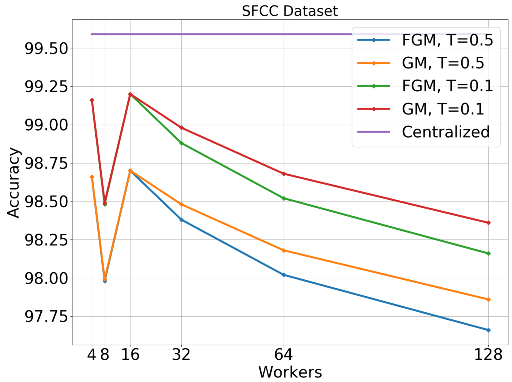
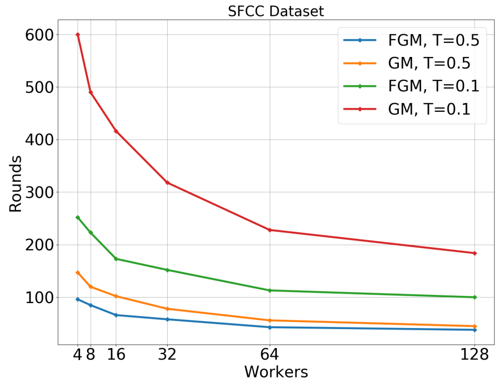


Figure 4.8

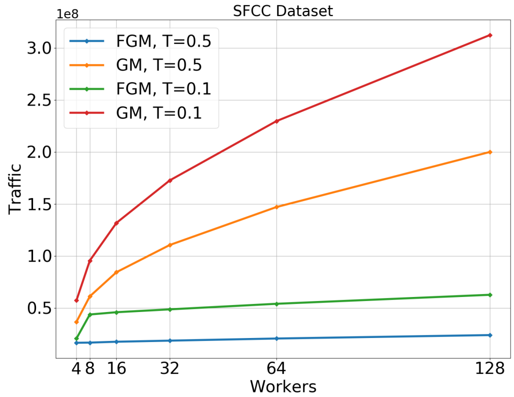
When I collected the experiments results, I and my professor noted that it would be interesting to show the above scenario for SFCC dataset for threshold equal to 0.1. Seeing the below figures, the most significant plot is in the figure 4.9 C, as the others follows the same direction having an absolute difference. So, we focus on the scalability analysis. With threshold equals to 0.1, FGM is better by 5.3 times than GM while with threshold equals to 0.5 FGM is better by 4.6 times.



(A)



(B)

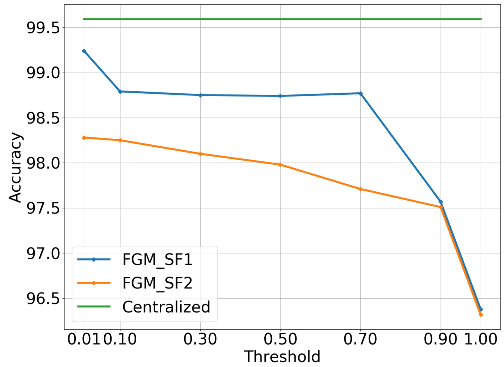


(C)

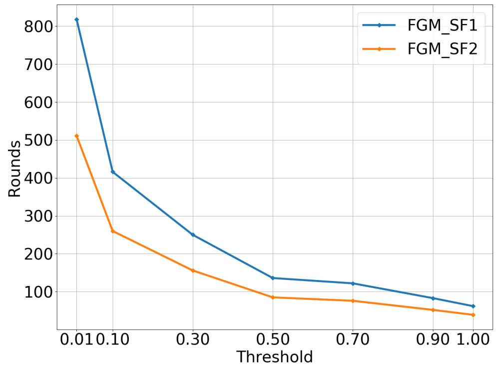
Figure 4.9

4.2.2 Safe functions comparison

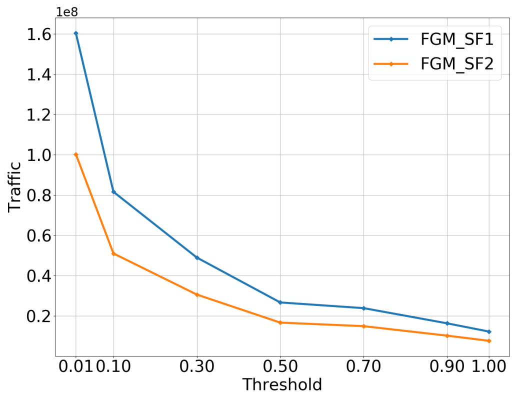
In the second part of this work, I wanted to make a comparison between the two safe functions. Michael Kamp used in his work the simple norm (Eq. 2.33) as his safe function. I used the spherical cap (Eq. 3.2). So, it remains to compare these functions to each other, to be able to decide which is the best for distributed Deep Learning. For the sake of brevity, I define the simple norm safe function as '*SF1*' and the spherical cap safe function as '*SF2*'.



