UNIVERSITY OF BUCHAREST

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Line search methods for first and second order optimization

Master Thesis

Supervisor:

Assoc. Prof. PhD Bogdan ALEXE

Author:

Ciprian-Mihai CEAUȘESCU

Bucharest

2019

Abstract

In recent years of research, *Deep Learning* has achieved considerable progress in many fields of study, such as *Natural Language Processing* and *Computer Vision*. In comparison with the traditional *Machine Learning* algorithms, using deep learning, the built models have a strong capability of learning the structure and the pattern of the data. Applying different techniques to build accurate deep neural networks, the main goal of the researchers is to understand how the data is represented. This task is difficult, because the data have multiple layers of abstractions and the models are represented by complicated functions with millions of parameters.

During the training process of a deep neural network, the purpose is to minimize the *loss function*, which measures how accurate is the created model with respect to the dataset. *Hyperparameters* define the ability of a neural network to learn the structure of the data and they have an important impact on it's performance. *Step size*, also known as *learning rate*, is a hyperparameter that controls how much the parameters of a model are adjusted with respect to the loss function, in order to improve it's prediction capability. Determining an ideal value of the step size will assure that the optimization process is done in a reasonable time and the loss function will converge to the global minimum.

In this thesis we present different *line search* algorithms, used to compute an optimal value for learning rate parameter, during the training of deep neural networks. We want to study the convergence rate of the model for both *convex* and *non-convex* problems using *backtracking*, *Goldstein* and *Weak Wolfe* methods.

In the first part of this thesis, we introduce the concept of neural networks and we describe the optimization process using *first order* (first derivative of the loss function) and *second order* (second derivative of the loss function) methods. After this introductory part, we summarize the idea of line search methods and we illustrate the differences between the methods. Finally, we briefly describe the implementation of the architectures for convex and non-convex problems, presenting the experimental results of the line search methods.

We conclude that, using line search methods, the models provide a better performance, but we need to make sure that they are able to *generalize*.

Abstract

În decursul ultimilor ani, metodele *Deep Learning* au realizat progrese considerabile în numeroase domenii de studiu precum *Procesarea Limbajului Natural* și *Vederea Artificială*. Folosind deep learning, modelele construite sunt capabile să înțeleagă mult mai precis structura și forma datelor. Printre obiectivele cercetătorilor, se numără și acela de a înțelege cum sunt reprezentate datele. Această sarcină este una dificilă, întrucât informația poate proveni din distribuții abstracte. De asemenea, modelele sunt reprezentate de funcții variate ce pot avea milioane de parametrii.

În timpul procesului de antrenare a unei rețele neuronale, obiectivul este acela de a minimiza o *funcție de cost*, pentru a măsura precizia modelului creat în raport cu setul de date. Capacitatea și performanța unei rețele neuronale de a învăța structura datelor este realizată cu ajutorul *hiperparametrilor*. Menționăm, *learning rate*, sau rata de învățare, ca fiind un criteriu ce controlează cât de mult se vor modifica parametrii modelului în raport cu valoarea funcției de cost. Prin determinarea valorii ideale a ratei de învățare, se asigură convergența către minimul global al funcției de cost într-un timp favorabil.

În această teză, prezentăm diferiți algoritmi de *line search*, ce vor determina valoarea optimă a ratei de învățare în timpul procesului de antrenare. Obiectivul acestei lucrări este acela de a studia convergența în cazul problemelor convexe și ne-convexe folosind metodele *backtracking*, *Goldstein* și *Weak Wolfe*.

În prima parte a tezei, introducem conceptul de rețele neuronale și descriem procesul de optimizare folosind metode de ordinul întâi (prima derivată a funcției de cost) și de ordinul al doilea (a doua derivată a funcției de cost). În continuare, vom sintetiza metodele de căutare a ratei de învățare și vom ilustra diferențele dintre acestea. În final, descriem succint implementarea arhitecturilor pentru probleme convexe și ne-convexe, prezentând rezultatele experimentale pentru aceste metode.

Concluzionăm faptul că, folosind metodele de căutare a ratei de învățare, modelele au o performanță mai ridicată, însă trebuie să fim atenți la capacitatea acestora de *generalizare*.

Acknowledgements

I would like to thank Bogdan Alexe, Associate Professor, PhD at Faculty of Mathematics and Computer Science, University of Bucharest, for his supervision and guidance throughout the thesis study. I appreciate his feedback, patience and constant help during my Master's studies.

I would like to thank Dan Alistarh, Associate Professor, PhD at Institute of Science and Technology, Austria, for giving me the chance to work as Scientific Intern in his research group. I appreciate his advices, motivation and constant support during my research activity at IST Austria. I would like to thank to all my colleagues from Alistarh research group for their help and encouragements.

I would like to thank Radu Gramatovici and Cristian Kevorchian, Associate Professors, PhDs at Faculty of Mathematics and Computer Science, University of Bucharest, for aspiring guidance and friendly advices during my Master's studies.

In the end, I would like to thank my family and friends for their continuous support, optimism and enthusiasm during my studies. I will forever be grateful.

Table of contents

List of Figures

List of Tables

Chapter 1

Introduction

Artificial intelligence (AI) [1] represents an area of computer science that focuses on the creation of smart systems that behave and work like humans. Today, it is a well developed field with active research subjects and many practical applications, used to automate different routine activities, understand images, recognize speech and make diagnoses in medicine. The real challenge to AI is demonstrated to be answering the tasks that are straightforward for people to accomplish, but difficult for people to describe formally - problems that we answer instinctive like recognizing words in text or faces in images. Machine Learning (ML) is a sub-field of AI. The most common objective of ML is to recognize the pattern and the structure of data. In order to do this, ML algorithms will fit the data into models which are utilized in data analysis. Even if ML is a sub-field within computer science domain, it differs from the traditional approaches, where the algorithms represent sets of programmed steps known as instructions used by the machine to solve a given problem. These ML algorithms allow the computers to train on input data and using statistical analysis to output values for unseen data. During the process of building the models, they are able to automatically learn and make progress without defining any explicitly instructions. Two of the most broadly methods in defining these models are:

1. **supervised learning** - the model maps the input to output labels in the case of **classification** task and the input to continuous output in the case of **regression** task;

2. **unsupervised learning** - the model learns to inherent the pattern of the data without having labels for. This method solve problems like **clustering** and **representation learning**.

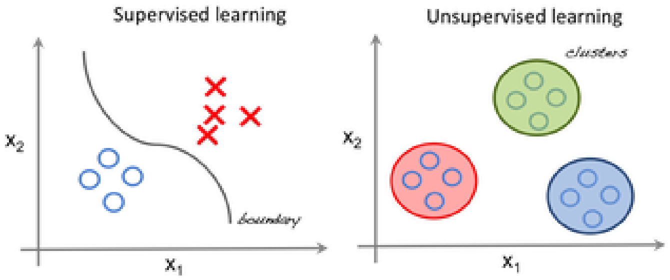

Figure 1.1: *Supervised and Unsupervised learning methods. Source: George Seif. [2]*

Artificial Neural Network [3, 4] (ANN - Figure 1.2) represents a computational model with a similar structure and functionality as biological neural network. It is a nonlinear model extensively used in ML tasks. There are three different layers in an artificial neural network:

1. **input** - the nodes provide data to the network. At this level, no computational operations are performed. The nodes just pass on the information to the hidden nodes;
2. **hidden** - the nodes perform computational operations and transfer information from the input nodes to the output nodes;
3. **output** - the nodes are responsible for computations and returning information for the data.

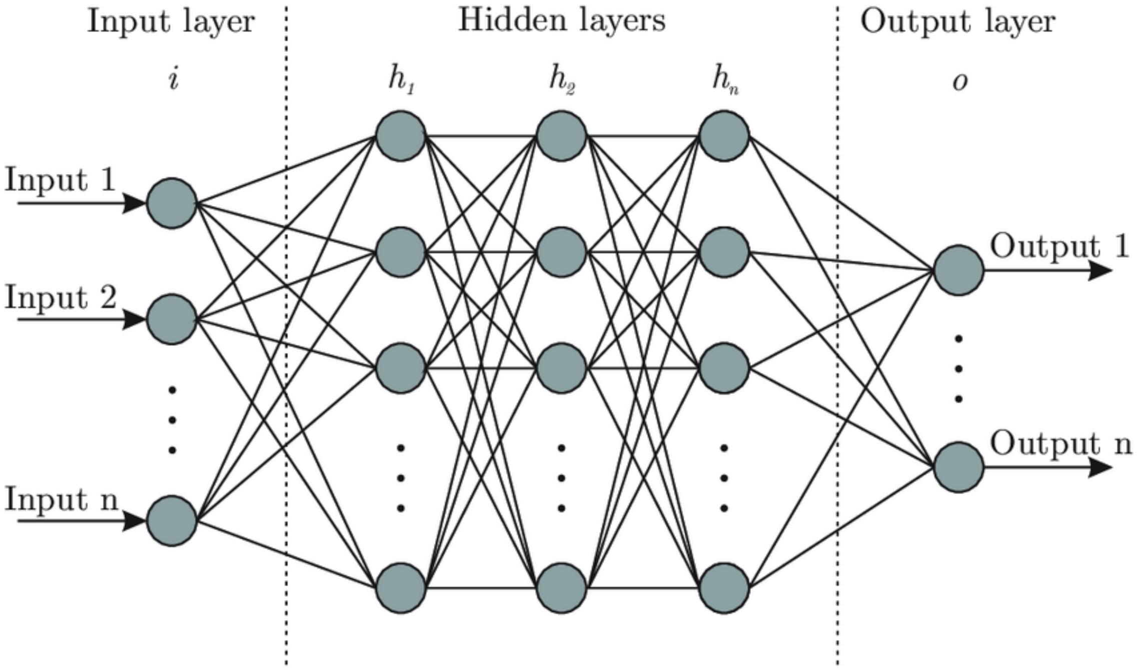

Figure 1.2: *The architecture of artificial neural network presents the three types of layers: input, hidden and output. The data are passed into the model in order to get a prediction for every sample. The data enter into the model through the input layer. During the computational process useful information is extracted from the data. Source: Facundo Bre. [5]*

Artificial neural network is defined by interconnected neurons. Each of the existing connections has associated a weight value which indicates the influence of that relationship when is multiplied by the corresponding input value during the computational process. Each neuron that is part of an artificial neural network has an activation function associated which determines the output value of the neuron. The activation function will add the non-linearity to the network.

During the training of an artificial neural network, the parameters of the model (weights and biases) need to be learn. This is an iterative process of fitting the data into the model and determining how accurate is the computed model.

The first step of this process is called **forward-propagation** and it happens when the network takes the training data and will fit these into it in order to compute the predictions (labels). Passing the data into the network will apply all the computational transformations from neurons and it will reach the final layer of the network returning a prediction for each sample. In this thesis our experiments and assumptions are made in a supervised learning scenario (for each sample we have it's label). In order to measure the performance of a model and to estimate how precise the predictions are, we will use a **loss function**. In the evaluation step, a high value of loss function will indicate that predictions are totally wrong. This means that during the training process the objective is to minimize the value of the loss function by updating the parameters of the model.

