INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Efficient algorithms for Tensor Network Training

- Research Initiation Project -

Author :
Victor BERTRET

Supervisor :
M. Jérémy COHEN , CNRS
PANAMA, IRISA Rennes

Applied Mathematics Department - 4th Year
Academic Year 2020-2021

Abstract

The aim of this study is to develop efficient algorithms for tensor network training. Miles Stoudenmire and David Schwab have published an experience where they studied the classification of images from the MNIST database [1] which contains handwritten digits using a tensor network in the article "Supervised Learning with Tensor Networks" [2]. We tried to reproduce the experiments in a first report [3] but the algorithm had vanishing gradients and normalization issues. To achieve our goals we improved the algorithmic procedure : constant initialization, new loss functions such as cross-entropy and new optimization algorithms like conjugate gradient method. These modifications led to a significant improvement in the classification performance on the MNIST database : the algorithm is able to classify the images used during the training. However, the large computation time prevented the model from being trained on the entire database to have a good generalization. Research perspectives can be the parallelization of the computation on the inputs or trying other tensor network models and other mapping functions.

Contents

Introduction

Recently, there has been a big development of machine learning algorithms in different areas of society such as image classification, data forecasting , text analysis... This was made possible by the abundance of data and the speed up of the computation power of computers. In this context, tensor methods have been developed in machine learning, especially in physics as dimensionality reduction tool.

This project focus on the study of tensor networks and kernel-based method : the kernel transforms the input data into a larger dimensional space non-linearly and the tensor network tries to make operations in this new space. Some kinds of tensor networks are presented on the image 1 below.

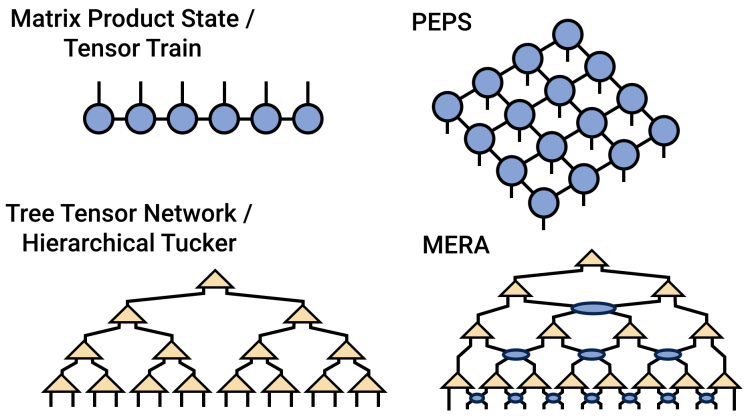

Figure 1: Presentation of Tensor Networks (`https://tensornetwork.org/`)

Miles Stoudenmire and David Schwab have published an experience in the article "Supervised Learning with Tensor Networks" [2]. In this paper, they studied the classification of images from the MNIST database [1] which contains handwritten digits. They used a map f(x) from the input space to the decision space such that $f(x) = W.\Phi(x)$ where W is a matrix product state (MPS) tensor network, and $\Phi(.)$ is a nonlinear tensor kernel inspired from quantum physics. A presentation of the MNIST Database is depicted on the figure 2.

In a first report [3], we tried to reproduce the experiments. Unfortunately, we weren't able to have the same results : the training of the model was inefficient, we had vanishing gradients and normalization issues. Consequently, in this study, we are going to develop efficient algorithms for

tensor network training in order to overcome the issues mentioned in the first report. This project was done in the Panama Team at IRISA. My study was supervised by Jéremy Cohen whom I thank for his availability and advice throughout my research.

This report is organized as follow : first we do a reminder on the model, algorithms and the issues of the first report. Then, we test new functionalities on algorithms such as constant initialization, new loss functions and optimization algorithms. Finally, we test our new approach on the MNIST handwritten digit database where we find the same results as Miles Stoudenmire and David Schwab.

For someone interested in reproducing our results, we have made our codes publicly available at: `https://github.com/victorius35/PIR-Tensor-Network-MNIST`. The codes are based on the Numpy [4] and Tensorly [5] libraries in Python.

Figure 2: MNIST Database

Chapter 1

Reminder on tensor networks for classification

The first report dealt with different models of tensor network. At first, no particular tensor structure was imposed on the clustering parameter tensor W. However, this model was intractable and was taking too much time even on a small example. Then, a second model was tested. This model performs an approximation of the big tensor with a Matrix Product State (MPS) form. This one is easier to train in higher dimensions but it still has some issues. Therefore, this part make reminders on the second model, the optimization algorithms used and the problems.

1.1 Model

In order to construct the model, the data must be clear and well defined. In the study, the data is represented by N_t input vectors x_i of dimension N and labels $y_i \ \forall i \in \{1, \dots, N_t\}$. The features of x_i are noted $(x_i^1, \dots, x_i^N)$. For example, if inputs are pictures, they are represented by the values of the pixels of the black and white picture x_i. Finally, there are in total N_l classes grouped into a set $\mathcal{L}$ so $y_i \in \{1, \dots, N_l\} \ \forall i \in \{1, \dots, N_t\}$.

The model consist of a non-linear embedding Φ which maps the input data to a higher dimensional space, and a tensor W which contains the parameters trained to perform clustering. Thus, the decision function is defined by their contraction :

$$\forall l \in \mathcal{L}, f^l(x) = W^l.\Phi(x) \tag{1.1}$$

Parametrization of Φ In the project, the mapping Φ is defined as :

$$\begin{gathered}\forall i \in \{1, ..., N_t\}, \Phi^{s_1 s_2 \dots s_N}(x_i) = \phi^{s_1}(x_i^1) \otimes \phi^{s_2}(x_i^2) \otimes ... \otimes \phi^{s_N}(x_i^N) \\ \text{with } \phi^{s_j}(x_i^j) = [\cos(\frac{\pi}{2} x_i^j), \sin(\frac{\pi}{2} x_i^j)], \quad \forall j \in \{1, ..., N\}\end{gathered} \tag{1.2}$$

The mapping Φ comes from physics where N independent systems combine as a tensor product of vectors describing each system ([6], [7]) . Moreover, each function ϕ allow to transform one pixel to a normalized two-component vector (d=2) thus $\phi^{s_j}(x_i^j)$ represents $\forall s_i \in \{0, 1\}$ the value of the vector.

Finally, the tensor is a N order tensor. Here is the tensor diagram for Φ :

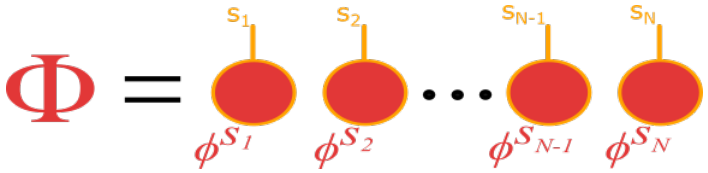

Figure 1.1: Representation of the tensor Φ

Parametrization of W The tensor W is a N+1 order tensor with 1 order additional for the labels. This tensor needs to be trained efficiently.

Therefore the tensor W is represented by a Matrix Product State form. In other words, W is factorized into a chain of N product of three order tensors and therefore every element of the tensor can be written as :

$$\forall s_1, ..., s_N, l,\ W^l_{s_1,...,s_N} = \sum_{\alpha_1,...,\alpha_{N-1}} A^{\alpha_1}_{s_1} ... A^{l;\alpha_i\alpha_{i+1}}_{s_i} ... A^{\alpha_{N-1}}_{s_N} \tag{1.3}$$

The upper bounds of the indexes α are called the bond dimensions, it's an important parameter. The larger the alpha, the more expressive the decomposition, but the larger the number of parameters. In the rest of the report, the bond dimension will be the same for all bonds and often be denoted by m. The tensors A will be called the factors A in the rest of the report. Here is the tensor diagram for W:

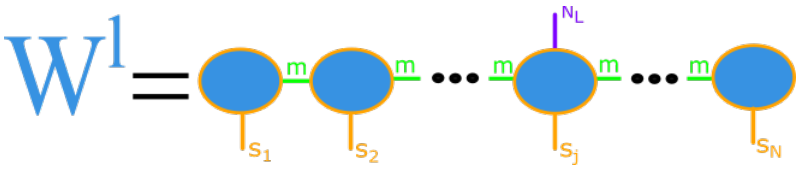

Figure 1.2: Representation of the tensor W

The decision function can now be represented as :

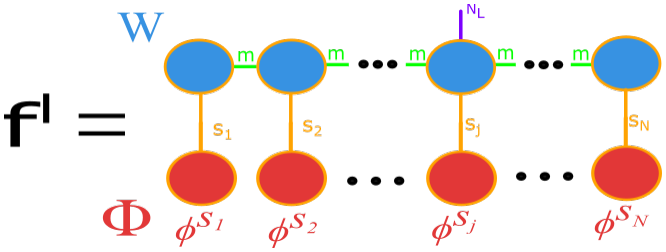

Figure 1.3: Representation of the decision function

Finally, the value of the label associated with an input x is determined by the maximum of the function f^l over $l \in \mathcal{L}$. Thus, to classify inputs, the following function is used :

$$\arg\max_{l \in \mathcal{L}} f^l(x_i) \tag{1.4}$$

1.2 Algorithms

The first report introduces two kind of algorithms : GD for gradient descent and DMRG for Density Matrix Renormalization Group.