(A)



(B)



(C)

Figure 4.10

When threshold is greater than 0.9 (Fig. 4.10 A), we see that both functions achieve the same accuracy. SF1 is more wasteful due to its geometry, so achieves more accuracy. This accuracy is approx up to 1%. For this reason, the number of rounds are more than the SF2 (Fig. 4.10 B). Especially, with threshold

< 0.1, the difference is the maximum between the two functions. In figure 4.10 C, both curves follows the same direction as the above, which is reasonable. In this case, rounds and traffic are proportional amounts.

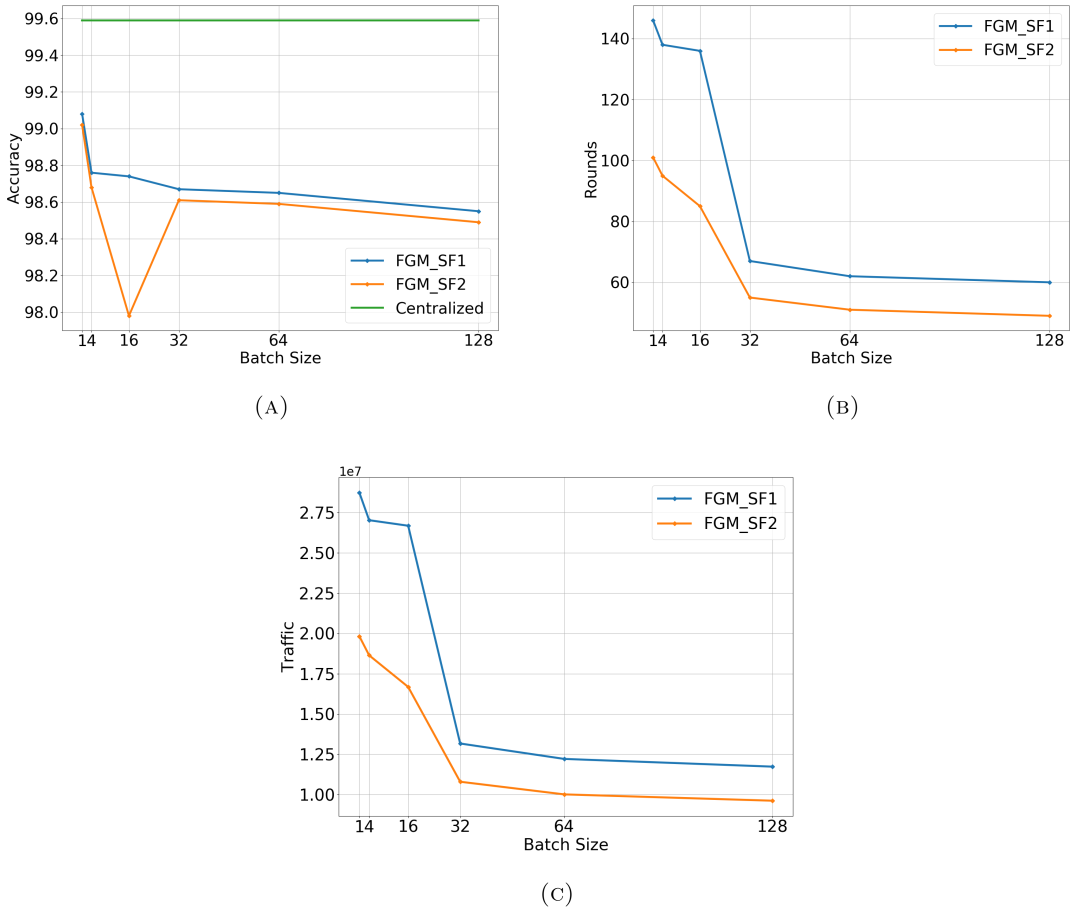
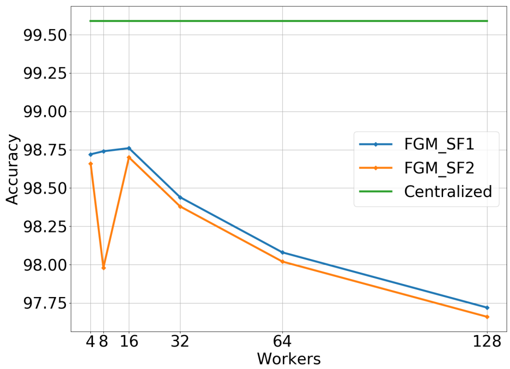
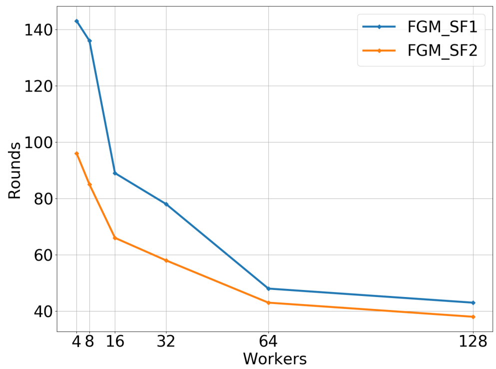