In order to have a better model, the value of the loss function is used in the **backward-propagation** step. Iteratively, from the output layer to input layer, the process will describe the contribution of each neuron to the value of the loss function. In this way the model will be able to update the parameters of neurons with respect to their impact. The process of training an artificial neural network can be visualized in Figure 1.3.

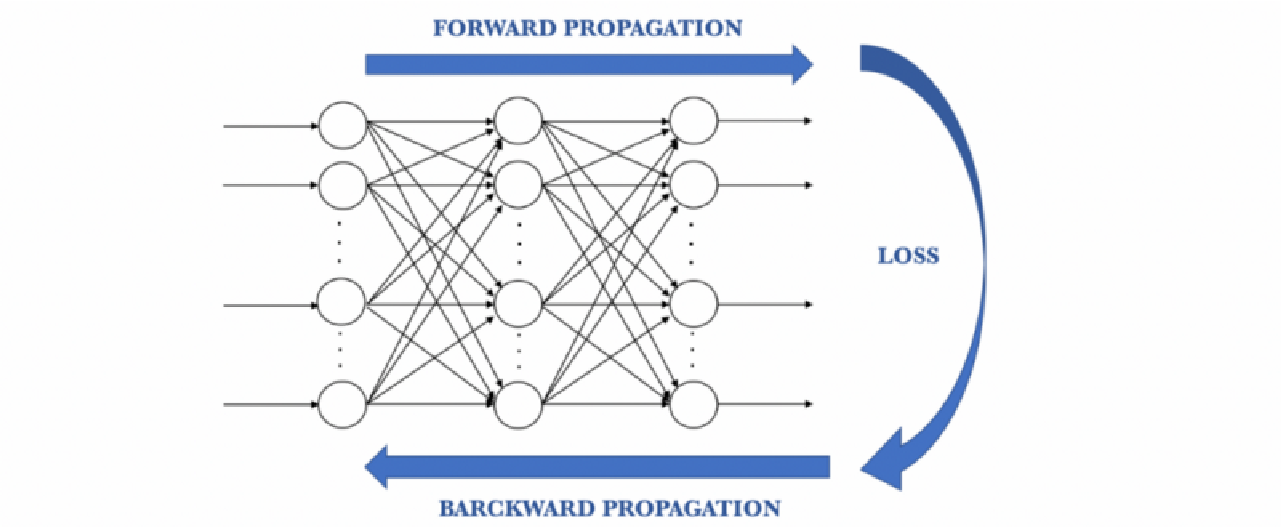

Figure 1.3: *The process of training an artificial neural network consists on two main steps: forward propagation, when the data are passed into model in order to get the prediction for each sample and backward propagation, when the model computes the impact of every neuron to the final result. Source: Jordi Torres. [6]*

The learning algorithm is based on the following steps:

1. the model is defined with starting values for the parameters (often random);
2. training samples are passed through the model to predict the labels (forward-propagation);
3. compare the computed predictions with the training labels and calculate the value of the loss function;
4. propagate the error value to each neuron (backward-propagation);
5. using the loss value, the parameters are updated in order to obtain a better model;
6. iterate these steps until a stop condition is met.

In their work, Goodfellow et al. explain that the optimization algorithms used in training neural networks to reduce the value of loss function and getting a better model differ from the traditional optimization approach. During the training process, a performance measure P is computed with respect to the test set. The main idea of the optimization task is to reduce the value of a cost function $J(\theta)$, in order to obtain a better performance measure P. The cost function is defined as an average over the samples from training set:

$$J(\theta) = \mathbb{E}_{(x,y)\sim\hat{p}_{data}} L(f(x;\theta),y), \tag{1.1}$$

where L is the loss function, $f(x;\theta)$ is the predicted value of the model for the x input and $\hat{p}_{data}$ is the empirical distribution. In this thesis, as we are working on the supervised learning case, the arguments of the loss function, L, are $f(x;\theta)$, the predicted output of the model for x and y, the label for x.

The equation 1.1 specifies the value of cost function, $J(\theta)$, with respect to the training samples. Normally, it is better to minimize $J(\theta)$ where the expectation is computed over the data-generating distribution, p_{data}, not just over a few samples that form a finite training set:

$$J^*(\theta) = \mathbb{E}_{(x,y)\sim p_{data}} L(f(x;\theta),y). \tag{1.2}$$

ML optimization algorithms intent to minimize the expected generalization error given by the equation 1.2 of $J^*(\theta)$. In order to train a neural network we use only a set of samples from the $p_{data}(x,y)$ distribution. In this way, the true distribution, $p(x,y)$, is replaced with the empirical distribution, $\hat{p}(x,y)$, which is defined with a set of samples and so we will minimize the empirical risk:

$$\mathbb{E}_{x,y\sim\hat{p}_{data}(x,y)}[L(f(x;\theta),y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta),y^{(i)}), \tag{1.3}$$

where m is the total number of samples from the empirical distribution.

Optimization algorithms are divided in two main parts, based on the number of samples that are used in one update step:

1. **batch** methods, the update will process all the training samples simultaneously in a large batch;

2. **stochastic** methods, the update will process only a single sample at a time.

In the optimization process of neural networks, algorithms fall somewhere in between. They are using more than one example at the update step, but fewer than all training samples. These methods are called **mini-batch** or **mini-batch stochastic** methods. In Figure 1.4 we can observe the differences (in the evolution of loss function) between using mini-batch method with 20 samples per update step and stochastic method with 1 sample per update step.

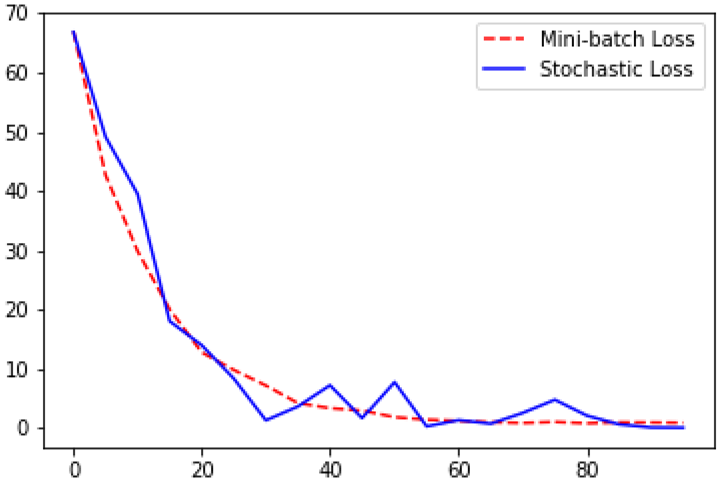

Figure 1.4: *Differences between training a neural network in mini-batch mode with 20 samples per update step and stochastic mode with 1 sample per update step. Using mini-batch mode the optimization process is smooth and the loss function is not noisy. In this way the model trained in mini-batch mode is predicting the data more precisely than the stochastic mode.*

From the above plot, we can notice that **mini-batch** training is smooth while the **stochastic** training is noisy. The experiment is implemented in **Python** using **Tensorflow** machine learning framework [7]. We have a random distribution of points and we want to fit the data using a **linear regression** model. In order to find the best model we go through a number of epochs and we adjust the parameters of the model based on a different mode: **stochastic** where the update is calculated with respect to 1 sample and **mini-batch** where the update is calculated with respect to 20 samples.

In neural network training process we have to choose the optimal hyperparameters that provide a good model. **Learning rate** is a hyperparameter that handles how much we will adjust the weights of our network with respect to the loss function. In this thesis we will present some methods of searching an optimal learning rate in both convex and non-convex optimization problems.

Chapter 2

Convex and non-convex optimization

Convex optimization problems [8] are more general than linear programming problems. An important aspect is represented by the fact that they share desirable properties of linear programming problems. In the optimization process, the convex problems can be solved rapidly even if they have a large number of parameters or constraints. Algebraically, a convex optimization problem has the following general form:

$$\text{minimize } f_0(x), \text{ subject to } f_i(x) \leq b_i, 1 \leq i \leq m, \tag{2.1}$$

where the functions $f_0 \ldots f_m : \mathbb{R}^n \to \mathbb{R}$ are convex functions, *i.e.*, satisfy

$$f_i(\alpha x + \beta y) \leq \alpha f_i(x) + \beta f_i(y), \tag{2.2}$$

for all $x,\ y \in \mathbb{R}^n$ and all $\alpha, \beta \in \mathbb{R}$ with $\alpha + \beta = 1, \alpha \geq 0, \beta \geq 0$.

From a geometric point of view a function is convex if a line segment that is drawn from any existing point, $(x, f(x))$, to another one, $(y, f(y))$, which is called the chord from point x to y lies above or on the graph of f function, as presented in Figure 2.1.

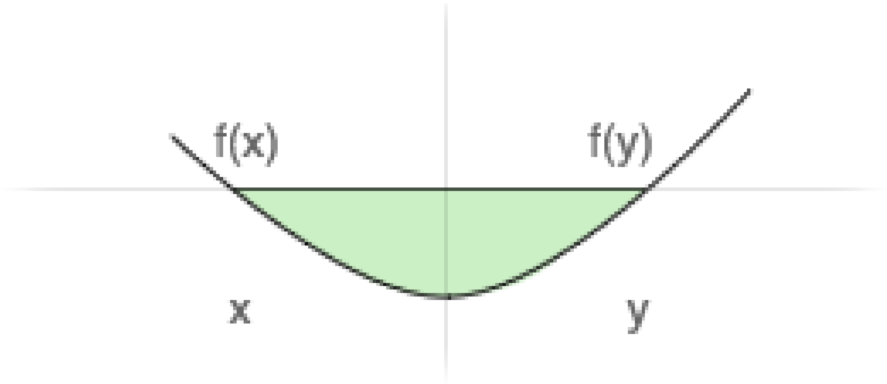

Figure 2.1: *In the convex problem case the chord from $(x, f(x))$ to $(y, f(y))$ lies above or on the graph of the function.*

A convex optimization problem is a problem where all of the existing constraints are convex functions. In the convex case, the objective is represented as a convex function if it is minimizing, or as a concave function if it is maximizing. In Figure 2.2 we present a convex function, with an important theoretical property - any local minimum, also known as stationary point, is a global minimum.

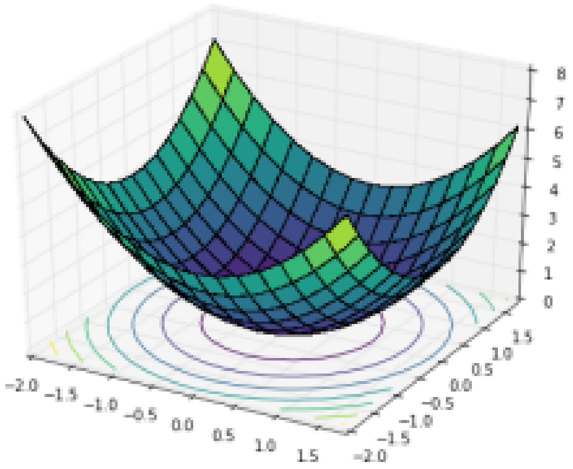

Figure 2.2: *Any local minimum point from a convex problem, also known as stationary point, is a global minimum. Source: Jan Mrkos. [9]*

Linear functions are defined as convex, so linear programming problems are convex problems. In this thesis we use as convex optimization problem - **linear regression**, which applies a linear transformation to the incoming data: $y = x * A^T + b$, where A, b are parameters, x data.