Firstly, the 2 algorithms use the same loss function : the quadratic loss. To introduce it, labels need to be in one hot encoding format. In other words, if the correct label of x_i is L_i, then $y_i^{L_i} = 1$ and $y_i^l = 0$ for all other labels l. Then, the quadratic loss is defined as :

$$C(W) = \frac{1}{2N_t} \sum_{n=1}^{N_T} \sum_{l} (f^l(X_n) - y_n^l)^2 = \frac{1}{2N_t} \sum_{n=1}^{N_T} \sum_{l} (W^l.\Phi(X_n) - y_n^l)^2 \tag{1.5}$$

The goal of an algorithm is to minimize the cost function. There exists two types of strategy :

- update one tensor A at a time, it's the strategy of GD
- update two tensors A at a time, it's the strategy of DMRG

The two approaches work but the second one has the advantage to adaptively change the bond dimension during the optimization. That's why, in the rest of this report, we will only focus on the DMRG algorithm and forget the GD algorithm.

DMRG Algorithm The DMRG algorithm sweeps the MPS form of W, iteratively minimizing the cost function by updating two tensors at a time. Here is the sequence of the algorithm :

- The first step is to contract two neighbouring tensors A according to their common index to make a tensor B.
- The second step is to compute the tensor $\tilde{\Phi}$ as described in the first report or on the diagram 1.4 . The cost function can now be defined as :

$$C(B) = \frac{1}{2N_t} \sum_{n=1}^{N_T} \sum_{l=1}^{N_l} (B.\tilde{\Phi}(X_n) - y_n^l)^2 \tag{1.6}$$

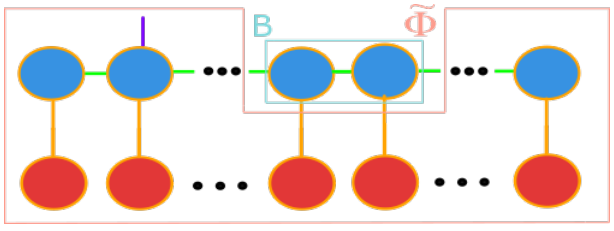

Figure 1.4: Construction of $\tilde{\Phi}$

It is shown on the equation 1.6 that the cost is quadratic according to B.

- Then, the algorithm computes the gradient of the cost function according to B. It has been shown in the first report that the gradient is defined as follows :

$$\frac{\partial C(B)}{\partial B} = \begin{cases} \frac{1}{N_t}\sum_{n=1}^{N_T}(B.\tilde{\Phi}^l(X_n) - y_n^l).\tilde{\Phi}^l(X_n) & \text{if } l \notin B \\ \frac{1}{N_t}\sum_{n=1}^{N_T}(B^l.\tilde{\Phi}(X_n) - y_n^l) \otimes \tilde{\Phi}(X_n) & \text{if } l \in B \end{cases} \tag{1.7}$$

- Next, it is necessary to make a gradient descent step with a fixed step size α so $B = B - \alpha\frac{\partial C(W)}{\partial B}$.
- Finally, the last step is to decompose the tensor B into two separate tensors to recover the MPS form. A solution is to compute a singular value decomposition (SVD) as described in the first report.

  Either multiply the singular values from the SVD in the direction of the sweeping or multiply the singular values in the opposite direction named respectively DMRG-SD and DMRG-OD (as the first report was in french, the name were DMRG-MSENS and DMRG-SENSI).

This last step is the key to the algorithm. Indeed, throughout the iterations, the algorithm can only keep the m largest singular values and therefore the rest are truncated, so the bond dimension is chosen adaptively during the execution. To keep only the m largest singular values, the algorithm gets rid of singular values lower than a threshold and has a maximal number of singular values to keep.

1.3 Issues

Firstly, the problem was the initialization. The factors A of W were either initialized with a constant or with entries drawn from a normal distribution. A representation of the problem is depicted on the following graph 1.5 on a simple random example with N=50.

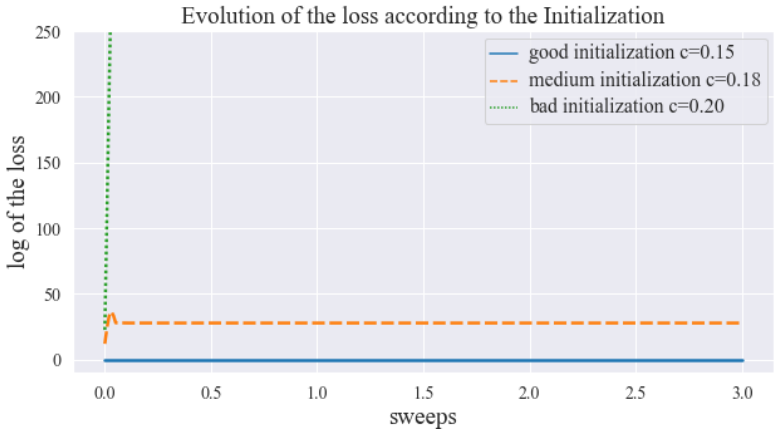

Figure 1.5: Issue with constant initialization

As it's shown on the graphic, a small variation of the constant for constant initialization can cause explosions of the loss. Therefore, at each test when we changed the data or a parameters like the bond dimension, we had to change the value.

Secondly, another problem was the stability of the gradient descent steps. Indeed, at each bond, the step size is the same in the DMRG algorithm and this is a problem. The gradient at each bond is not the same and so the optimal learning rate is not the same too. Moreover, if the step size is too large for one bond, the cost can increase. Thus, the learning rate must respect the largest value accepted at each bond. Consequently, the step size is too small for numerous bonds and it results in a slow training.

Finally, DMRG-SD has a vanishing gradients problem and DMRG-OD has an exploding gradients problem. Indeed, in the first case, all the singular values are multiplied with the factor A which will be in B at the next bond and the other tensor A is unitary. Therefore, iteratively, all the factors A which are part of the gradient are unitaries and the values of $\tilde{\Phi}$ are near 0 as it's the contraction of these factors. That's why, the gradient becomes very small and one gradient descent step does not make a lot of decrease for the loss. On the contrary, the gradient of DMRG-OD becomes very large and explodes.

A representation of the two last issues is presented on the graph 1.6 for N equal 100.

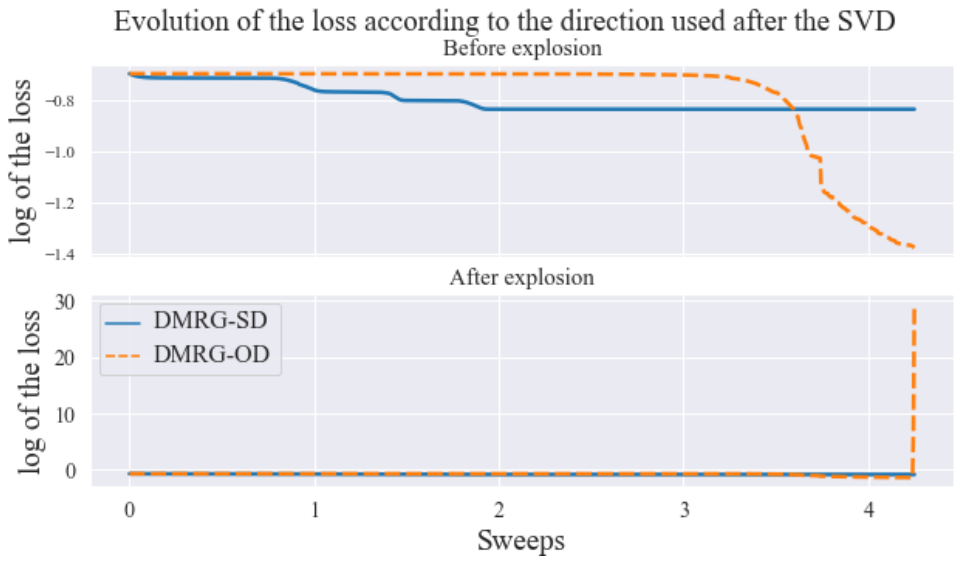

Figure 1.6: Issues algorithms DMRG

On the first graphic, the log of the loss for DMRG-SD is not decreasing after the second sweep : it's the problem of vanishing gradients. On the contrary, on the second graphic, the log of the loss for DMRG-OD explodes in just one gradient descent step : it's the problem of exploding gradients. The step size is fixed so it's not possible to change it to avoid vanishing or exploding gradients.

In the first report, we have developed an algorithm in order to train a MPS tensor network. The algorithm works well with small values of N, the number of features, but has some issues when N gets bigger. In the next chapter, we will try to develop solutions to the problems.