FIGURE 4.11

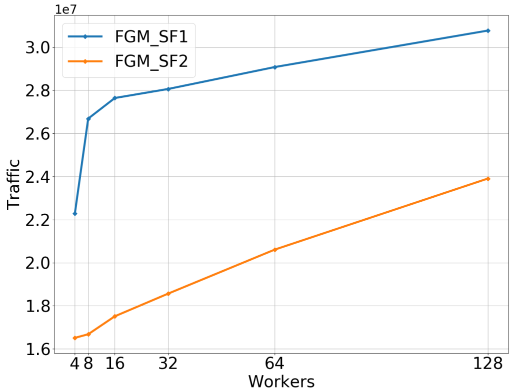
Here we have another one outlier (Fig. 4.11 A). Both safe functions have approximately the same accuracy when the batch size changes. In Fig. 4.11 B, we can detect that for small batch sizes, the functions reach up to 50% percentage change. In the same way (Fig. 4.11 C), SF2 is better up to 42%.



(A)



(B)



(C)

Figure 4.12

Both accuracy and number of rounds on the number of workers (Fig. 4.12 A, B) follow the expected direction. While the network is getting bigger (Fig. 4.12 C), SF2 is more efficient by up to 56%. The maximum difference is noted when the workers are 8. After that, the change is less about 10% compared to the maximum difference.

Chapter 5

Conclusions

5.1 Contribution

Our recommended method for distributed DL achieves high predictive performance, yet needs essentially less communication than GM. Furthermore, the method handles not only the learning algorithm but also the optimizer as black-boxes. In this work, I proved that FGM is better than GM for distributed DL learning and especially using LSTM networks, a subset of the Recurrent Neural Networks. I tested this architecture on solving two types of problems, classification, and sentiment analysis. In both cases, the results were impressive. But if we have to choose one of these two for which the architecture is more suitable, the answer is the NLP problem. Taking into account the difficulty of both problems, our architecture reacted almost in the same way in both cases.

In the second phase, I compared the two functions with each other. The results revealed that SF2 is much cheaper than SF1 in terms of network cost, but the latter achieves better accuracy on the prediction. Of course, this difference is not so important as to make us prefer it. Therefore, sacrificing minimal accuracy in the model, we choose SF2 as the best for distributed deep learning.

5.2 Future Work

In this work, I simulated a scenario calculating the network traffic cost of the training process of RNN by the GM and FGM protocol. We know this time in practice that the FGM protocol is more efficient than GM. Thus, a future direction would be an actual system that uses FGM to train these networks. Recently, Sofia Kampioti [10] implemented such a system to train an ML model for classification purposes using the Support Vector Machines (SVM) algorithm.

Another future direction would be the usage of the rebalancing version of FGM on RNN training. Using the rebalancing version, we can undoubtedly achieve much more efficient training concerning the network cost.

Last, in this project, we made offline learning. Future work could attempt to make this process online, taking into consideration some meaning like Concept Drift. An online learning algorithm can resolve some issues like concept changes. To make this more specific, in this task I used the food reviews as training samples. In an online learning system, we could change the concept of training samples to cloth reviews without accepting a large reduction in the forecast performance.