Non-convex optimization problems are problems where the objective or any of the constraints are non-convex. Geometrically, a non-convex function is curving up and down, so it is neither concave nor convex. As an example we can take *sine* function, presented in Figure 2.3, where the function is convex from $-\pi$ to 0, and concave from 0 to $+\pi$.

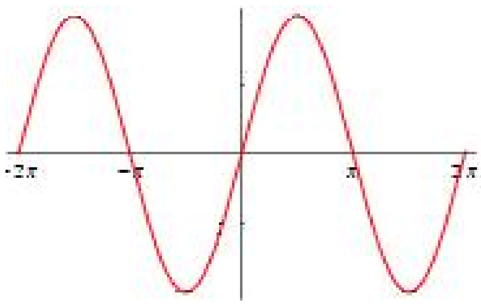

Figure 2.3: *In the non-convex problem case the function is curving up and down, so it is neither concave nor convex.*

Non-convex problems can have multiple locally optimal points and feasible regions in each region, as presented in Figure 2.4. In order to find an optimal solution of the non-convex problem, which is the **global optimum** across all existing feasible regions, we need to take an exponential time in the number of parameters and constraints.

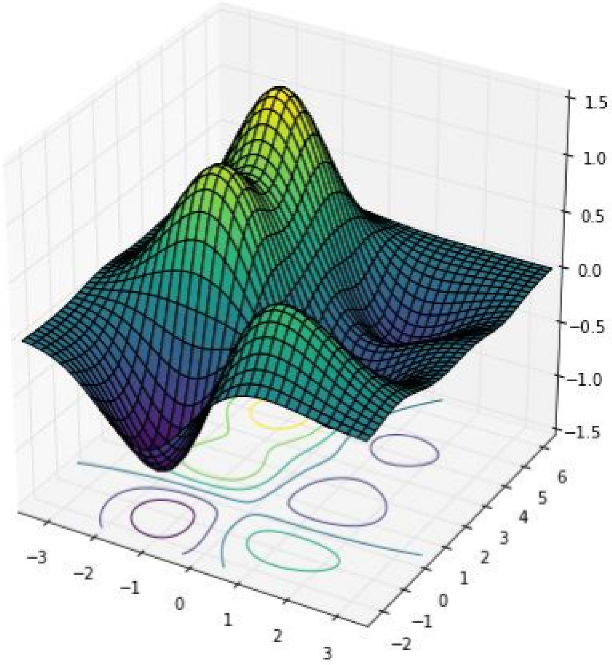

Figure 2.4: *The optimal solution of a non-convex problem, which is represented by the global optimum, can be found in exponential time, based on the number of parameters and constraints of the model. Source: Jan Mrkos. [9]*

In modern applications, the characteristic objective of the learning task is represented by a non-convex function [10]. As an example, we can take the training of deep neural networks or tensor decomposition problems. Despite that non-convex functions give the possibility to precisely model learning problems, they represent a difficult optimization task. The challenge is to design the algorithms for these non-convex problems.

The methods that are frequently used in case of non-convex optimization include simple and effective primitives such as gradient descent or stochastic optimization. These are very fast in practice and remain favorites of practitioners. In this thesis we use as non-convex optimization problem a neural network and more specifically a Convolutional Neural Network (CNN) [11] which is most commonly applied to analyzing visual imagery. A CNN is an algorithm which takes as input an image and predicts it's label, by applying several operations as presented in Figure 2.5.

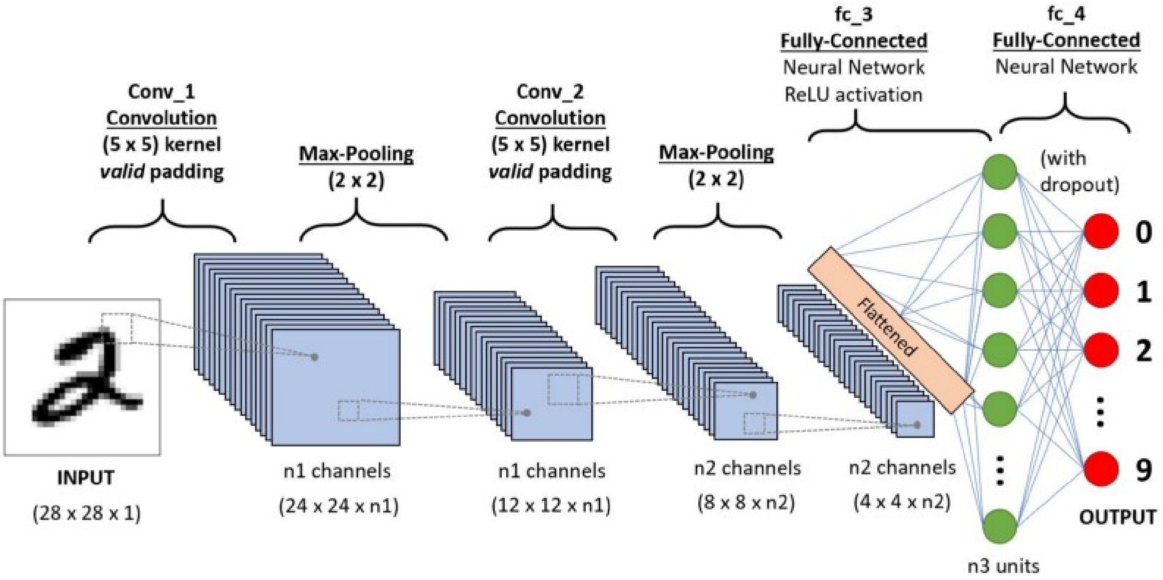

Figure 2.5: *In this neural network, the data are represented by images with handwritten digits with dimension* 28×28 *pixels. After the process of feeding the data into the model, the hidden layers apply different operations (max-pooling, convolutional kernels) in order to extract the useful information from images and to make the predictions. Source: Sumit Saha. [12]*

Chapter 3

First and Second order methods for optimization

3.1 First order methods

Algorithms that use the first-derivative or gradient of the loss function in order to update the parameters of a model are defined as **first order** [13] algorithms.

The gradient of the loss function with respect to each neuron that is part of our model is computed during the backward-propagation step. The result shows the impact of each neuron to the final model. After this step, the determined gradients are used to perform the update step, using several approaches, like vanilla stochastic gradient descent, momentum stochastic gradient descent, nesterov momentum stochastic gradient descent [14]. The optimization methods iterate the following update:

$$x = x - \eta \nabla f(x), \tag{3.1}$$

where $\nabla f(x)$ is the Jacobian matrix, $n \times n$ square matrix of first-order partial derivatives of the function, where n is the number of parameters of the model and η the step size, also known as learning rate and x represent the parameters of the model (weights and biases). Gradient descent method uses the partial derivative of the computed loss function to propagate the updates to the network. The general idea of gradient descent algorithm is presented in Figure 3.1.

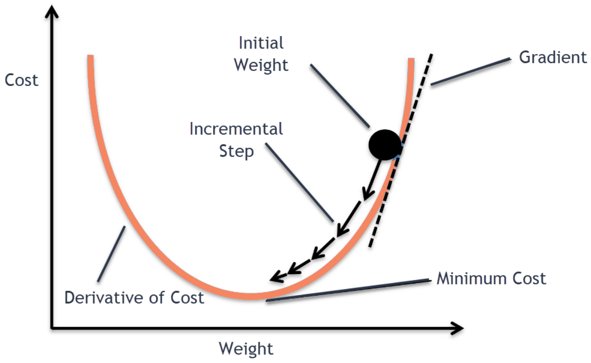

Figure 3.1: *First derivative of the loss is used in the update step. Source: Divakar Kapil. [15]*

3.2 Second order methods

Algorithms that require second-derivative of the loss function in order to update the parameters of a model are defined as **second order** algorithms. In the case of deep learning, these optimization methods are based on Newton's method, which iterates the following update:

$$x = x - [Hf(x)]^{-1}\nabla f(x), \tag{3.2}$$

where $Hf(x)$ represents the Hessian matrix, $n \times n$ square matrix of second-order partial derivatives of the function, n is the number of parameters of the model and x represent the parameters of the model (weights and biases). The Hessian matrix is used to describe the local curvature of the error function. It also allows the optimization process to perform a more efficient update for a specific model.

The update based on the second-order partial derivatives is impractical for most neural network models because computing the Hessian matrix and then inverting it is a very costly process in both space and time. For example, a neural network that has 1.000.000 parameters will compute a Hessian matrix of size [1.000.000 × 1.000.000], occupying a huge amount of memory. Consequently, quasi-Newton methods [16] have been developed. These methods try to make an approximation of the inverse of Hessian matrix. The most popular algorithm for the approximation inverse of Hessian matrix is Limited-memory Broyden – Fletcher – Goldfarb – Shanno (L-BFGS) [17], which uses the information in the gradients during the optimization process to form the approximation implicitly. In this way, the whole Hessian matrix is never computed.

3.3 Hyperparameter optimization

Learning rate, also known as step size, is a hyperparameter that handles how effectively the model is updated with respect to the computed loss function. If learning rate is low, the training process will be slow, because model is making tiny updates of the parameters. However, if the value of learning rate is high, the training process will be faster, but this way can cause abnormal behavior of the loss function. In Figure 3.2 we can observe the differences between using a low value for learning rate (*left*) and a high value for learning rate (*right*).

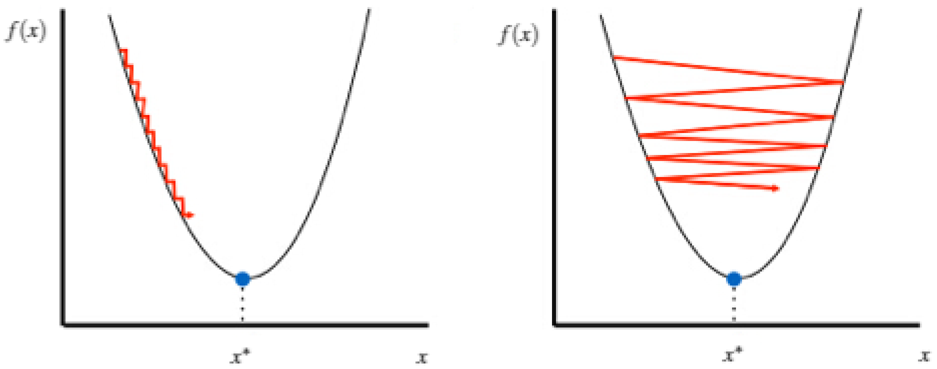

Figure 3.2: *Using a low value of learning rate makes tiny updates for the parameters of the model and the training process takes too much time. In comparison, using a high value of learning rate makes huge updates and the loss function is diverging from the minimum value, making the training process difficult.*

In Figure 3.3 we present the evolution of the loss function in three different phases:

- on the *left* side, learning rate is low and the loss is decreasing, but very slow;
- on the *middle* side, learning rate is optimal and we get a quick drop for loss function;
- on the *right* side, learning rate is high and the loss is diverging from the minimum value.