Chapter 2

Improving the efficiency of the algorithms

The DMRG algorithm has some difficulties to converge to a satisfying solution when the number of features per input is large because of issues mentioned in the previous section. The project dealt with the research of efficient algorithms to train a tensor network. In this chapter, some solutions are going to be developed.

2.1 Initialization

It has been shown in the first chapter that the initialization is a very challenging task. As there are a lot of parameters, a little change can create big consequences on the values of the decision function. That's why, this section deals with a new way to initialize tensors.

The problem with the initialization is that when the values of the decision function are too large, it triggers an explosion of the gradient. Therefore, the goal is to avoid big values but also to avoid too small value. Indeed, if all the values of the tensors are near 0, the gradient is near 0 too and so the training is inefficient.

Firstly, in order to control the decision function, it can be developed as :

$$\begin{aligned}\forall l \in \mathcal{L}, \quad f^l(X) = W^l.\Phi(X) &= \sum_{s_1,\dots,s_N} W^l_{s_1,\dots,s_N}.\phi^{s_1}(x_1)\dots\phi^{s_N}(x_N) \\ &= \sum_{\substack{s_1,\dots,s_N \\ \alpha_1,\dots,\alpha_{N-1}}} A^{\alpha_1}_{s_1}\dots A^{l;\alpha_{i-1},\alpha_i}_{s_i}\dots A^{\alpha_{N-1}}_{s_N}.\phi^{s_1}(x_1)\dots\phi^{s_N}(x_N)\end{aligned}$$

In our case, we decided to implement a constant initialization. Therefore, let's H be the constant affected to all the factors. Now, each value of a factor A is equal to H so $A^{\alpha_1}_{s_1} = \dots = A^{l;\alpha_{i-1},\alpha_i}_{s_i} = \dots = A^{\alpha_{N-1}}_{s_N} = H$. The decision function can now be developed as :

$$\begin{aligned}\forall l \in \mathcal{L}, \quad f^l(X) &= \sum_{\substack{s_1,\dots,s_N \\ \alpha_1,\dots,\alpha_{N-1}}} H \dots H \dots H.\phi^{s_1}(x_1)\dots\phi^{s_N}(x_N) \\ &= H^N \sum_{\substack{s_1,\dots,s_N \\ \alpha_1,\dots,\alpha_{N-1}}} 1\dots1\dots1.\phi^{s_1}(x_1)\dots\phi^{s_N}(x_N) \\ &= H^N \mathbb{1}^l.\Phi(X)\end{aligned}$$

with $\mathbb{1}^l$ a tensor with the same shape as W but with only 1 for the values of the factors. It's now possible to compute the value of the decision function for every input. However, the contraction between $\mathbb{1}^l$ and Φ has the same cost as the contraction of the decision function, it would be better to have a formula with the input features $(x_i^1, \dots, x_i^N)$ directly.

To compute this value, it's possible to separate each step thanks to the MPS form of W^l and that of $\mathbb{1}^l$. Firstly, it's possible to contract each tensor ϕ^{s_i} with the factor A_{s_i} connected with it to obtain a new tensor denoted T_i. There are two cases :

- If i is equal to 1 or N,

$$T_i^\alpha = \sum_{s_i=0}^{1} \mathbb{1}_{s_i}^\alpha.\phi^{s_i}(x_i)$$

For all α, when s_i is equal to 0 or 1, $\mathbb{1}_{s_i}^\alpha$ is a vector with dimension equal to the bond dimension between A and its neighbour. Moreover, for any α, the vector $\mathbb{1}_{s_i}^\alpha$ is the same because it was initialized with the same values. In our case, we have considered that for the initialization the bond dimension is the same for each bond and is denoted by m so the length of the vector is m and the vector will be denoted by $\mathbb{1}_m^\alpha$.

Therefore, $T_i^\alpha = \mathbb{1}_m^\alpha \cos(\frac{\pi}{2}x^i) + \mathbb{1}_m^\alpha \sin(\frac{\pi}{2}x^i) = \mathbb{1}_m^\alpha(\cos(\frac{\pi}{2}x^i) + \sin(\frac{\pi}{2}x^i))$

- If i is different from 1 or N, it's the same result except that $\mathbb{1}_m^\alpha$ is a matrix as $\alpha = (\alpha_{i-1}, \alpha_i)$. It's a square m×m matrix.

Therefore, $T_i^\alpha = \mathbb{1}_m^{\alpha_{i-1},\alpha_i} \cos(\frac{\pi}{2}x^i) + \mathbb{1}_m^{\alpha_{i-1},\alpha_i} \sin(\frac{\pi}{2}x^i) = \mathbb{1}_m^{\alpha_{i-1},\alpha_i}(\cos(\frac{\pi}{2}x^i) + \sin(\frac{\pi}{2}x^i))$

Finally, to complete the computation of the contraction, it's necessary to make the contractions between the tensors T_i^α. As $\mathbb{1}^l$ has the same shape as W, it's only necessary to make all the contractions between two neighbouring tensors. The contraction between $\mathbb{1}^l$ and Φ can be written as follow :

$$\begin{aligned}\forall l \in \mathcal{L}, \quad \mathbb{1}^l.\Phi(X) &= T_1^{\alpha_1}.\dots.T_i^{l;\alpha_{i-1}\alpha_i}.\dots.T_N^{\alpha_{N-1}} \\ &= [\prod_{i=1}^{N}(\cos(\frac{\pi}{2}x^i) + \sin(\frac{\pi}{2}x^i))]\mathbb{1}_m^{\alpha_1}.\mathbb{1}_m^{\alpha_1,\alpha_2}...\mathbb{1}_m^{\alpha_{N-2},\alpha_{N-1}}.\mathbb{1}_m^{\alpha_{N-1}}\end{aligned}$$

Given that $\mathbb{1}_m^{\alpha_1}.\mathbb{1}_m^{\alpha_1,\alpha_2}$ is simply a matrix-vector product, $\forall\alpha_2 \quad \mathbb{1}_m^{\alpha_1}.\mathbb{1}_m^{\alpha_1,\alpha_2} = \sum_{\alpha_1=1}^{m} 1*1 = m$. Thus, $\mathbb{1}_m^{\alpha_1}.\mathbb{1}_m^{\alpha_1,\alpha_2} = m\mathbb{1}_m^{\alpha_2}$ and iteratively $\mathbb{1}_m^{\alpha_1}.\mathbb{1}_m^{\alpha_1,\alpha_2}...\mathbb{1}_m^{\alpha_{N-2},\alpha_{N-1}}.\mathbb{1}_m^{\alpha_{N-1}} = m^{N-1}$.

Consequently,

$$\forall l \in \mathcal{L}, \quad f^l(X) = H^N m^{N-1} M$$

with $M = \prod_{i=1}^{N} (\cos(\frac{\pi}{2} x^i) + \sin(\frac{\pi}{2} x^i))$.

It's now possible to control the decision function according to the constant H and the constant M which depends on the inputs. The algorithm behaves better with smaller values of the decision function at the beginning of the execution, so the goal is to find an upper bound of the decision function. Thus, it has been chosen to take the largest M computed on the data. Then, it is possible to fix a goal called f^l_{max} and to find the corresponding value of H so that the decision function will be equal to the goal for the the largest M and lower than the goal for other inputs. The value of H can be found as follows :

$$\forall l \in \mathcal{L}, f^l(X_{max}) = H^N m^{N-1} M_{max} = f^l_{max} \iff H = \frac{{f^l_{max}}^{\frac{1}{N}}}{m^{1-\frac{1}{N}} M_{max}^{\frac{1}{N}}} \tag{2.1}$$

In order to observe the new results of the initialization, we made an experience. We took 1000 inputs from the MNIST database and we used them in order to initialize the tensor W with different values for the goal. Finally, we computed the value of the decision function for each input. A representation of the experience is presented on the graph 2.1 .