References

[1] Ryan R. Curtin et al. “mlpack 3: a fast, flexible machine learning library”. In: *Journal of Open Source Software* 3 (26 2018), p. 726. DOI: 10.21105/joss.00726. URL: https://doi.org/10.21105/joss.00726.

[2] *Detection and Classification of Adult and Fetal ECG Using Recurrent Neural networks , Embedded Volterra and Higher - Order Statistics* . en. OCLC: 1111192923. INTECH Open Access Publisher, 2012. ISBN: 978-953-51-0409-4. URL: http://www.intechopen.com/articles/show/title/detection-and-classification-of-biomedical-signals-using-embeded-volterra-and-higher-order-statistic (visited on 12/11/2019).

[3] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. 2nd ed. New York: Wiley, 2001. ISBN: 978-0-471-05669-0.

[4] Minos Garofalakis, Daniel Keren, and Vasilis Samoladas. “Sketch-based geometric monitoring of distributed stream queries”. In: *Proceedings of the VLDB Endowment* 6.10 (Aug. 26, 2013), pp. 937–948. ISSN: 2150-8097. DOI: 10.14778/2536206.2536220. URL: http://dl.acm.org/doi/10.14778/2536206.2536220 (visited on 06/02/2020).

[5] Minos Garofalakis and Vasilis Samoladas. “Distributed Query Monitoring through Convex Analysis: Towards Composable Safe Zones”. In: (), p. 18.

[6] Barbara Hammer. *Learning with recurrent neural networks*. en. Lecture notes in control and information sciences 253. London ; New York: Springer, 2000. ISBN: 978-1-85233-343-0.

[7] J. D. Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[8] Michael Kamp et al. “Communication- Efficient Distributed Online Prediction by Dynamic Model Synchronization”. en. In: 8724 (2014). Ed. by Toon Calders et al., pp. 623–639. DOI: 10.1007/978-3-662-44848-9_40. URL: http://link.springer.com/10.1007/978-3-662-44848-9_40 (visited on 12/09/2019).

[9] Michael Kamp et al. "Efficient Decentralized Deep Learning by Dynamic Model Averaging". en. In: *arXiv:1807.03210 [cs, stat]* (Nov. 2018). arXiv: 1807.03210. URL: http://arxiv.org/abs/1807.03210 (visited on 12/09/2019).

[10] Sofia Kampioti. "A functional geometric approach to distributed support vector machine (SVM) classification". en. In: (2020). URL: http://purl.tuc.gr/dl/dias/3C8837A2-7745-4025-B874-D756A30E6261.

[11] Vissarion Konidaris. "Distributed Machine Learning Algorithms via Geometric Monitoring". en. In: (2019). URL: http://purl.tuc.gr/dl/dias/07D2D80D-F1AC-4937-B5F6-69D82C9A996F.

[12] Vissarion B. Konidaris. *Distributed Online Machine Learning*. https://github.com/ArisKonidaris/Distributed_Online_Machine_Learning. 2019.

[13] Mu Li. "Scaling Distributed Machine Learning with the Parameter Server". en. In: *Proceedings of the 2014 International Conference on Big Data Science and Computing - BigDataScience '14*. Beijing, China: ACM Press, 2014. ISBN: 978-1-4503-2891-3. DOI: 10.1145/2640087.2644155. URL: http://dl.acm.org/citation.cfm?doid=2640087.2644155.

[14] SF OpenData. *San Francisco Crime Classification*. Aug. 2016. URL: https://www.kaggle.com/kaggle/san-francisco-crime-classification.

[15] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[16] Stanford Network Analysis Project. *Amazon Fine Food Reviews*. May 2017. URL: https://www.kaggle.com/snap/amazon-fine-food-reviews.

[17] Rushabhwadkar. *LSTM on Amazon Fine Food Reviews*. June 2019. URL: https://www.kaggle.com/rushabhwadkar/lstm-on-amazon-fine-food-reviews.

[18] Vasileios Samoladas. *ddssim*. https://github.com/vsamtuc/ddssim. 2018.

[19] Vasilis Samoladas and Minos Garofalakis. "Functional Geometric Monitoring for Distributed Streams". In: (), p. 12.