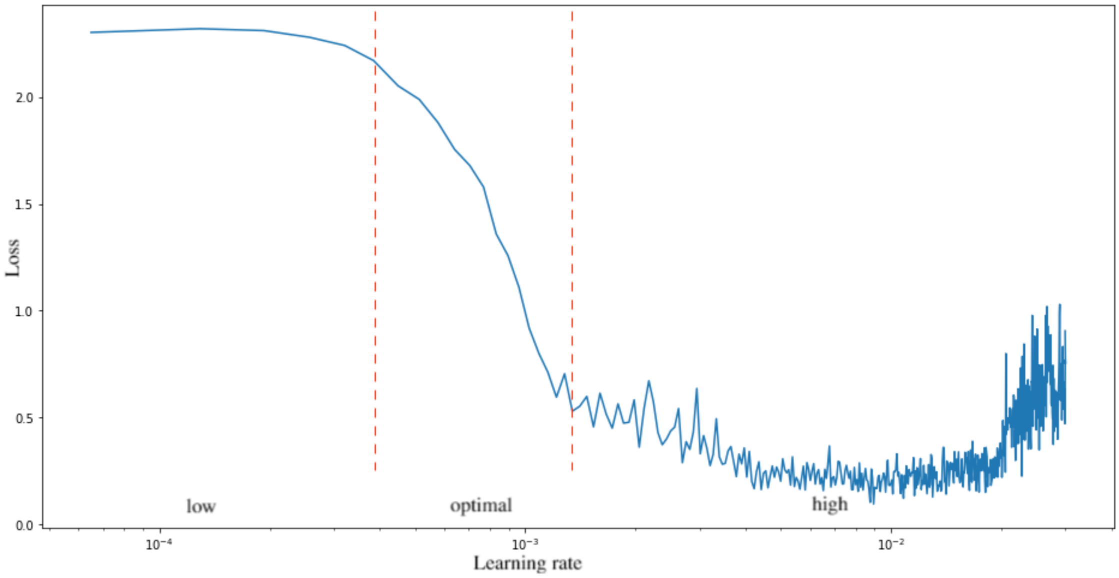

Figure 3.3: *The evolution of loss function can be represented in three different phases: (left) when the learning rate is low and the model is updating slowly, (middle) when the learning rate is optimal and the model has a good update rate, (right) when the learning rate is high and the model is diverging. Source: Jeremy Jordan. [18]*

In this thesis we will present different **line search** methods that are used to find the optimal learning rate value during the optimization process in order to achieve a local minimum of a loss function.

Chapter 4

Line search methods

Line search methods [19] are defined as algorithms that try to reach the minimum value of a nonlinear function by selecting a reasonable **direction vector**. Also, they provide a function value which is closer to absolute minimum when the selected direction vector is iteratively computed with an optimal **step size** value. For a given function $f(x)$ and an initial x_k, in order to find a lower value of function, x_{k+1} needs to be increased by the following iteration scheme:

$$x_{k+1} = x_k + \alpha_k p_k, \tag{4.1}$$

where $\alpha_k > 0$ represents the step size, p_k represents the direction vector and x represent the parameters of the model (weights and biases).

The general algorithm of a line search method is presented in Figure 4.1. The main idea is that for a given step direction and step size we evaluate how accurate is the model, by computing the value of the loss function. If the model has good predictions of data, the algorithm stops, otherwise the value of α_k is updated and we evaluate the model based on the new update.

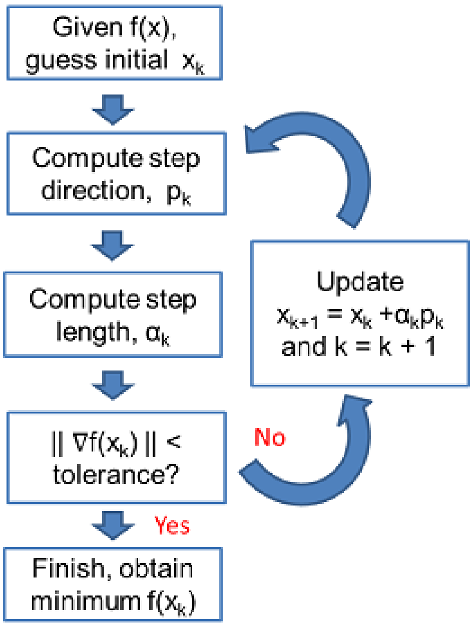
Given f(x), guess initial x_k
Compute step direction, p_k
Compute step length, α_k
$\| \nabla f(x_k) \| <$ tolerance?
Yes
Finish, obtain minimum $f(x_k)$
No
Update $x_{k+1} = x_k + \alpha_k p_k$ and k = k + 1

Figure 4.1: *Line search algorithm evaluates how accurate is the updated model for a given pair of direction vector and step size. Source: Elizabeth Conger. [20]*

If the algorithm finds a value of α_k such that the function $f(x)$ in the p_k direction is minimized, *i.e.*,

$$f(x_k + \alpha_k p_k) = \min_{\alpha_k} f(x_k + \alpha_k p_k), \ \alpha_k > 0, \tag{4.2}$$

this line search method is an **optimal** or **exact** line search method and α_k is an optimal step length. This α_k value will reach the global minimum of the function, as presented (Figure 4.2) by Nocedal and Wright in their work [21].

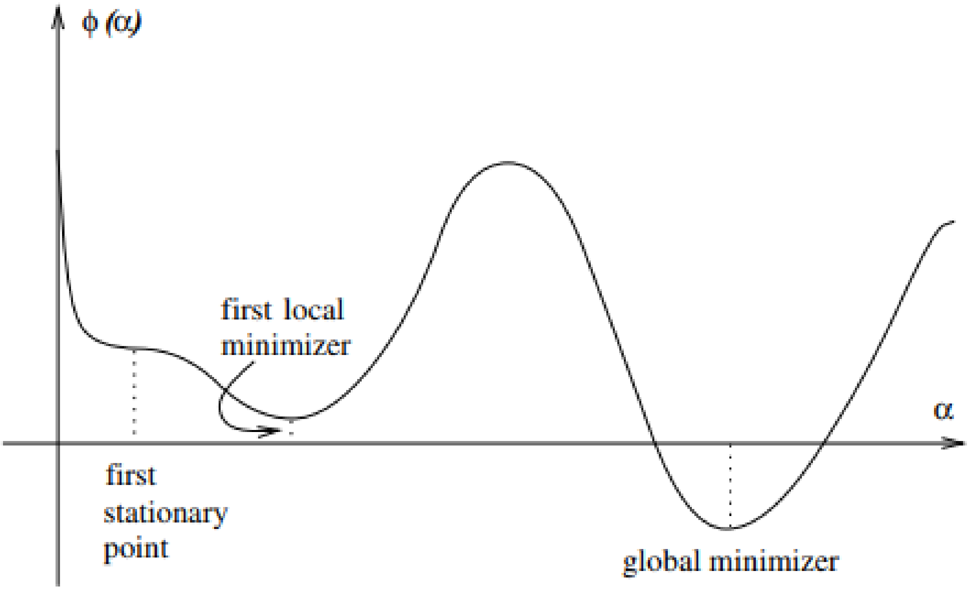

Figure 4.2: *The best value for the step length parameter is the one that reaches the global minimum of a function. Source: Nocedal and Wright. [21]*

If α_k is chosen such that the function $f(x)$ will provide an acceptable descent value, *i.e.*,

$$f(x_k) - f(x_k + \alpha_k p_k) > 0, \tag{4.3}$$

this line search method is an **approximate** or **inexact** line search method and α_k is an approximate step length.

In practical situations, exact optimal value of α_k cannot be found, because the process is too costly in time and memory. Inexact line search methods provide step size values with less computational overhead. In this thesis we present inexact line search techniques (**backtracking**, **goldstein** and **weak wolfe**) for optimization process in convex and non-convex cases.

4.1 The backtracking method

The backtracking line search method with Armijo condition goes through the following steps:

1. Given $\alpha_{init} > 0$ (e.g. $\alpha_{init} = 1$), let $\alpha_{(0)} = \alpha_{init}$ and $l = 0$, where p_k is the direction vector.
2. For $\beta \in (0,1)$, as long as the condition

$$f(x_k + \alpha_{(l)} p_k) \leq f(x_k) + \alpha_{(l)} \beta [\nabla f(x_k)]^T p_k$$

is not satisfied:

- set $\alpha_{(l+1)} = \tau \alpha_{(l)}$, where $\tau \in (0,1)$ is fixed (e.g., $\tau = \frac{1}{2}$);
- $l = l + 1$.

3. Set $\alpha_k = \alpha_{(l)}$.

The impact of the regularization term, β, is presented in Figure 4.3:

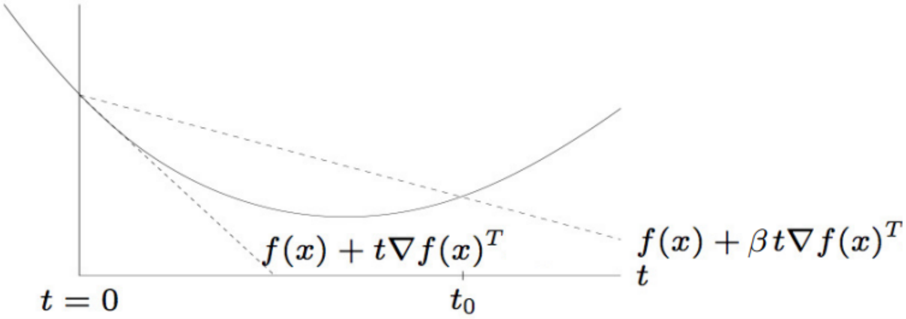

Figure 4.3: *The regularization term, β, controls how fast the descent of the function will be. Source: Boyd and Vandenberghe. [8]*

Backtracking line search algorithm with Armijo condition ensures that during the optimization process:

- the direction of search is a descent direction;
- β is a regularization term, in order to control how fast the descent will be done.

The backtracking algorithm provides a value of step size, α_k, that is fixed or that it is small enough to meet the sufficient decrease condition (Figure 4.4).

4.2 The Goldstein method

The Goldstein line search method has two reasonable conditions:

1. $f(x_k + \alpha p_k) \leq f(x_k) + \rho \alpha g_k^T p_k$ - sufficient decrease condition (Figure 4.4)

2. $f(x_k + \alpha p_k) \geq f(x_k) + (1-\rho)\alpha g_k^T p_k$ - control step size condition (Figure 4.5),

where $0 < \rho < \frac{1}{2}$ and $g_k = \nabla f(x)$.