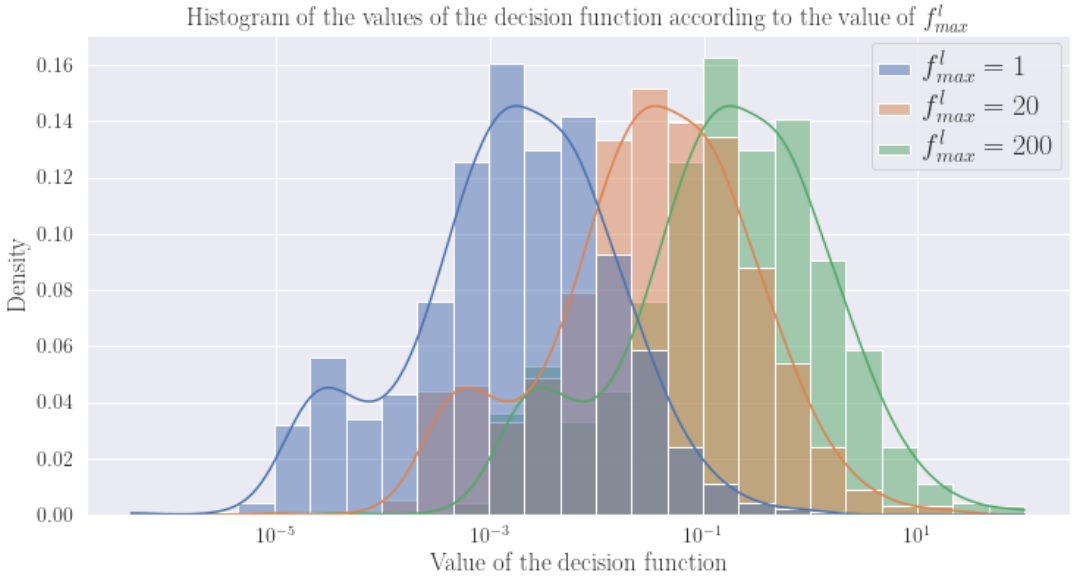

Figure 2.1: Results of the new initialization

It's interesting to see that it's possible to adapt the mean of the values of the decision function according to f^l_{max}. The value f^l_{max} fix the maximum value accepted. In the histogram, f^l_{max} equal to 1 is a little bit too small. Then, with f^l_{max} equal to 20, values are centered on 0.2 and the maximum value is not too big. It's the best choice here. Finally, with f^l_{max} equal to 200, values are centered around 2 but the maximal value is equal to 200 so it can cause exploding gradients issues.

In this section, we have proposed a new to initialize the factors of W. In the rest of the report, this new initialization will be used.

2.2 Loss functions

In the first chapter, it has been shown that the gradient was very unstable. The loss can explode in only one gradient descent step or it can get stuck for a lot of time.

That's why, it can be interesting to use other kind of loss functions in order to overcome issues as the computation of the gradients will not be the same. We decided to test two of them :

- the cross entropy loss function. Indeed, the task of the algorithm is to classify inputs and not perform a regression. Thus, a natural choice is to measure loss using the cross entropy which is used extensively for classification. Cross entropy is defined by :

$$C^{entropy}(f) = -\frac{1}{N_t}\sum_{n=1}^{N_T}\sum_{l} y_n^l \log(p_n^l), \quad \text{with } p_n^l = \frac{e^{f^l(X_n)}}{\sum_i e^{f^i(X_n)}} \tag{2.2}$$

- the log quadratic loss function. This loss function adds a log function around the quadratic loss $(f^l(X_n) - y_n^l)^2$ in order to avoid exploding gradients problem when going to the opposite direction of the sweeps (DMRG-OD). However, this loss function is ineficient with DMRG-SD because the gradient is smaller than with the quadratic loss function. This loss function is compared with the quadratic loss on the graphic 2.2. Log quadratic is defined by :

$$C^{log}(f) = \frac{1}{2N_t}\sum_{n=1}^{N_T}\sum_{l} \log((f^l(X_n) - y_n^l)^2 + b), \quad \text{with b a constant} \tag{2.3}$$

The constant b is equal to 1 in the rest of the report.

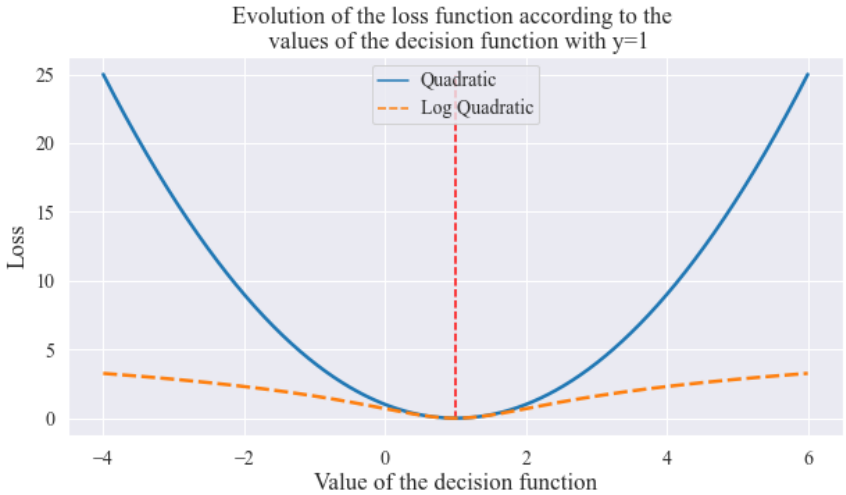

Figure 2.2: Difference between the log quadratic and quadratic loss function

The goal is now to compute the gradient with respect to B for the 2 new loss function. Firstly, it was shown in the chapter 1 and on the picture 1.4 that the decision function can be written as :

$$\forall l \in \mathcal{L}, f^l(x) = W^l.\Phi(x) = \begin{cases} B.\tilde{\Phi}^l(X_n) & \text{if } l \notin B \\ B^l.\tilde{\Phi}(X_n) & \text{if } l \in B \end{cases} \tag{2.4}$$

Then, using the chain rule, it is possible for any cost function C to write :

$$\frac{\partial C(f)}{\partial B} = \frac{\partial C(f)}{\partial f}\frac{\partial f}{\partial B}$$

Given that the decision function is just the contraction between B and $\tilde{\Phi}$ as shown on equations 2.4, $\frac{\partial f}{\partial B} = \tilde{\Phi}$.

Thus, according to the position of the index label l, the gradient for any cost function C is equal to :

$$\frac{\partial C(f)}{\partial B} = \begin{cases} \frac{\partial C(f)}{\partial f}.\tilde{\Phi}^l(X_n) & \text{if } l \notin B \\ \frac{\partial C(f)}{\partial f} \otimes \tilde{\Phi}(X_n) & \text{if } l \in B \end{cases} \tag{2.5}$$

with $\frac{\partial C(f)}{\partial f}$ a vector of dimension N_l.

Now, it's necessary to compute the term $\frac{\partial C(f)}{\partial f}$ for the 2 new loss functions.

Log quadratic

$$\begin{aligned} \forall l \in \mathcal{L}, \quad \frac{\partial C^{log}(f)}{\partial f^l} &= \frac{1}{2N_t}\sum_{n=1}^{N_T} \frac{\partial \log((f^l(X_n) - y_n^l)^2 + b)}{\partial f^l} \\ &= \frac{1}{2N_t}\sum_{n=1}^{N_T} \frac{\partial \log((f^l(X_n) - y_n^l)^2 + b)}{\partial (f^l(X_n) - y_n^l)^2 + b)} \frac{\partial ((f^l(X_n) - y_n^l)^2 + b)}{\partial f^l} \\ &= \frac{1}{2N_t}\sum_{n=1}^{N_T} \frac{1}{((f^l(X_n) - y_n^l)^2 + b)} 2(f^l(X_n) - y_n^l) \\ &= \frac{1}{N_t}\sum_{n=1}^{N_T} \frac{f^l(X_n) - y_n^l}{(f^l(X_n) - y_n^l)^2 + b} \end{aligned}$$

Therefore,

$$\frac{\partial C^{log}(f)}{\partial B} = \begin{pmatrix} \frac{1}{N_t}\sum_{n=1}^{N_T} \frac{f^1(X_n) - y_n^1}{(f^1(X_n) - y_n^1)^2 + b} \\ \cdots \\ \frac{1}{N_t}\sum_{n=1}^{N_T} \frac{f^{N_l}(X_n) - y_n^{N_l}}{(f^{N_l}(X_n) - y_n^{N_l})^2 + b} \end{pmatrix} \begin{matrix} .\tilde{\Phi}^l(X_n) & \text{if } l \notin B \\ \otimes \tilde{\Phi}(X_n) & \text{if } l \in B \end{matrix} \tag{2.6}$$

Cross entropy

Let's $\sum_i e^{f^i(X_n)} = \mathcal{S}_n$ and $f_n^l = f^l(X_n) \quad \forall n$. Thus, $p_n^l = \frac{e^{f^l(X_n)}}{\mathcal{S}_n}$ and :

$$\frac{\partial p_n^i}{\partial f_n^j} = \begin{cases} \partial \frac{e^{f_n^i}}{\mathcal{S}_n} = \frac{e^{f_n^i}\mathcal{S}_n - e^{f_n^i}e^{f_n^i}}{\mathcal{S}_n^2} = \frac{e^{f_n^i}}{\mathcal{S}_n}[\frac{\mathcal{S}_n - e^{f_n^i}}{\mathcal{S}_n}] = p_n^i(1-p_n^i) & \text{if } i = j \\ \partial \frac{e^{f_n^i}}{\mathcal{S}_n} = -\frac{e^{f_n^i}e^{f_n^j}}{\mathcal{S}_n^2} = -p_n^i p_n^j & \text{if } i \neq j \end{cases}$$