[20] Izchak Sharfman, Assaf Schuster, and Daniel Keren. "A geometric approach to monitoring threshold functions over distributed data streams". In: *ACM Transactions on Database Systems* 32.4 (Nov. 2007), p. 23. ISSN: 0362-5915, 1557-4644. DOI: 10.1145/1292609.1292613. URL: https://dl.acm.org/doi/10.1145/1292609.1292613 (visited on 06/02/2020).

[21] Izchak Sharfman, Assaf Schuster, and Daniel Keren. "Aggregate Threshold Queries in Sensor Networks". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. 2007 IEEE International Parallel and Distributed Processing Symposium. Long Beach, CA , USA:

IEEE, 2007, pp. 1–10. ISBN: 978-1-4244-0909-9. DOI: 10.1109/IPDPS.2007.370297. URL: http://ieeexplore.ieee.org/document/4228025/ (visited on 06/02/2020).

[22] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. URL: https://doi.org/10.5281/zenodo.3509134.

[23] Jie Zhang. *multi-class-text-classification-cnn-rnn*. https://github.com/jiegzhan/multi-class-text-classification-cnn-rnn. 2018.

Appendix A

Abbreviations

ML	Machine Learning
AI	Artificial Intelligence
RL	Reinforcement Learning
MDP	Markov Decision Process
DL	Deep Learning
DML	Distributed Machine Learning
ANN	Artificial Neural Network
NN	Neural network
FFNN	Feed-Forward Neural Network
MSE	Mean Squared Error
BP	Backpropagation
DNN	Deep Neural Network
GD	Gradient Descent
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
BPTT	Backpropagation Through Time
SGD	Stochastic Gradient Descent
LSTM	Long Short Term Memory
GM	Geometric Monitoring
FGM	Functional Geometric Monitoring
NLP	Natural Language Processing
SFCC	San Francisco Crime Classification
AFFR	Amazon Fine Food Reviews
SVM	Support Vector Machines

Appendix B

Detailed Experimental Results

B.1 SFCC Dataset Results

id	Threshold	Batch Size	Workers	Rounds	Rebalances	Accuracy	Traffic (bytes)
1	0.01	16	8	1296	8424	99.42	660,214,152
2	0.1	16	8	781	5075	99.01	397,758,650
3	0.3	16	8	589	3827	98.97	299,949,146
4	0.5	16	8	192	1248	98.74	97,809,504
5	0.7	16	8	149	968	98.77	75,802,366
6	0.9	16	8	98	635	98.25	49,719,832
7	1	16	8	67	437	97.36	34,233,326
8	0.5	1	8	222	1443	99.18	113,015,826
9	0.5	4	8	198	1235	98.78	96,765,185
10	0.5	16	8	192	1248	98.74	97,809,504
11	0.5	32	8	78	510	98.67	39,938,881
12	0.5	64	8	76	495	98.65	38,797,770
13	0.5	128	8	74	480	98.55	37,656,659
14	0.5	16	4	198	1291	98.72	49,516,061
15	0.5	16	8	192	1248	98.74	97,809,504
16	0.5	16	16	138	895	98.76	113,886,941
17	0.5	16	32	105	684	98.54	149,191,893
18	0.5	16	64	76	491	98.24	198,425,217
19	0.5	16	128	61	396	97.92	269,858,295

Table B.1. (SFCC) Training by GM protocol using as safe function the *'simple norm'*

id	Threshold	Batch Size	Workers	Rounds	Rebalances	Accuracy	Traffic (bytes)
1	0.01	16	8	810	5,265	98.31	412,633,845
2	0.1	16	8	488	3,172	98.29	248,599,156
3	0.3	16	8	368	2,392	98.3	187,468,216
4	0.5	16	8	120	780	97.99	61,130,940
5	0.7	16	8	93	605	97.81	47,376,479
6	0.9	16	8	61	397	97.63	31,074,895
7	1	16	8	42	273	96.45	21,395,829
8	0.5	1	8	153	995	99.12	77,941,949
9	0.5	4	8	131	852	98.72	66,734,610
10	0.5	16	8	120	780	97.99	61,130,940
11	0.5	32	8	70	455	98.61	35,659,715
12	0.5	64	8	68	442	98.59	34,640,866
13	0.5	128	8	66	429	98.49	33,622,017
14	0.5	16	4	147	956	98.66	36,678,564
15	0.5	16	8	120	780	97.99	61,130,940
16	0.5	16	16	102	663	98.7	84,360,697
17	0.5	16	32	78	507	98.48	110,512,513
18	0.5	16	64	56	364	98.18	146,981,642
19	0.5	16	128	45	293	97.86	199,895,033