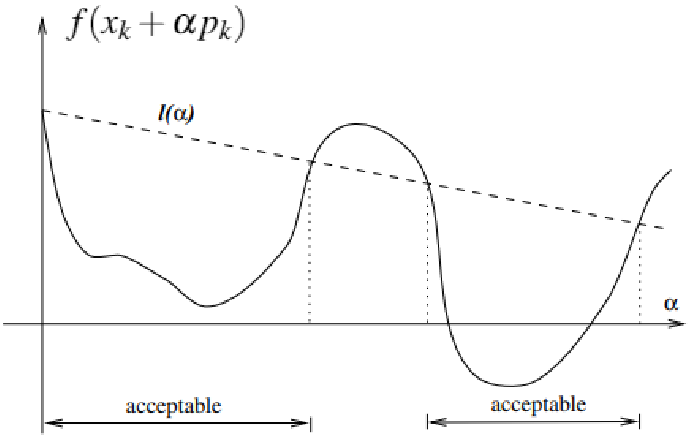

Figure 4.4: *The condition ensures that using the step length, α, the function f decreases considerably. Source: Nocedal and Wright. [21]*

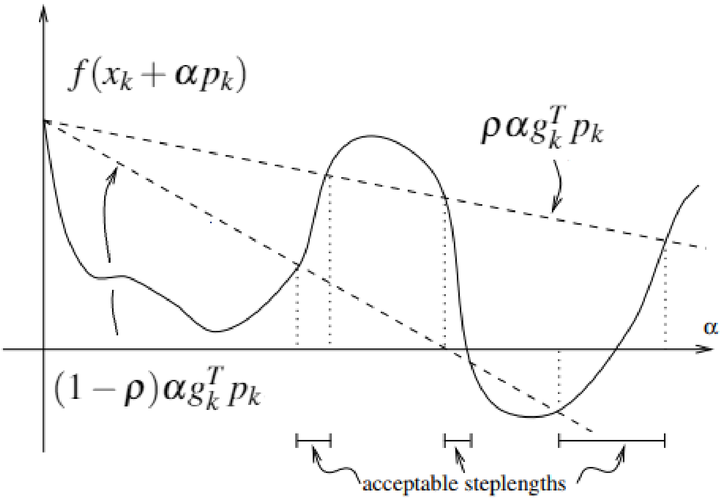

Figure 4.5: *The condition controls the impact of the step length, in order to have considerably decrease of the function f. Source: Nocedal and Wright. [21]*

The Goldstein algorithm goes through the following steps, presented in Figure 4.6:

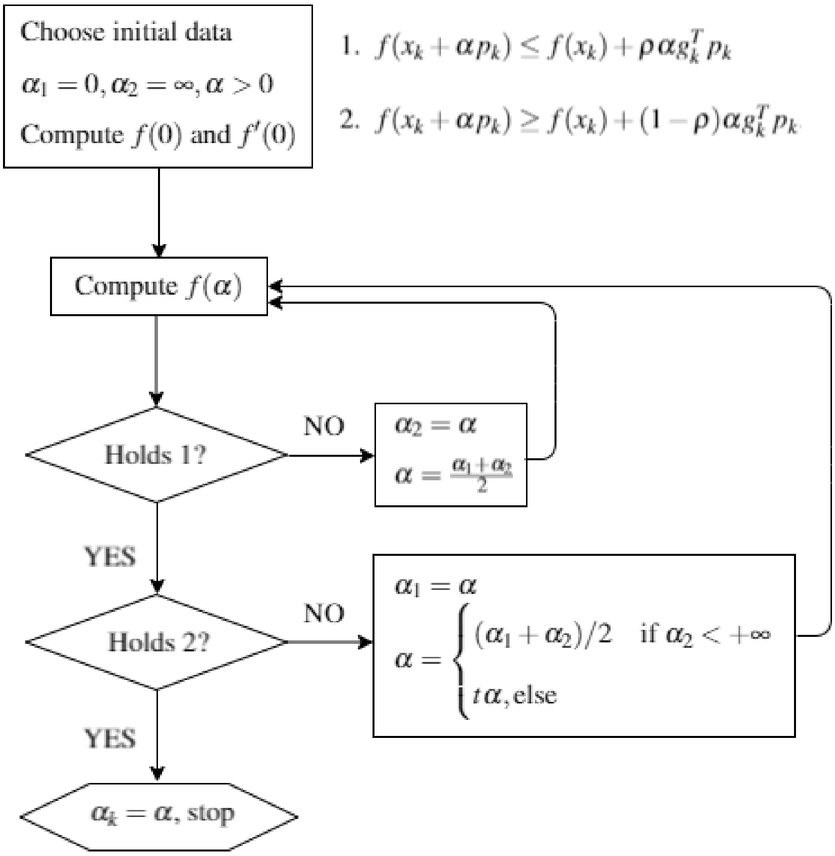
Choose initial data
$\alpha_1 = 0, \alpha_2 = \infty, \alpha > 0$
Compute $f(0)$ and $f'(0)$
1. $f(x_k + \alpha p_k) \leq f(x_k) + \rho \alpha g_k^T p_k$
2. $f(x_k + \alpha p_k) \geq f(x_k) + (1-\rho) \alpha g_k^T p_k$
Compute $f(\alpha)$
Holds 1?
NO
$\alpha_2 = \alpha$
$\alpha = \frac{\alpha_1 + \alpha_2}{2}$
YES
Holds 2?
NO
$\alpha_1 = \alpha$
$\alpha = \begin{cases} (\alpha_1 + \alpha_2)/2 & \text{if } \alpha_2 < +\infty \\ t\alpha, \text{else} \end{cases}$
YES
$\alpha_k = \alpha$, stop

Figure 4.6: *During the Goldstein algorithm the conditions are checked in order to stop the line search. When both conditions are satisfied, the value of α_k is returned. Source: Wenyu Sun and Ya-Xiang Yuan. [19]*

The Goldstein conditions confirm that the step size, α_k, provides sufficient decrease of the loss function during the optimization process. Also, the conditions avoid that the value of α_k will become too small.

4.3 The Weak Wolfe method

The Weak Wolfe line search method has two reasonable conditions:

1. $f(x_k + \alpha p_k) \leq f(x_k) + \rho \alpha g_k^T p_k$ - sufficient decrease condition (Figure 4.4)

2. $g_{k+1}^T p_k \geq \sigma g_k^T p_k$ - curvature condition (Figure 4.7),

where $0 < \rho < \frac{1}{2}$, $\sigma \in (\rho, 1)$ and $g_k = \nabla f(x)$.

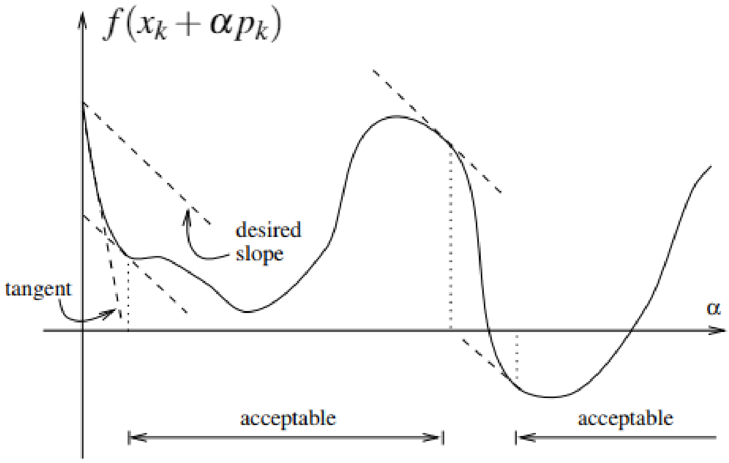

Figure 4.7: *The condition ensures that using the step length, α, the slope of the function f is reduced considerably. Source: Nocedal and Wright. [21]*

The Weak Wolfe algorithm goes through the following steps, presented in Figure 4.8:

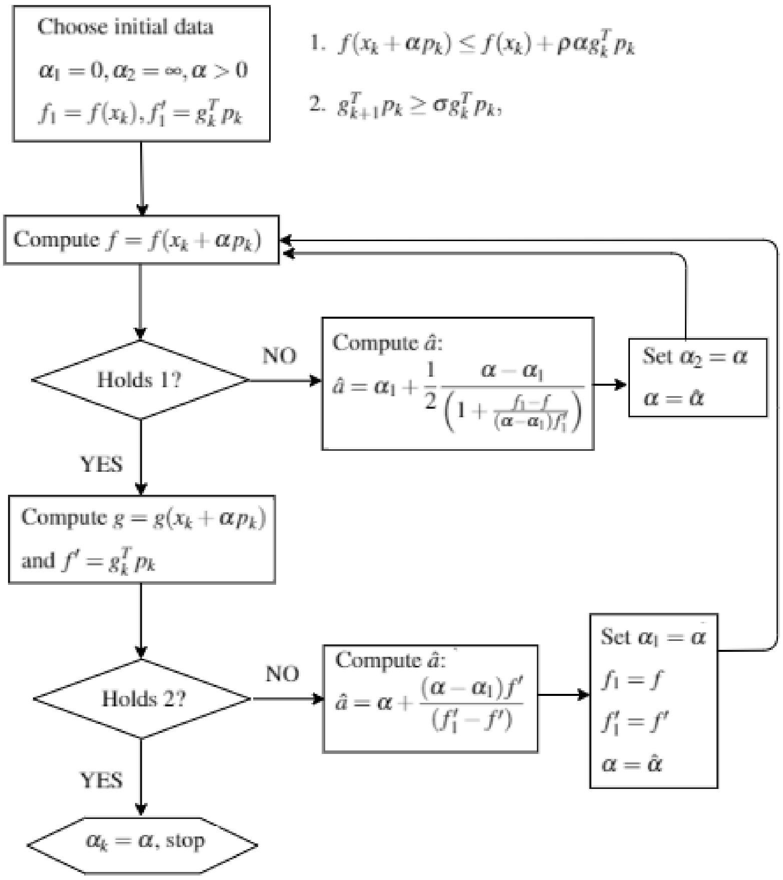
Choose initial data
$\alpha_1 = 0, \alpha_2 = \infty, \alpha > 0$
$f_1 = f(x_k), f_1' = g_k^T p_k$
1. $f(x_k + \alpha p_k) \le f(x_k) + \rho \alpha g_k^T p_k$
2. $g_{k+1}^T p_k \ge \sigma g_k^T p_k$,
Compute $f = f(x_k + \alpha p_k)$
Holds 1?
NO
Compute $\hat{a}$:
$\hat{a} = \alpha_1 + \frac{1}{2} \frac{\alpha - \alpha_1}{\left(1 + \frac{f_1 - f}{(\alpha - \alpha_1) f_1'}\right)}$
Set $\alpha_2 = \alpha$
$\alpha = \hat{\alpha}$
YES
Compute $g = g(x_k + \alpha p_k)$
and $f' = g_k^T p_k$
Holds 2?
NO
Compute $\hat{a}$:
$\hat{a} = \alpha + \frac{(\alpha - \alpha_1) f'}{(f_1' - f')}$
Set $\alpha_1 = \alpha$
$f_1 = f$
$f_1' = f'$
$\alpha = \hat{\alpha}$
YES
$\alpha_k = \alpha$, stop

Figure 4.8: *During the algorithm the conditions are checked in order to stop. When the conditions are satisfied, α_k is returned. Source: Wenyu Sun and Ya-Xiang Yuan. [19]*

The Wolfe conditions confirm that the step size, α_k, provides sufficient decrease of the loss function during the optimization process. Also, using the curvature condition if the slope of loss function is slightly positive or negative, the decrease in this direction cannot be very significant and it makes sense to finish the algorithm.

Chapter 5

Implementation and results

5.1 Optimizers

In our implementation, line search methods for convex and non-convex optimization are applied for the following algorithms:

1. *first order*: Stochastic Gradient Descent (SGD);

2. *second order*: Limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS).