Consequently,

$$\begin{aligned} \forall j \in \mathcal{L}, \quad \frac{\partial C^{entropy}(f)}{\partial f^j} &= -\frac{1}{N_t}\sum_{n=1}^{N_T}\sum_l y_n^l \frac{\partial \log(p_n^l)}{\partial f_n^j} \\ &= -\frac{1}{N_t}\sum_{n=1}^{N_T}\sum_l y_n^l \frac{\partial \log(p_n^l)}{\partial p_n^l}\frac{\partial p_n^l}{f_n^j} \\ &= \frac{1}{N_t}\sum_{n=1}^{N_T}[-\sum_{l\neq j} y_n^l \frac{(-\cancel{p_n^l} p_n^j)}{\cancel{p_n^l}} - y_n^j \frac{\cancel{p_n^j}(1-p_n^j)}{\cancel{p_n^j}}] \\ &= \frac{1}{N_t}\sum_{n=1}^{N_T}[-y_n^j + p_n^j \sum_l y_n^l] \\ &= \frac{1}{N_t}\sum_{n=1}^{N_T}[p_n^j - y_n^j] \quad \text{as } \sum_l y_n^l = 1 \end{aligned}$$

Whence,

$$\frac{\partial C^{entropy}(f)}{\partial B} = \begin{pmatrix} \frac{1}{N_t}\sum_{n=1}^{N_T}[p_n^1 - y_n^1] \\ \dots \\ \frac{1}{N_t}\sum_{n=1}^{N_T}[p_n^{N_l} - y_n^{N_l}] \end{pmatrix} \begin{matrix} .\tilde{\Phi}^l(X_n) & \text{if } l \notin B \\ \otimes \tilde{\Phi}(X_n) & \text{if } l \in B \end{matrix} \tag{2.7}$$

The gradient for the 2 new loss functions is now defined, thus it is possible to use the DMRG algorithm with different loss functions. In order to compare the different loss functions, below we propose an experiment to compare the proposed loss functions. We computed the accuracy along the execution for the sweeps loss functions and algorithms (DMRG-SD and DMRG-OD) on the same example with N=100. The results are presented on the graphic 2.3. The quadratic loss function with DMRG-OD was not used as it has been shown that is a very unstable algorithm on the graphic 1.6. Moreover, we didn't try with DMRG-SD and the log quadratic as it will not solve the problem of vanishing gradients.

It's shown on the graphic that the cross entropy with DMRG-SD has better results than the others. However, from the second sweeps, the accuracy is stuck for the loss functions with DMRG-SD. Moreover, the accuracies with DMRG-OD don't move so much. Indeed, it is necessary to put a low training rate otherwise the loss explodes.

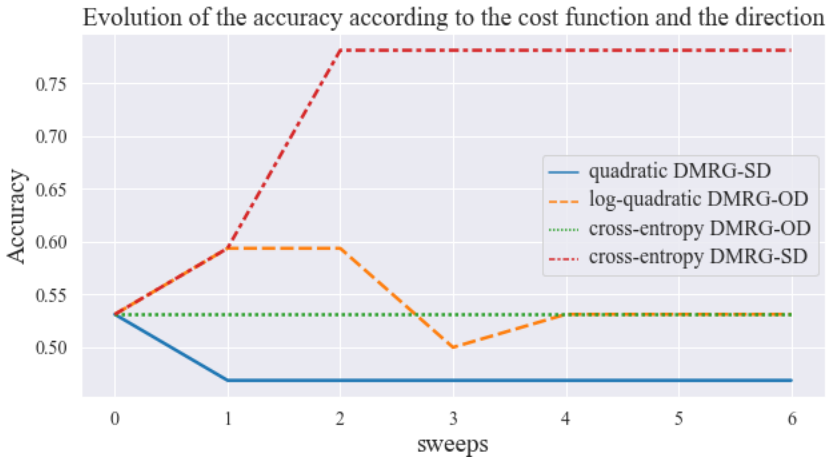

Figure 2.3: Example of execution with different loss functions and algorithms

Consequently, the cross entropy loss function gives better results but the log-quadratic doesn't make improvement. However, there are still problems of vanishing or exploding gradients. To deal with these issues, the next section introduces new algorithms of optimization.

2.3 Optimization Methods

The most important problem of the gradient descent with fixed step size in our context is the learning rate. If the step size is fixed, it is difficult to train the different factors. In this section, a bond refers to the tensor B computed by the contraction of two neighbouring factors A.

The objective of the algorithm is to minimize the cost function at each bond (each B) by modifying only the tensor B while the other tensors are fixed. It's a similar approach to the Block-Coordinate descent.

To minimize the cost, the DMRG algorithm performs only one gradient descent step for each B. However, as it is shown on the equation 1.6, with the quadratic loss function, the minimization problem at each bond is quadratic so it is possible to solve it easily with a certain precision. Therefore, it is possible to perform numerous gradient descent steps at each bond in order to solve the problem. With the other loss functions, the problem is not quadratic, but it is possible to make the same operations. However, doing too much steps is inefficient because after several passes the gradient and the decrease of the loss are small while the computation time is the same. The maximal number of passes is denoted by Npass. The method is named NGD and is depicted below:

Method 1: NGD

Compute $\frac{\partial C(f)}{\partial B}$; $k = 0$
while $||\frac{\partial C(f)}{\partial B}|| > \epsilon$ *and* $k < Npass$ **do**
  $B = B - \alpha \frac{\partial C(f)}{\partial B}$
  Compute $\frac{\partial C(f)}{\partial B}$
  $k = k + 1$
end

On the graphic 2.4 below, we made an experience with DMRG-SD, NGD and the quadratic loss function with different Npass to compare the loss versus the time of the execution.

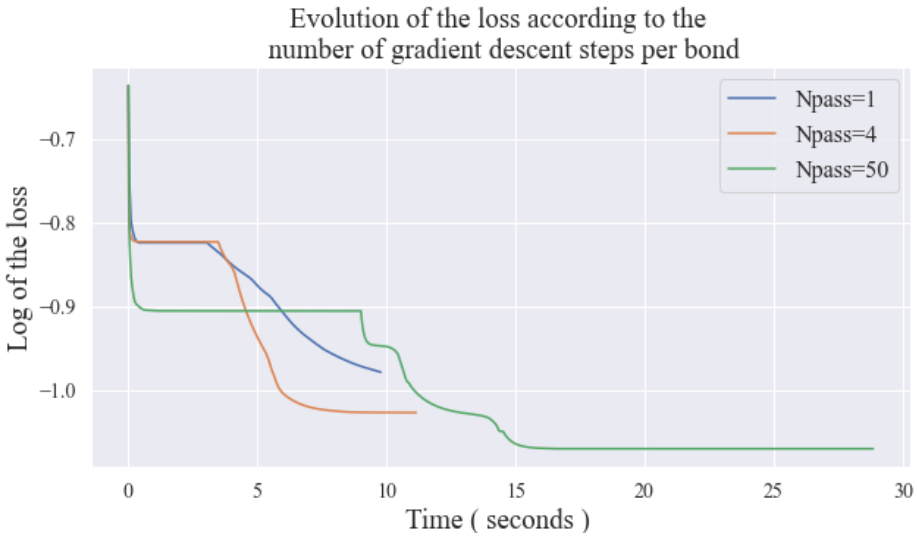

Figure 2.4: Example on Npass

Figure 2.4 shows that doing several gradient descent steps helps to minimize the cost function : the loss decreases faster with Npass equal 4 than with Npass equal 1. However, if the number of passes is too big, the algorithm can take a lot of time and can also get stuck. Indeed, on the graphic, the loss for Npass equal 50 decreases very rapidly at the beginning but after it gets stuck and it takes time so that the loss decreases again. In the rest of the report, method 1 will be used with Npass equal 4 and the algorithm will be denoted DMRG-(SD/OD)-NGD.

This method is a good improvement but that will not solve the main problems, it will just speed up the learning a little. In order to cope with the problems, it's important to use another optimization method than just a simple gradient descent with fixed step size. We decided to test two optimization methods :

- Adam, Adaptative Moment Estimation
- Conjugate Gradient Descent

Adam The first one is a well-known optimization method in the world of neural net and Deep Learning. Adam is a stochastic optimization algorithm that only requires first-order gradients with little memory requirement. It has the advantage to work well with sparse gradients. In the report, we are not going to explain how it works because all is well described in the publication [8] but it has been implemented and the algorithm DMRG with Adam is called DMRG-(SD/OD)-Adam.

In the construction of DMRG, each time the algorithm goes to the next bond, it has to contract the factor obtained after performing the SVD of B which will not be updated next with other tensors in order to compute $\tilde{\Phi}$ for the computation of the gradient so that the algorithm scales linearly with N. However, this operation forces DMRG to compute the gradient of B on all the data at each bond. That's why, we use Adam to realize several passes at each bond by using each time all the data, and the method is not stochastic.

Conjugate Gradient Descent The second method is very different and it's not a stochastic optimization method. This optimization method only works with quadratic problem. In our case, we can use this algorithm only with the quadratic loss function because as it is written on the equation 1.6 the problem at each bond can be written as

$$\min_B \frac{1}{2N_t} \sum_{n=1}^{N_T} \sum_{l=1}^{N_l} (B.\tilde{\Phi}(X_n) - y_n^l)^2 \tag{2.8}$$

and the problem is quadratic according to B.