Table B.2. (SFCC) Training by GM protocol using as safe function the *'spherical cap'*

id	Threshold	Batch Size	Workers	Rounds	Subrounds	Accuracy	Traffic (bytes)
1	0.01	16	8	818	6133	99.24	160,453,182
2	0.1	16	8	416	3120	98.79	81,639,584
3	0.3	16	8	250	1872	98.75	48,983,750
4	0.5	16	8	136	1021	98.74	26,689,864
5	0.7	16	8	122	912	98.77	23,863,878
6	0.9	16	8	83	624	97.57	16,327,917
7	1	16	8	62	469	96.38	12,245,938
8	0.5	1	8	146	1099	99.08	28,740,666
9	0.5	4	8	138	1034	98.76	27,033,300
10	0.5	16	8	136	1021	98.74	26,689,864
11	0.5	32	8	67	504	98.67	13,168,308
12	0.5	64	8	62	467	98.65	12,210,613
13	0.5	128	8	60	449	98.55	11,731,765
14	0.5	16	4	143	972	98.72	22,294,377
15	0.5	16	8	136	1021	98.74	26,689,864
16	0.5	16	16	89	668	98.76	27,645,551
17	0.5	16	32	78	587	98.44	28,064,284
18	0.5	16	64	48	362	98.08	29,081,419
19	0.5	16	128	43	319	97.72	30,774,446

Table B.3. (SFCC) Training by FGM protocol using as safe function the *'simple norm'*

id	Threshold	Batch Size	Workers	Rounds	Subrounds	Accuracy	Traffic (bytes)
1	0.01	16	8	511	3,833	98.28	100,283,239
2	0.1	16	8	260	1,950	98.25	51,024,740
3	0.3	16	8	156	1,170	98.1	30,614,844
4	0.5	16	8	85	638	97.98	16,681,165
5	0.7	16	8	76	570	97.71	14,914,924
6	0.9	16	8	52	390	97.51	10,204,948
7	1	16	8	39	293	96.32	7,653,711
8	0.5	1	8	101	758	99.02	19,821,149
9	0.5	4	8	95	713	98.68	18,643,655
10	0.5	16	8	85	638	97.98	16,681,165
11	0.5	32	8	55	413	98.61	10,793,695
12	0.5	64	8	51	383	98.59	10,008,699
13	0.5	128	8	49	368	98.49	9,616,201
14	0.5	16	4	96	720	98.66	16,514,353
15	0.5	16	8	85	638	97.98	16,681,165
16	0.5	16	16	66	495	98.7	17,515,223
17	0.5	16	32	58	435	98.38	18,566,136
18	0.5	16	64	43	323	98.02	20,608,410
19	0.5	16	128	38	285	97.66	23,905,755

Table B.4. (SFCC) Training by FGM protocol using as safe function the *'spherical cap'*

B.2 AFFR Dataset Results

id	Threshold	Batch Size	Workers	Rounds	Rebalances	Accuracy	Traffic (bytes)
1	0.01	16	8	6747	8859	98.07	56,461,271,587
2	0.1	16	8	2542	712	97.48	18,657,422,040
3	0.3	16	8	1285	1037	97.1	15,478,253,016
4	0.5	16	8	1203	848	96.99	14,513,127,048
5	0.7	16	8	768	581	96.98	9,975,494,304
6	0.9	16	8	605	483	96.65	8,292,636,408
7	1	16	8	293	389	96.54	6,192,102,888
8	0.5	1	8	1492	2416	97.38	62,086,540,644
9	0.5	4	8	1418	3795	97.22	40,020,099,414
10	0.5	16	8	1203	848	96.99	14,513,127,048
11	0.5	32	8	677	1877	96.6	19,996,243,164
12	0.5	64	8	662	1866	96.59	19,996,242,993
13	0.5	128	8	633	1807	96.56	19,996,241,454
14	0.5	16	4	1296	2075	96.98	17,851,868,737
15	0.5	16	8	1203	848	96.99	14,513,127,048
16	0.5	16	16	898	975	96.97	41,059,298,095
17	0.5	16	32	340	834	96.94	59,946,575,218
18	0.5	16	64	205	762	96.78	75,097,517,043
19	0.5	16	128	95	407	96.67	117,152,126,588