5.1.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is perhaps the most used optimization algorithm in training deep neural networks. In the following algorithm we present Stochastic Gradient Descent which is described by Goodfellow et al. in their work [3].

Algorithm 1: Stochastic Gradient Descent (SGD) update

```
Input: Learning rate schedule $\varepsilon_1$, $\varepsilon_2$,...
Input: Initial parameter $\theta$
$k \leftarrow 1$
repeat
    Sample a mini-batch of m examples from the training set $x^{(1)},...,x^{(m)}$ with
    corresponding targets $y^{(i)}$.
    Compute gradient estimate: $\hat{g} \leftarrow \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta),y^{(i)})$.
    Apply update: $\theta \leftarrow \theta - \varepsilon_k \hat{g}$.
    $k \leftarrow k+1$.
until stopping criterion is met;
```

The idea behind SGD is that the algorithm computes the gradient of loss function with respect to a mini-batch of data from the entire dataset. After this step, the parameters are updated using learning rate and the computed gradients. The learning rate parameter used during the SGD algorithm may be chosen running multiple times the training and recording learning curves of the loss function.

5.1.2 L-BFGS

BFGS algorithm tries to use Newton's method during the optimization process without computational overhead. Recall that the parameters update based on Newton's method is:

$$x = x - [Hf(x)]^{-1} \nabla f(x), \tag{5.1}$$

where H is the Hessian matrix of loss function, $f(x)$. The difficulty in applying this algorithm is the computation of H^{-1}. Quasi-Newton methods make an approximation to the inverse of Hessian matrix, M_t, which is iteratively computed by low-rank updates.

In order to reduce the memory costs, the Limited-memory BFGS (L-BFGS) algorithm computes the inverse Hessian matrix, M_t, with the assumption that, M_{t-1}, is the identity matrix. Also, this approach avoids to store information from an update step to another.

5.2 Convex problem

In our experiments we used for the convex problem a linear regression model, which applies a linear transformation to the incoming data: $y = x * A^T + b$, where A, b represent the parameters and x the data. The idea consists of finding a straight fitting line through the data points. The best line that fits the data represents the *regression line*. In Figure 5.1 we can observe a linear regression model that fits the data presented in the plot.

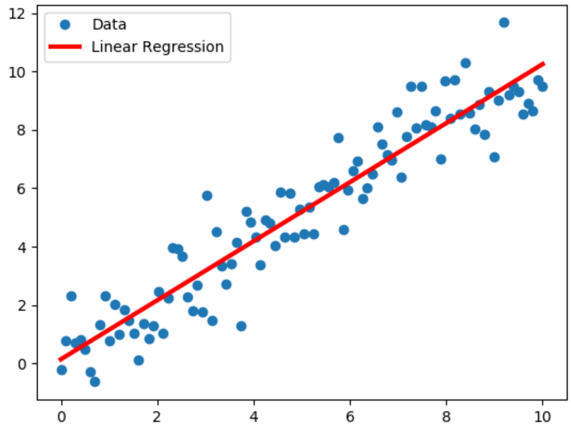

Figure 5.1: *Linear regression represents a model that fits the data using a straight line.*

5.3 Non-convex problem

In our experiments we used for the non-convex problem a Residual neural network - ResNet [22]. Many deep neural networks are performing well due to the additional layers added. The idea of making the network deeper is that the layers gradual learn more complex features or properties of images. During the training process, the first layer of the network learns the edges, the second layer learns the shapes, the objects and specific features. He et al. in their article empirically demonstrate that there is a threshold value for the depth of a convolutional neural network. In Figure 5.2 we can observe that training error (*left*) and test error (*right*) for CIFAR-10 dataset using plain CNNs are decreasing faster using a 20-layer network that 56-layer network.

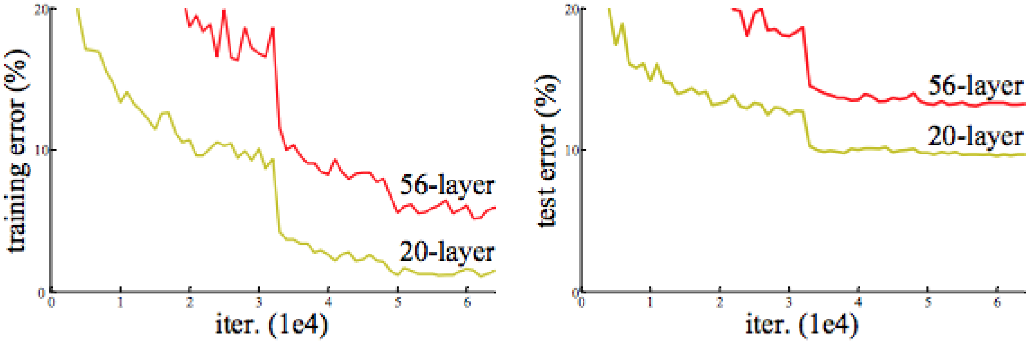

Figure 5.2: *Deep neural networks work well for a high number of layers because it learns specific features from data. He et al. empirically demonstrate that there exists a threshold value for the depth of the neural network. Source: He et al. [22]*

The problem with training deep neural networks has been reduced in residual neural networks by adding the residual block, presented in Figure 5.3, which is the most important result from "Deep residual learning for image recognition" paper.

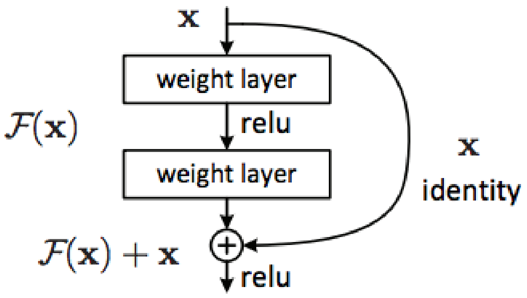
x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 5.3: *Residual blocks for vanishing gradients problem. Source: He et al. [22]*

The residual connections created between a layer and layers after it try to solve **vanishing gradients** problem. In the backward propagation step when the gradients of loss function with respect to the parameters are computed, they tend to become smaller and this behavior will cause vanishing gradients problem. In this way, the neurons from the first layers are learning very slow compared to the neurons from the last layers. In order to learn and to detect simple patterns and structure of the data, first layers of the model, also called **building blocks**, need to be well trained to give proper results. These results are then forward propagated to the next layers which will perform accurate prediction for our data. Using the skip connections, it takes longer for the model to vanish the gradients.

5.4 Datasets

In our experiments we used 2 different datasets:

1. Modified National Institute of Standards and Technology - MNIST [23] - *convex problem*;

2. Canadian Institute for Advanced Research - CIFAR-10 [24] - *non-convex problem.*

MNIST dataset consists of handwritten digits images (10 different classes for digits from 0 to 9, with dimension 28×28 pixels). The dataset contains 60k images for the training process and 10k images for the test process. Some samples from MNIST dataset are presented in Figure 5.4.

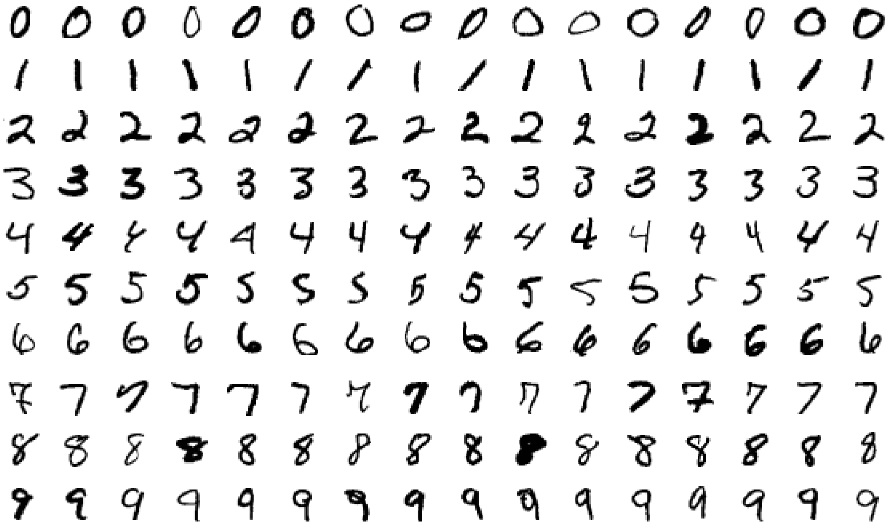

Figure 5.4: *Samples from handwritten digits dataset. Source: Yann LeCun. [23]*

CIFAR-10 dataset consists of color images (10 different classes - *airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck*, with dimension 32×32 pixels). The dataset contains 50k images for the training process and 10k images for the test process. Some samples from CIFAR-10 dataset are presented in Figure 5.5.

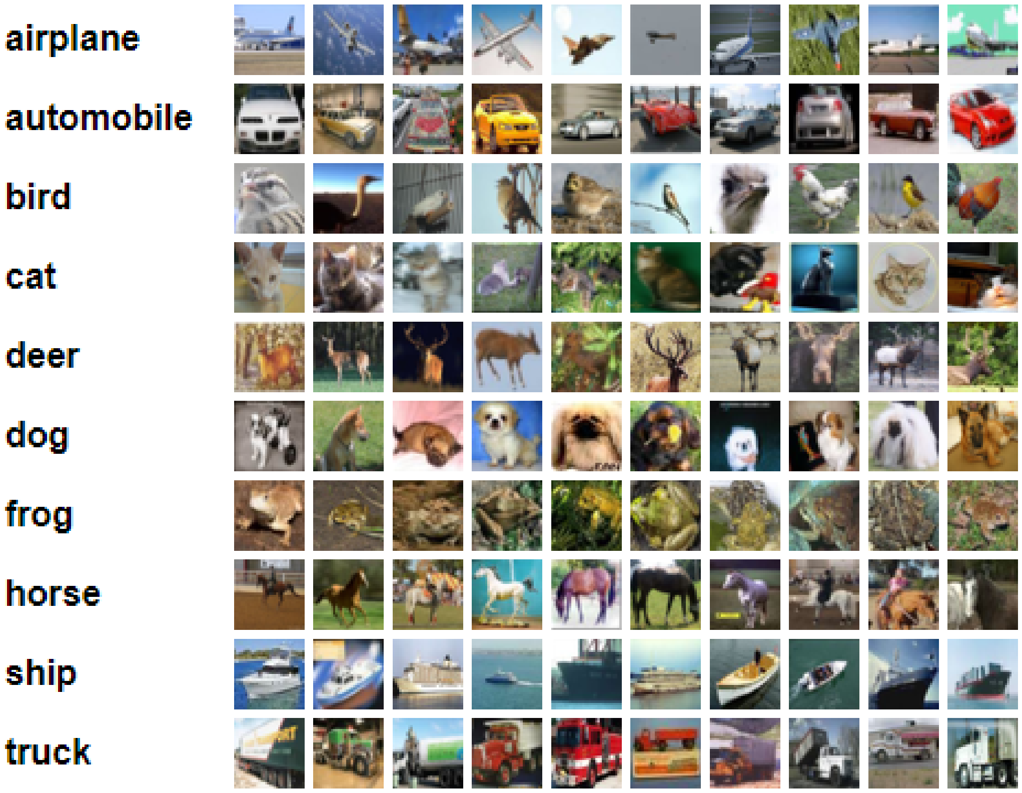

Figure 5.5: *Samples from CIFAR-10 dataset. Source: Alex Krizhevsky. [24]*

In order to train a neural network model the dataset needs to be prepared for the process. There are different steps that we should take in consideration:

1. get the data;
2. check the missing data;
3. encode the data in different formats;
4. split the dataset into train and test sets;
5. perform feature scaling, if some columns are scaled incorrectly;
6. perform same steps for unseen data that need to be predicted.