As the problem is quadratic, to find the optimal value of B, it's sufficient to cancel out the gradient of the function to minimize. Thus,

$$\text{B optimal} \iff \frac{\partial C(B)}{\partial B} = \begin{cases} \frac{1}{N_t} \sum_{n=1}^{N_T} (B.\tilde{\Phi}^l(X_n) - y_n^l).\tilde{\Phi}^l(X_n) = 0 & \text{if } l \notin B \\ \frac{1}{N_t} \sum_{n=1}^{N_T} (B^l.\tilde{\Phi}(X_n) - y_n^l) \otimes \tilde{\Phi}(X_n) = 0 & \text{if } l \in B \end{cases}$$

$$\iff \begin{cases} \sum_{n=1}^{N_T} B.\tilde{\Phi}^l(X_n).\tilde{\Phi}^l(X_n) = \sum_{n=1}^{N_T} y_n^l.\tilde{\Phi}^l(X_n) & \text{if } l \notin B \\ \sum_{n=1}^{N_T} B^l.\tilde{\Phi}(X_n) \otimes \tilde{\Phi}(X_n) = \sum_{n=1}^{N_T} y_n^l \otimes \tilde{\Phi}(X_n) & \text{if } l \in B \end{cases}$$

$$\iff \begin{cases} B.\sum_{n=1}^{N_T} A_n = \sum_{n=1}^{N_T} b_n & \text{if } l \notin B \text{ with } A_n = \tilde{\Phi}^l(X_n)\tilde{\Phi}^l(X_n)^T, b_n = y_n^l.\tilde{\Phi}^l(X_n) \\ B^l.\sum_{n=1}^{N_T} A_n = \sum_{n=1}^{N_T} b_n & \text{if } l \in B \text{ with } A_n = \tilde{\Phi}^l(X_n) \otimes \tilde{\Phi}^l(X_n), b_n = y_n^l \otimes \tilde{\Phi}^l(X_n) \end{cases}$$

The notation $\tilde{\Phi}^l(X_n)\tilde{\Phi}^l(X_n)^T$ can be ambiguous but it's linked with the form of the tensor $\tilde{\Phi}$. Indeed, when l $\notin$ B, there are 3 different cases :

- the first tensor is in B, so $\tilde{\Phi} = \phi^{s_1} \otimes \phi^{s_2} \otimes \tilde{\Phi}_1^{\alpha_2,l}$. All the elements are matrices or vectors thus $\tilde{\Phi}^l(X_n)\tilde{\Phi}^l(X_n)^T = \phi^{s_1}{\phi^{s_1}}^T \otimes \phi^{s_2}{\phi^{s_2}}^T \otimes \tilde{\Phi}_1^{\alpha_2,l}\tilde{\Phi}_1^{\alpha_2,l^T}$.
- the last tensor is in B, so $\tilde{\Phi} = \phi^{s_{N-1}} \otimes \phi^{s_N} \otimes \tilde{\Phi}_1^{\alpha_{N-1},l}$. All the elements are matrices or vectors thus $\tilde{\Phi}^l(X_n)\tilde{\Phi}^l(X_n)^T = \phi^{s_{N-1}}{\phi^{s_{N-1}}}^T \otimes \phi^{s_N}{\phi^{s_N}}^T \otimes \tilde{\Phi}_1^{\alpha_{N-1},l}\tilde{\Phi}_1^{\alpha_{N-1},l^T}$.
- neither the last tensor nor the first is in B, so $\tilde{\Phi} = \phi^{s_{i-1}} \otimes \phi^{s_i} \otimes \tilde{\Phi}_1^{\alpha_{i-2},l} \otimes \tilde{\Phi}_2^{\alpha_i}$. The label index can be in $\tilde{\Phi}_1$ or $\tilde{\Phi}_2$ according to the position of B compared with the tensor with the label index. Thus $\tilde{\Phi}^l(X_n)\tilde{\Phi}^l(X_n)^T = \phi^{s_{i-1}}{\phi^{s_{i-1}}}^T \otimes \phi^{s_i}{\phi^{s_i}}^T \otimes \tilde{\Phi}_1^{\alpha_{i-1},l}\tilde{\Phi}_1^{\alpha_{i-1},l^T} \otimes \tilde{\Phi}_1^{\alpha_i}\tilde{\Phi}_1^{\alpha_i{}^T}$.

Finally, let $A = \sum_{n=1}^{N_T} A_n$ and $b = \sum_{n=1}^{N_T} b_n$. Thus, B is optimal if and only if it verifies the following equation :

$$A.B = b \tag{2.9}$$

We didn't put the index label but in the case l $\in$ B, B contains the index label l.

Let's $P = \{p_1, \ldots, p_K\}$ a set of K mutually conjugate vectors with respect to A, that is for each pair (u, v) of P, u.A.v = 0. Then P forms a basis for the space where B lives and the goal of the conjugate gradient method is to express the tensor B in this basis in order to find K conjugate directions and so to make descent gradient steps in these directions. The method never uses the same direction two times. It's the principal advantage of the method. On the following picture 2.5 , a comparison between the conjugate gradient method (green) and gradient descent with optimal step size (red) is depicted.

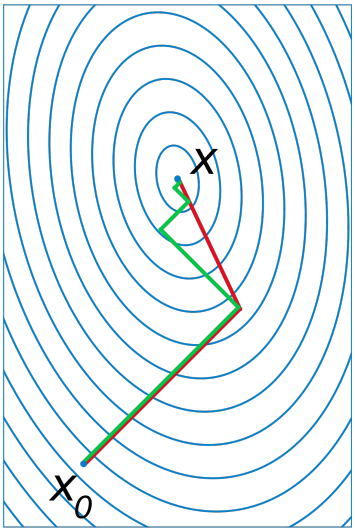

Figure 2.5: Comparison between the conjugate gradient method (red) and gradient descent with optimal step size (green) (`https://en.wikipedia.org/wiki/Conjugate_gradient_method`)

Interestingly, the fact that the conjugate gradient method must use conjugate directions helps it to converge faster than gradient descent with optimal step size.

However, it's impossible to find all the directions, it would take too much time. That's why, the conjugate gradient method can be seen as an iterative method by searching a new direction until the method finds a solution near the optimum. The method needs to find the best direction at each iteration.

Let's define the kth iteration. Suppose that the method has already found k-1 conjugate directions before. Let r_k be the residual at the kth iteration :

$$r_k = b - A.B_k$$

The tensor r_k is also equal to the negative gradient of the loss function according to B so the gradient descent would require to move in the direction r_k. However, the new direction needs to be conjugate with all the precedent directions. To respect the 2 precedent conditions, the new direction can be build on the residual r_k and the residual can become conjugate with the other directions by using the Gram-Schmidt-orthonormalization as the conjugation constraint is an orthonormal-type constraint. Thus, it gives us the following expression which defined the new direction p_k :

$$p_k = r_k - \sum_{i<k} \frac{p_i.A.r_k}{p_i.A.p_i} p_i$$

Next, the tensor B_{k+1} is defined by $B_{k+1} = B_k + \alpha_k p_k$ and it's necessary to find the optimal value α_k for the gradient descent step. Firstly, the quadratic loss function can be defined as $\frac{1}{2}B.A.B - B.b$ with the notation of A and b defined before. To find the optimal value of α_k, it's necessary to search the smallest value of the quadratic loss function at the point B_{k+1} according to α_k so we have to develop the expression :

$$\begin{aligned} C(B_{k+1}) &= \frac{1}{2}B_{k+1}.A.B_{k+1} - B_{k+1}.b \\ &= \frac{1}{2}(B_k + \alpha_k p_k).A.(B_k + \alpha_k p_k) - (B_k + \alpha_k p_k).b \\ &= \alpha_k p_k.A.B_k + \frac{1}{2}\alpha_k^2 p_k.A.p_k - \alpha_k p_k.b + \frac{1}{2}B_k.A.B_k - B_k.b \end{aligned}$$

To find the smallest value of the above expression, we must cancel out the derivative of the expression according to α_k (single variable) as the expression is quadratic and convex :

$$\begin{aligned} C'(\alpha_k) = 0 &\iff p_k.A.B_k + \alpha_k p_k.A.p_k - p_k.b = 0 \\ &\iff \alpha_k = \frac{p_k.(b - A.B_k)}{p_k.A.p_k} \end{aligned}$$

Therefore, it's now possible to make a gradient descent step $B_{k+1} = B_k + \alpha_k p_k$ with $\alpha_k = \frac{p_k.r_k}{p_k.A.p_k}$ and to go to the next iteration.