Table B.5. (AFFR) Training by GM protocol using as safe function the *'simple norm'*

id	Threshold	Batch Size	Workers	Rounds	Rebalances	Accuracy	Traffic (bytes)
1	0.01	16	8	4,217	5,537	98.01	35,288,294,742
2	0.1	16	8	1,589	445	97.42	11,660,888,775
3	0.3	16	8	803	648	97.04	9,673,908,135
4	0.5	16	8	752	530	96.93	9,070,704,405
5	0.7	16	8	480	363	96.92	6,234,683,940
6	0.9	16	8	378	302	96.59	5,182,897,755
7	1	16	8	183	243	96.48	3,870,064,305
8	0.5	1	8	1,223	1,980	97.32	50,890,607,085
9	0.5	4	8	781	2,090	97.05	22,039,344,380
10	0.5	16	8	752	530	96.93	22,039,344,120
11	0.5	32	8	373	1034	96.42	11,012,068,832
12	0.5	64	8	365	1028	96.41	11,012,068,738
13	0.5	128	8	348	995	96.38	11,012,067,890
14	0.5	16	4	960	1537	96.92	13,223,606,472
15	0.5	16	8	752	530	96.93	22,039,344,120
16	0.5	16	16	665	722	96.91	30,414,294,885
17	0.5	16	32	252	618	96.88	44,404,870,532
18	0.5	16	64	183	680	96.72	67,051,354,503
19	0.5	16	128	85	363	96.61	104,600,113,025

Table B.6. (AFFR) Training by GM protocol using as safe function the *'spherical cap'*

id	Threshold	Batch Size	Workers	Rounds	Subrounds	Accuracy	Traffic (bytes)
1	0.01	16	8	4498	4498	98.04	48,636,137,741
2	0.1	16	8	1694	1760	96.97	18,322,918,669
3	0.3	16	8	856	1085	96.61	9,256,664,346
4	0.5	16	8	802	1062	96.52	8,668,401,869
5	0.7	16	8	512	869	96.47	5,536,744,634
6	0.9	16	8	403	635	96.17	4,360,177,235
7	1	16	8	195	296	96.07	2,110,876,838
8	0.5	1	8	633	743	97.83	6,846,804,429
9	0.5	4	8	601	657	97.51	6,494,812,347
10	0.5	16	8	802	1062	96.52	8,668,401,869
11	0.5	32	8	622	784	97.03	6,732,350,582
12	0.5	64	8	569	754	97.02	6,150,543,055
13	0.5	128	8	395	542	96.97	4,280,445,178
14	0.5	16	4	802	1062	96.52	8,668,401,869
15	0.5	16	8	676	896	96.52	1,799,367,206
16	0.5	16	16	173	470	96.5	1,890,244,338
17	0.5	16	32	135	410	96.46	1,984,756,554
18	0.5	16	64	64	241	96.27	1,811,274,129
19	0.5	16	128	55	109	96.19	2,064,852,507

Table B.7. (AFFR) Training by FGM protocol using as safe function the *'simple norm'*

id	Threshold	Batch Size	Workers	Rounds	Subrounds	Accuracy	Traffic (bytes)
1	0.01	16	8	2,811	2,811	97.98	30,397,586,088
2	0.1	16	8	1,059	1,100	96.91	11,451,824,168
3	0.3	16	8	535	678	96.55	5,785,415,216
4	0.5	16	8	501	664	96.46	5,417,751,168
5	0.7	16	8	320	543	96.41	3,460,465,396
6	0.9	16	8	252	397	96.11	2,725,110,772
7	1	16	8	122	185	96.01	1,319,298,024
8	0.5	1	8	633	743	97.83	6,846,804,429
9	0.5	4	8	601	657	97.51	6,494,812,347
10	0.5	16	8	501	664	96.46	5,417,751,168
11	0.5	32	8	510	643	96.97	5,518,320,149
12	0.5	64	8	466	618	96.96	5,041,428,734
13	0.5	128	8	324	444	96.91	3,508,561,621
14	0.5	16	4	501	664	96.43	1,332,864,597
15	0.5	16	8	249	321	96.46	1,346,327,876
16	0.5	16	16	128	348	96.44	1,400,180,991
17	0.5	16	32	100	304	96.4	1,470,190,040
18	0.5	16	64	57	215	96.21	1,617,209,044
19	0.5	16	128	49	97	96.13	1,843,618,310

Table B.8. (AFFR) Training by FGM protocol using as safe function the *'spherical cap'*