5.5 Implementation

In this section we explain how to build a classifier in order to predict data for convex and non-convex cases. Our implementation is written completely in **Python**. In order to ease the development process we used **Pytorch** to create the architectures of the models and to use the implemented optimizers. In the section 5.8 we briefly describe these technologies.

We load the modules and we define the architectures (Code Snippets 5.1, 5.2, 5.3):

```python
class LogisticRegression(nn.Module):
    def __init__(self, input_size, num_classes):
        super(LogisticRegression, self).__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, x):
        out = self.linear(x)
        return out
```

Code Snippet 5.1: *Convex architecture based on the number of final classes.*

```python
# 3x3 convolution
def conv3x3(in_channels, out_channels, stride=1):
    return nn.Conv2d(in_channels, out_channels, kernel_size=3,
                     stride=stride, padding=1, bias=False)

# Residual block
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels,
                 stride=1, downsample=None):
        super(ResidualBlock, self).__init__()
        self.conv1 = conv3x3(in_channels, out_channels, stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(out_channels, out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out
```

Code Snippet 5.2: *Residual block for non-convex architecture.*

In Code Snippet 5.1 we observe the definition of the **linear regression** model, from **torch.nn** package, based on **torch.nn.Module** class. **Linear** [25] is the class which applies the linear transformation to the incoming data.

In Code Snippets 5.2 and 5.3 we observe the definition of the **ResNet 20** model, based on **torch.nn.Module** class. It has different layers that are presented in Convolutional Neural Networks, like **convolutional**, **batch normalization**, **fully connected** and also, **residual block**, presented by He et al.

```
# ResNet
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 16
        self.conv = conv3x3(3, 16)
        self.bn = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self.make_layer(block, 16, layers[0])
        self.layer2 = self.make_layer(block, 32, layers[1], 2)
        self.layer3 = self.make_layer(block, 64, layers[2], 2)
        self.avg_pool = nn.AvgPool2d(8)
        self.fc = nn.Linear(64, num_classes)

    def make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        if (stride != 1) or (self.in_channels != out_channels):
            downsample = nn.Sequential(
                conv3x3(self.in_channels, out_channels, stride=stride),
                nn.BatchNorm2d(out_channels))
        layers = []
        layers.append(block(self.in_channels, out_channels, stride, downsample))
        self.in_channels = out_channels
        for i in range(1, blocks):
            layers.append(block(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv(x)
        out = self.bn(out)
        out = self.relu(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out

def resnet20(**kwargs):
    return ResNet(ResidualBlock, [3, 3, 3], **kwargs)
```

Code Snippet 5.3: *Non-convex architecture ResNet 20 based on the number of final classes.*

We create two **scripts**, for each problem, where we will load the datasets (MNIST for the convex problem and CIFAR-10 for the non-convex problem), Code Snippets 5.4 and 5.5:

```
# MNIST Train dataset
train_dataset = torchvision.datasets.MNIST(root ='Datasets/mnist/',
                                           train = True,
                                           transform = transforms.ToTensor(),
                                           download = True)
# MNIST Test dataset
test_dataset = torchvision.datasets.MNIST(root ='Datasets/mnist/',
                                          train = False,
                                          transform = transforms.ToTensor())
# Train dataset loader
train_loader = torch.utils.data.DataLoader(dataset = train_dataset,
                                           batch_size = args.batch_size,
                                           shuffle = True)
```

```
# Test dataset loader
test_loader = torch.utils.data.DataLoader(dataset = test_dataset,
                                          batch_size = args.batch_size,
                                          shuffle = False)
```

Code Snippet 5.4: *Load MNIST dataset for convex problem.*

```
# CIFAR-10 Train dataset
train_dataset = torchvision.datasets.CIFAR10(root ='Datasets/cifar10/',
                                             train = True,
                                             transform = transforms.ToTensor(),
                                             download = True)
# CIFAR-10 Test dataset
test_dataset = torchvision.datasets.CIFAR10(root ='Datasets/cifar10/',
                                            train = False,
                                            transform = transforms.ToTensor())
# Train dataset loader
train_loader = torch.utils.data.DataLoader(dataset = train_dataset,
                                           batch_size = args.batch_size,
                                           shuffle = True)
# Test dataset loader
test_loader = torch.utils.data.DataLoader(dataset = test_dataset,
                                          batch_size = args.batch_size,
                                          shuffle = False)
```

Code Snippet 5.5: *Load CIFAR-10 dataset for non-convex problem.*

For each **Python script** we need to define different **elements** and **hyperparameters**, which describe how the model will be trained, as following:

1. model - **Linear regression** for convex problem and **ResNet 20** for non-convex problem;

2. dataset - **MNIST** for convex problem and **CIFAR-10** for non-convex problem;

3. loss function - **Cross Entropy** [26], which measures how performant is a classifier whose output is represented by a probability between 0 and 1. Cross Entropy loss has an increased value if the predicted probability diverges from the correct label of sample;

4. optimizer - **Stochastic Gradient Descent** (presented in sub-section 5.1.1) with/without momentum term, **L-BFGS** (presented in sub-section 5.1.2) which are currently defined in **Pytorch**, but we added the **line search** methods in order to find the best step length;

5. epochs - how many times we go through all training samples in order to obtain a good model;

6. batch size - the number of images that are used for one update step. The total number of update steps per epoch is represented by the ratio between the **total number of images from training set** and **batch size**;

7. learning rate - the length of the update in the case of fixed step size.

After defining all these elements and hyperparameters we do the **forward-propagation** process (Code Snippet 5.6). We will take each mini-batch of data to compute the prediction for each sample and the loss function of the model with respect to the current mini-batch of data. Using the value of the loss function, the parameters of the model are updated with different optimizers (first or second order methods) in the **backward-propagation** process (Code Snippet 5.6). The accuracy for training samples and the value of loss function are printed in a log file (Code Snippet 5.7). After updating the model and finishing one epoch of training, we will measure the performance of the model with respect to the test set (Code Snippet 5.8). Finally, we will save the state of the model (parameters), in a ”.pth” file, in order to load it when we need a classifier for new data (Code Snippet 5.9). In the sections 5.6 and 5.7 we present the experimental results for convex and non-convex problems. For each experiment we provide the elements and hyperparameters used for training the model and the plots with the results for **loss evolution**, **train accuracy**, **test accuracy** and **learning rates**.

```
for epoch in range(args.epochs):
    # Train the model
    model.train()
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        # Define the closure for optimizer
        def closure():
            # Clear gradients, not be accumulated
            optimizer.zero_grad()
            # Forward pass to get output
            outputs = model(images)
            # Calculate loss: softmax + cross entropy loss
            loss = criterion(outputs, labels)
            # Get gradients
            loss.backward()
            # Return loss
            return loss

        # Get lr and loss
        if args.optimizer == ’sgd’ or args.optimizer == ’lbfgs’:
            lr = get_lr(optimizer)
            loss = optimizer.step(closure)
        elif args.optimizer == ’sgd_ls’ or args.optimizer == ’lbfgs_ls’:
            loss, lr = optimizer.step(closure)

        # Compute accuracy for the model
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total = labels.size(0)
        correct = (predicted == labels).sum().item()
        train_accuracy = 100 * correct / total
```

Code Snippet 5.6: *Forward-propagation and backward-propagation processes.*

```
if i == 0 or (i + 1) % args.print_freq == 0:
        print (”Epoch: [{}][{}/{}]_, \
                Loss: {:.4f}_ \
                Accuracy: {:.2f}_ \
                Learning rate: {:.5f}”
               .format(epoch, i, total_step,
               loss.item(), train_accuracy, lr))
```

Code Snippet 5.7: *Print results in log file.*

```
# Test the model on test set
  model.eval()
  with torch.no_grad():
      correct = 0
      total = 0
      for images_test, labels_test in test_loader:
          images_test = images_test.to(device)
          labels_test = labels_test.to(device)
          outputs = model(images_test)
          _, predicted = torch.max(outputs.data, 1)
```

```
            total += labels_test.size(0)
            correct += (predicted == labels_test).sum().item()
        test_accuracy = 100 * correct / total

        print('Test accuracy: {}_'.format(test_accuracy))
```

Code Snippet 5.8: *Print accuracy for test set in log file.*

```
# Save the model checkpoint
if args.save_model:
    torch.save(model.state_dict(), args.net+'.pth')
```

Code Snippet 5.9: *Saving the model's state into ".pth" file.*

5.6 Experimental results - Convex problem

Table 5.1: *Convex problem SGD - all line search methods.*

Model	Linear Regression
Dataset	`MNIST` 60k – 10k
Loss function	Cross Entropy
Optimizer	`SGD` with momentum 0.9
Epochs	30
Batch size	2048
Learning rate SGD	0.001
α_1 for SGD LS	0
α_2 for SGD LS	0.001