These are the explanations of the conjugate gradient method but it requires a lot of computation with for example the computation of r_k and the utilization of Gram-Schmidt-orthonormalization and storage. A closer analysis of the method can suppress a lot of steps and gives the algorithm bellow with less computation and storage :

Method 2: Conjugate Gradient Method

$r_0 = b - A.B_0$; $p_0 = r_0$; $k = 0$
while $||r_k|| > \epsilon$ *and* $k < Npass$ **do**
  $\alpha_k = \frac{r_k.r_k}{p_k.A.p_k}$
  $B_{k+1} = B_k + \alpha_k p_k$
  $r_{k+1} = r_k - \alpha_k A.p_k$
  if $||r_{k+1}|| > \epsilon$ **then**
    $\beta_k = \frac{r_{k+1}.r_{k+1}}{r_k.r_k}$
    $p_{k+1} = r_{k+1} + \beta_k p_k$
    $k = k + 1$
end

As it's defined in the algorithm, the first direction is simply the negative of the gradient so like a basic gradient descent. DMRG algorithm with the conjugate gradient method is called DMRG-(SD/OD)-CG in the rest of the report.

To sum up, this section introduces two new optimization methods : Adam and Conjugate Gradient Method. To compare them on the same example, we must use the quadratic loss function. A comparison of the 3 optimization methods with the quadratic loss function , Npass=4 and N=100 is presented on the graph 2.6 below :

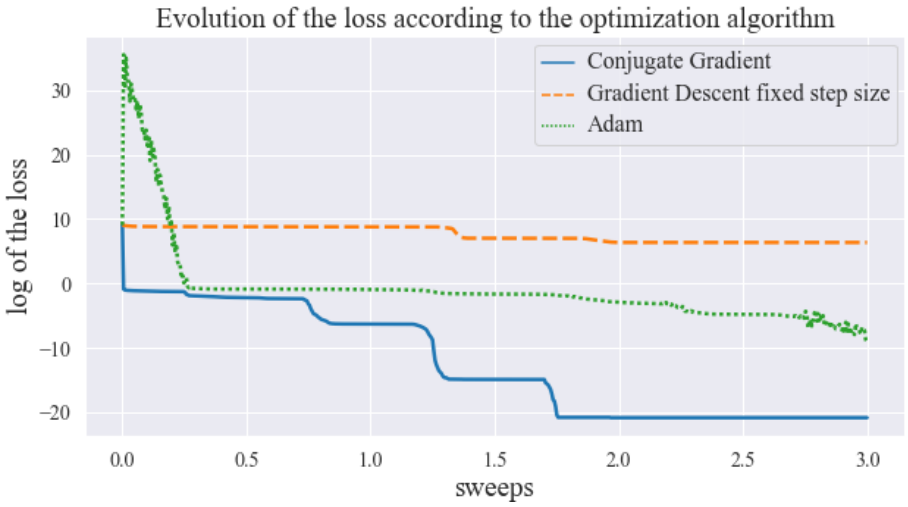

Figure 2.6: Comparison between the optimization algorithms with DMRG-SD

The results are very promising. After only 2 sweeps, the loss with DMRG-SD-GD method reaches 10^{-10}, it's better than DMRG-SD-NGD. Moreover, the precision ϵ used in the methods was fixed at 10^{-10}, that's why the loss doesn't decrease after the second sweeps. RegardingDMRG-SD-Adam, the results are encouraging, better than DMRG-SD-NGD. Figure 2.6 shows that for the first bonds of the first sweep, the learning rate was too large but it allowed to speed up the learning. Given the results, in the rest of the report, we will only use DMRG-(SD/OD)-CG with the quadratic loss function.

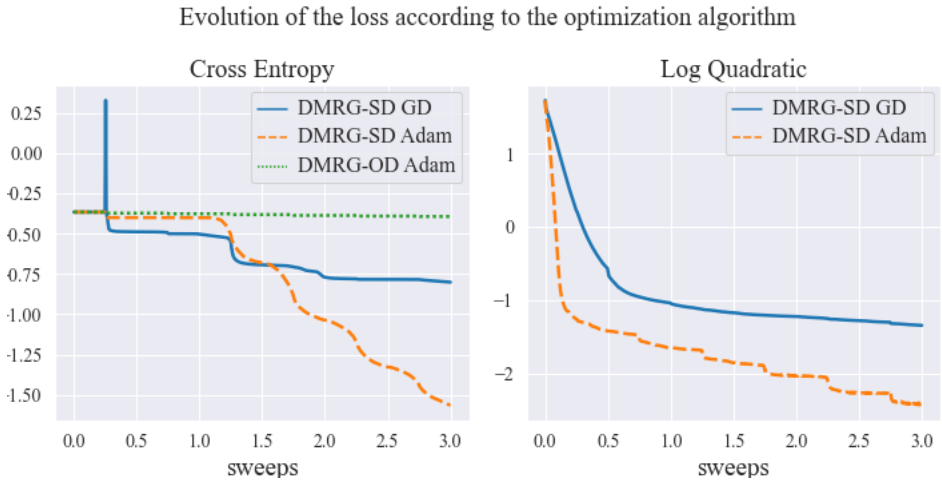

Figure 2.7: Comparison between the optimization algorithms for cross entropy and log quadratic

In addition, a comparison of the Adam and GD with the log quadratic and cross entropy loss function , Npass=4 and N=100 is presented on the graph 2.7 above . Firstly, the DMRG-OD-Adam with the cross-entropy doesn't work well so it can be put aside. Moreover, results are better with Adam than for GD with the two loss functions. Thus, in the rest of the report, we will use the two loss functions only with DMRG-(SD/OD)-Adam.

To finish this second chapter on the new implementations, the graph 2.8 below makes a recap of the different algorithms on an example randomly generated with Npass=4 et N=100.

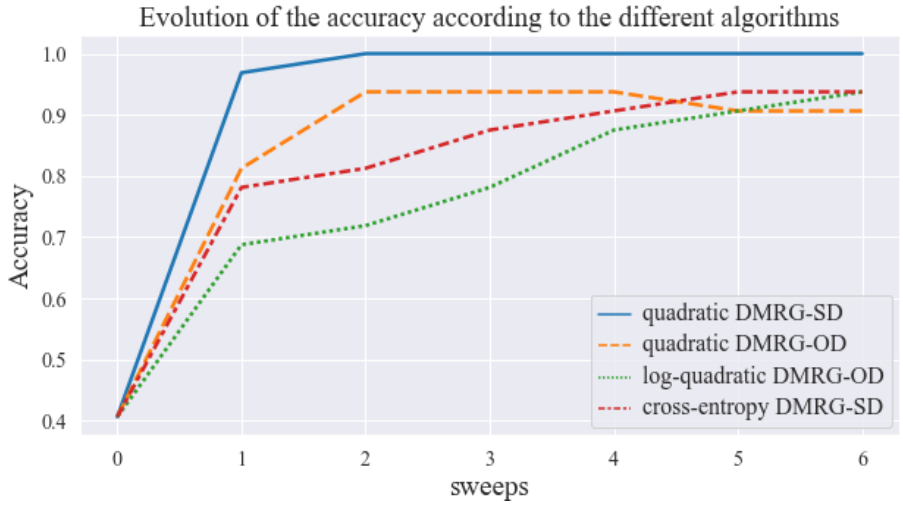

Figure 2.8: Comparison of the different algorithms

Results are better than on the graph 2.3 whereas it was the same example. All the accuracies are near 1 and DMRG-SD-CG with the quadratic loss function is very promising. Indeed, in only two sweeps, it reaches 100% accuracy.

The use of more advanced optimization methods helps a lot the training of the tensor network. In the next chapter, we are going to test the different algorithms on the MNIST Dataset.

Chapter 3

Practical Applications on MNIST

This last chapter deals with the test of all the algorithms on the MNIST Database [1].

The MNIST database is a base of pictures of handwritten digits. This database is easy to use because all the pictures have been size-normalized and centered in a fixed-size image. The database has a training set of 60,000 examples and a test set of 10,000 examples with pictures of 28 by 28 pixels.

Two example samples taken from the training set are shown on the figure below.

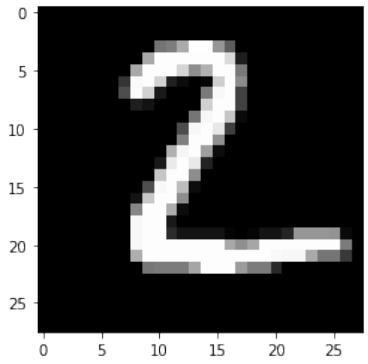

Figure 3.1: Picture of an handwritten 2

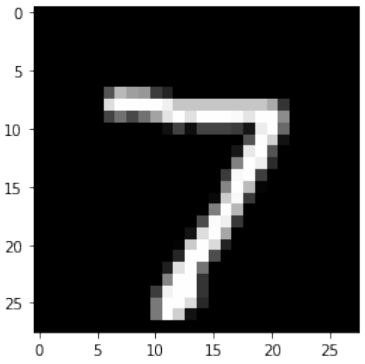

Figure 3.2: Picture of an handwritten 7

It has been shown in the first report that it was difficult to train the tensor network on the initial pictures because the time for learning was big. That's why, we decided to work with smaller images, the computation time become lower but the information is also smaller. It is a tradeoff and in the case of the MNIST Database, it's clearly possible to recognize the digits with smaller images. As in the publication [2], each images was scaled down from 28×28 to 14×14 by averaging clusters of fours pixels.

3.1 Comparison of the algorithms

Firstly, in this section, DMRG-(SD/OD)-CG with the quadratic loss, DMRG-OD-Adam with log quadratic loss and DMRG-SD-Adam with the cross entropy are going to be compared on a small number of images. Indeed, the computation time is large so it's easier to compare them on a small parts of the pictures. The number of gradient steps, the threshold for the gradient and the learning rate were tuned for all the algorithms.

Figure 3.3 below shows the results of the experiment on 100 pictures and a maximal bond dimension of 10.

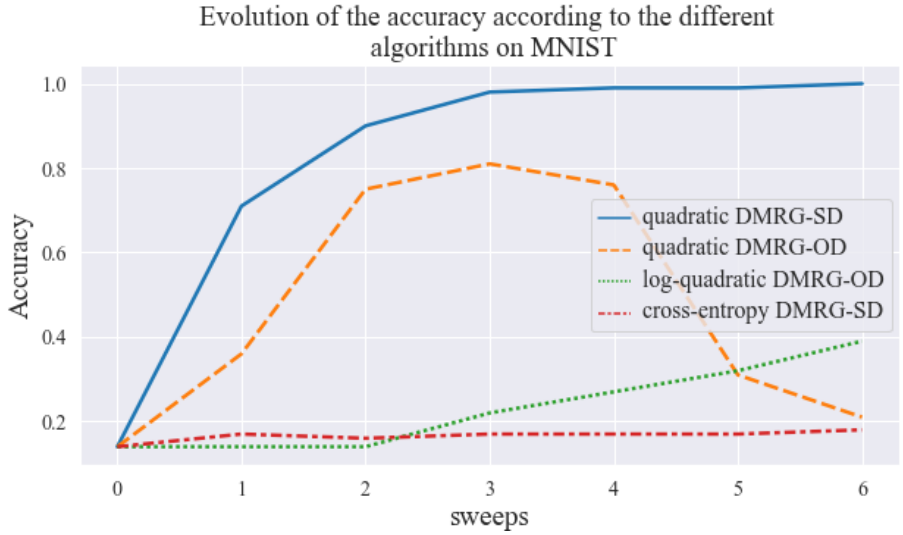

Figure 3.3: Comparison of the different algorithms on MNIST

Results are good, DMRG-SD-CG with the quadratic loss reaches 100 % of correct classification in about 5 sweeps. We reached our goal. For other algorithms, results are less good :

- quadratic DMRG-OD-CG : in the first sweeps the accuracy grow up rapidly until 80 % but from the third sweeps, the accuracy fall to 20 %. Maybe, it's because the number of passes of gradient descent step is too big, and during the SVD a lot of proper values need to be cut and so the error increases.
- log-quadratic DMRG-OD-Adam : at the beginning, it's necessary to put a small learning rate otherwise the error explodes so the learning is slow but the graph shows that at the end the accuracy begins to increase faster. However, it's very unstable and it can fall like the previous item.
- cross-entropy DMRG-SD-Adam : as for the previous item, the learning rate needs to be small so the learning is too small. Maybe, it can be a good idea to change the learning rate after the first sweep.

Given the results, in the next section, we will only use the quadratic loss function with DMRG-SD.

3.2 Comparison according to the maximal bond dimension

In the previous section, the accuracy was only computed on the same data as training so the accuracy was overestimated (overfitting). Moreover, the maximal bond dimension was fixed and was equal to 10.

Therefore, in this section, the accuracy on a test set and a training set is going to be computed according to different values of the maximal bond dimension. However, the computation time is very large so it is difficult to make the experiment on the whole dataset.Thus, both the training set and testing set will contain 1000 pictures.

Results are presented on the graph 3.4 below. We had to stop the training after 7 sweeps for all the cases otherwise the computation time was too long.

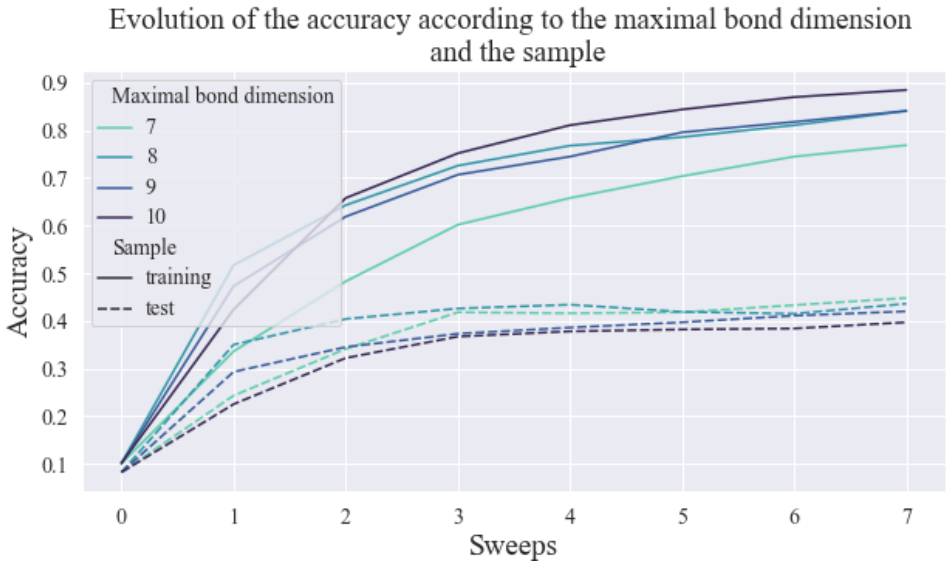

Figure 3.4: Comparison of accuracies according to the maximal bond dimension

The graph shows that if the maximal bond dimension increases, the accuracy on the training set increases too. This comes from the fact the larger the bond dimension, the more parameters there are and thus the bias decreases. Here, the algorithm with a maximal bond dimension reaches 90% accuracy. In addition, it would have continue to increase if the training had been longer.

However, for the test set, it's the opposite, big models with high bond dimension have the worst accuracy and the accuracies are low for all the models. This is explained by the fact that the number of data is small while there are a lot of parameters, far too much compared to the number of training samples. Models are overfitting, there is too much variance. In order to solve this problem, it would be necessary either to add data or to add a term of regularization.

To conclude, the algorithm DMRG-SD-CG with the quadratic loss function is now able to classify pictures from the MNIST Dataset with very good percentage of accuracy. It would be interesting to train the model with all the training set and compare the accuracy on the training and test set with different bond dimensions.

Conclusion

To conclude, in the last report, we had not reach our goal, the training of the tensor network was inefficient. Thus, this time we tried to make some improvements on the algorithms.

Firstly, we tried a new way to initialize the tensors of W in order to avoid the explosion of the loss. Secondly, we tested other losses than the quadratic loss : the log quadratic and the cross-entropy loss. Then, we added to the project other optimization algorithms : Adam and the Conjugate Gradient Method.

Finally, we were able to test all the new functionalities on the training of a tensor network in order to classify the MNIST Database. We saw that DMRG-SD-CG with the quadratic loss and the conjugate gradient method is very efficient for training, thus the goal is reached. However, some functionalities we added as the cross-entropy loss are not interesting for the training yet. To finish, we saw that the maximal bond dimension was a very important parameter for the bias-variance trade-off. However, the computation time is typically quite large, which limits the applicability of the proposed method on large scale data.

It would be interesting in a next part to test the model on the all the training set in order to know the real accuracy. Moreover, it would be interesting to reduce the computation time. Research perspectives can be the parallelization of the computation on inputs but also try other models and algorithms as what is described in the publications [9] and [10].

Bibliography

[1] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[2] E. M. Stoudenmire and D. J. Schwab, "Supervised learning with quantum-inspired tensor networks," 2017.

[3] V. Bertret, "Tensor networks and kernels for supervised machine learning," 2020. [Online]. Available: https://github.com/victorius35/PIR-Tensor-Network-MNIST/blob/main/reports/Report-1.pdf

[4] C. R. Harris, K. J. Millman, S. J. der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[5] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic, "Tensorly: Tensor learning in python," 2018.

[6] V. N. Vapnik, *The Nature of Statistical Learning Theory*, 2nd ed. Springer, November 1999.

[7] W. Waegeman, T. Pahikkala, A. Airola, T. Salakoski, M. Stock, and B. De Baets, "A kernel-based framework for learning graded relations from data," *IEEE Transactions on Fuzzy Systems*, vol. 20, no. 6, p. 1090–1101, Dec 2012. [Online]. Available: http://dx.doi.org/10.1109/TFUZZ.2012.2194151

[8] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[9] S. Efthymiou, J. Hidary, and S. Leichenauer, "Tensornetwork for machine learning," 2019.

[10] I. Glasser, N. Pancotti, and J. I. Cirac, "From probabilistic graphical models to generalized tensor networks for supervised learning," 2019.