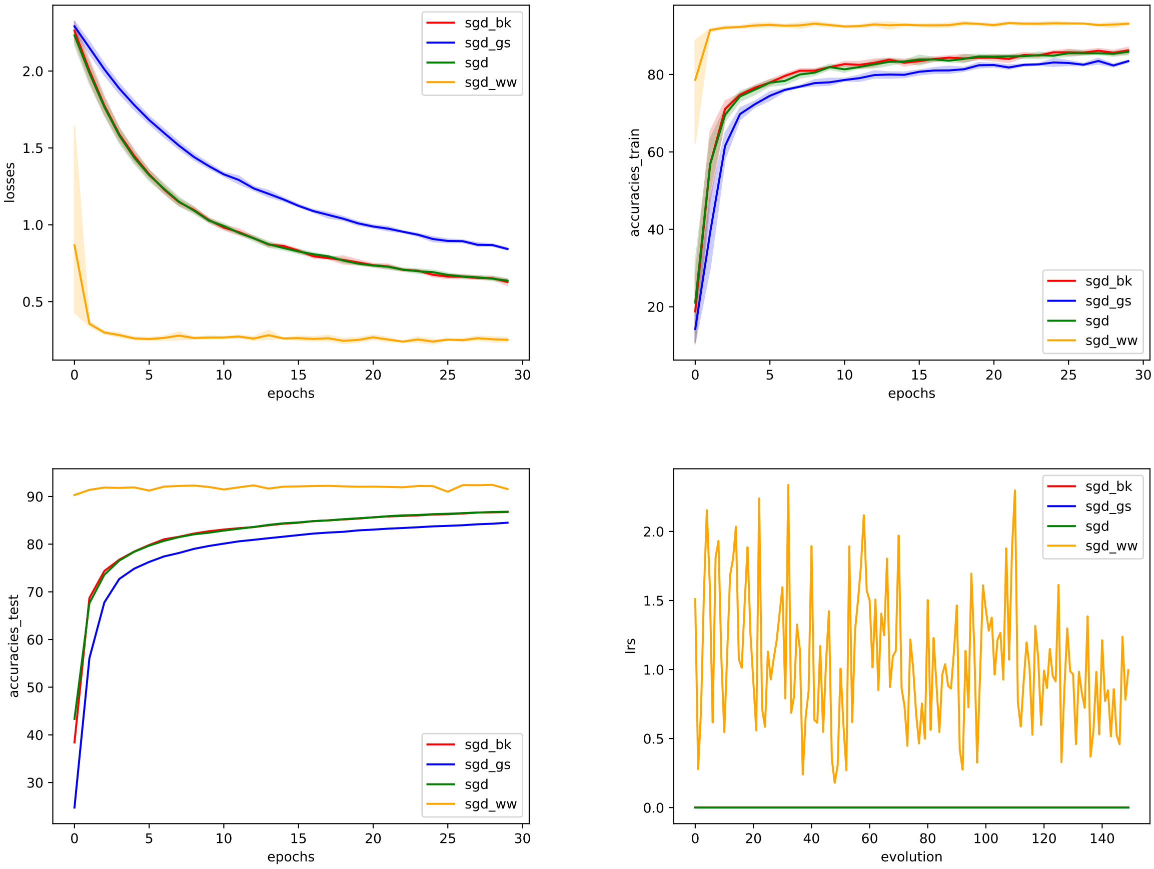

Figure 5.6: *Loss evolution / Train accuracy / Test accuracy / Learning rates.*

Table 5.2: *Convex problem L-BFGS - all line search methods.*

Model	Linear Regression
Dataset	MNIST 60k – 10k
Loss function	Cross Entropy
Optimizer	L-BFGS
Epochs	30
Batch size	2048
Learning rate L-BFGS	0.001
α_1 for L-BFGS LS	0
α_2 for L-BFGS LS	0.001

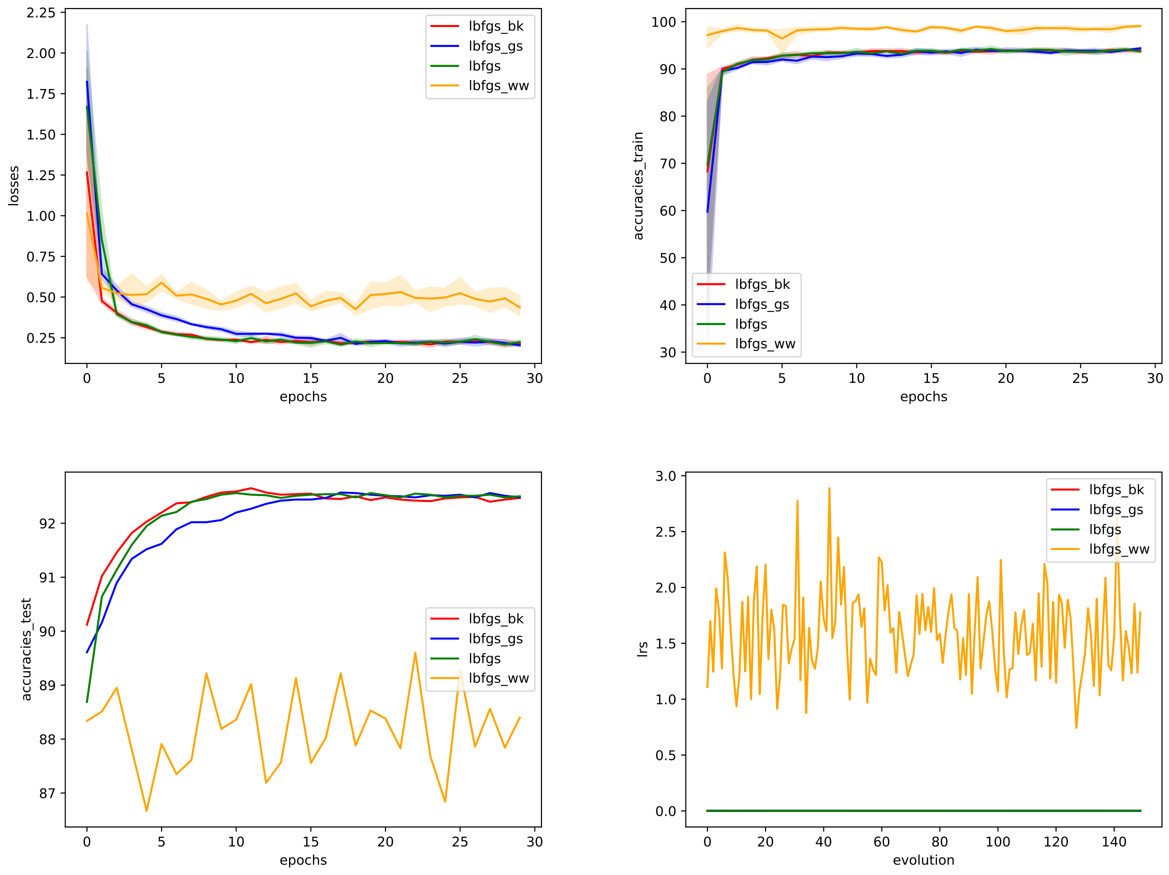

Figure 5.7: *Loss evolution / Train accuracy / Test accuracy / Learning rates.*

Table 5.3: *Convex problem SGD with and without mom vs L-BFGS.*

Model	Linear Regression
Dataset	MNIST 60k – 10k
Loss function	Cross Entropy
Optimizer	L-BFGS
	SGD (mom 0)
	SGD (mom 0.5)
Epochs	30
Batch size	1024
Learning rate 1	0.01
Learning rate 2	0.1

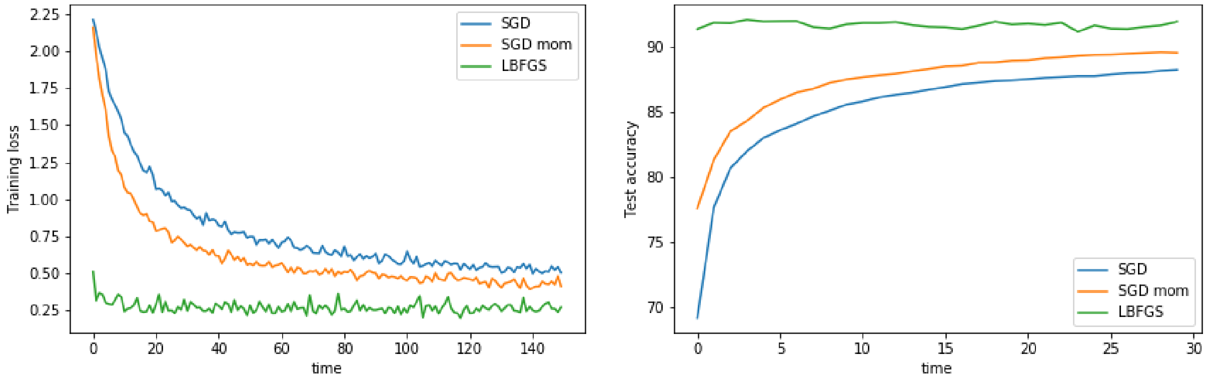

Figure 5.8: *Loss evolution / Test accuracy.*

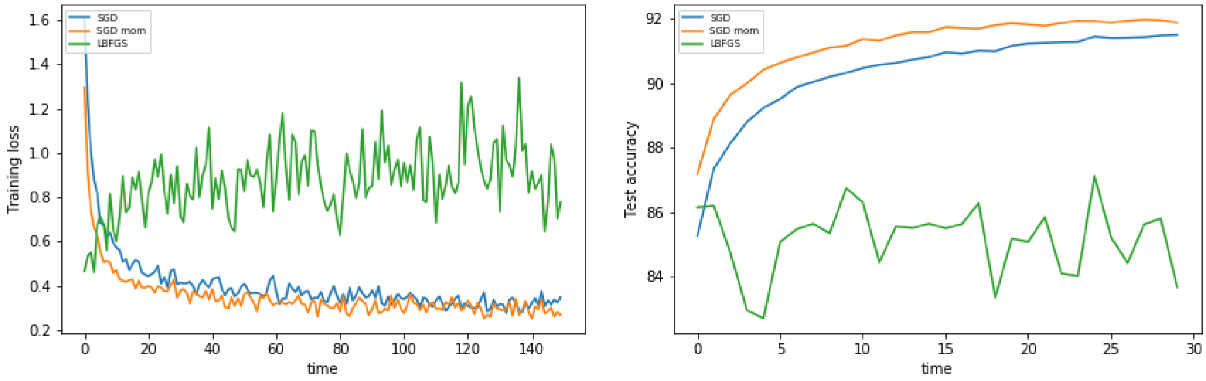

Figure 5.9: *Loss evolution / Test accuracy.*

From the experiment presented in Table 5.1 and the plots presented in Figure 5.6 we can observe that the training process of the convex problem is performing better using **weak wolfe** line search method for determining the learning rate for each step. The loss function is decreasing faster using **weak wolfe** and the accuracy on train set and test set is better than the other methods. The **step length**, also known as learning rate, is changing at each step in comparison with other methods where the learning rate is 0.1 or lower.

From the experiment presented in Table 5.2 and the plots presented in Figure 5.7 we can observe that the training process of the convex problem is performing quite better using **second order** method (L-BFGS) than **first order** method (SGD). In comparison with the experiment with **first order** optimizer, the loss function is decreasing faster using **weak wolfe** and the accuracy on train set is better than the other line search methods. An important observation is that on test set the accuracy is noisy. In general, we want a trained model that performs good on unseen data, property which is known as generalization [27], in order to avoid overfitting [28] (modelling the data too well).

From the experiment presented in Table 5.3 and the plots presented in Figures 5.8, 5.9 we can observe the comparison between the training process of the convex problem using **first order** and **second order** methods. The loss function is decreasing quickly using L-BFGS optimizer than using SGD optimizer, but we need to be careful of overfitting the model. Also, we note that the momentum term [29] helps the acceleration of gradient vectors in the right descent direction and the process of convergence is faster. By comparing plots from Figures 5.8 and 5.9 we can notice an important fact. During the optimization process using a fixed step size of 0.01 we get better results for L-BFGS method than using a fixed step size of 0.1. This is the reason that while training a neural network we need to check different values for hyperparameters (this process is called **grid search** - finding optimal hyperparameters for a model).

During the training of neural networks in the first epochs we need to use a high value for learning rate, because the model starts with randomly selected parameters, which will produce bad predictions for the data. Recall that using a high learning rate the model will make important updates for the parameters. After a number of epochs, the learning rate needs to become smaller because the model should make tiny updates. This process is known as decaying the value of learning rate during the optimization process. In Residual neural network architecture, the learning rate is decaying (by a factor of 10) after 80, 120 and 160 epochs.

5.7 Experimental results - Non-convex problem

Table 5.4: *Non-convex problem SGD - all line search methods.*

Model	ResNet 20
Dataset	`CIFAR-10` 50k – 10k
Loss function	Cross Entropy
Optimizer	`SGD` with momentum 0.9
Epochs	100
Batch size	2048
Learning rate SGD	0.1 / 0.01 (ep. 80)
α_1 for SGD LS	0
α_2 for SGD LS	0.1

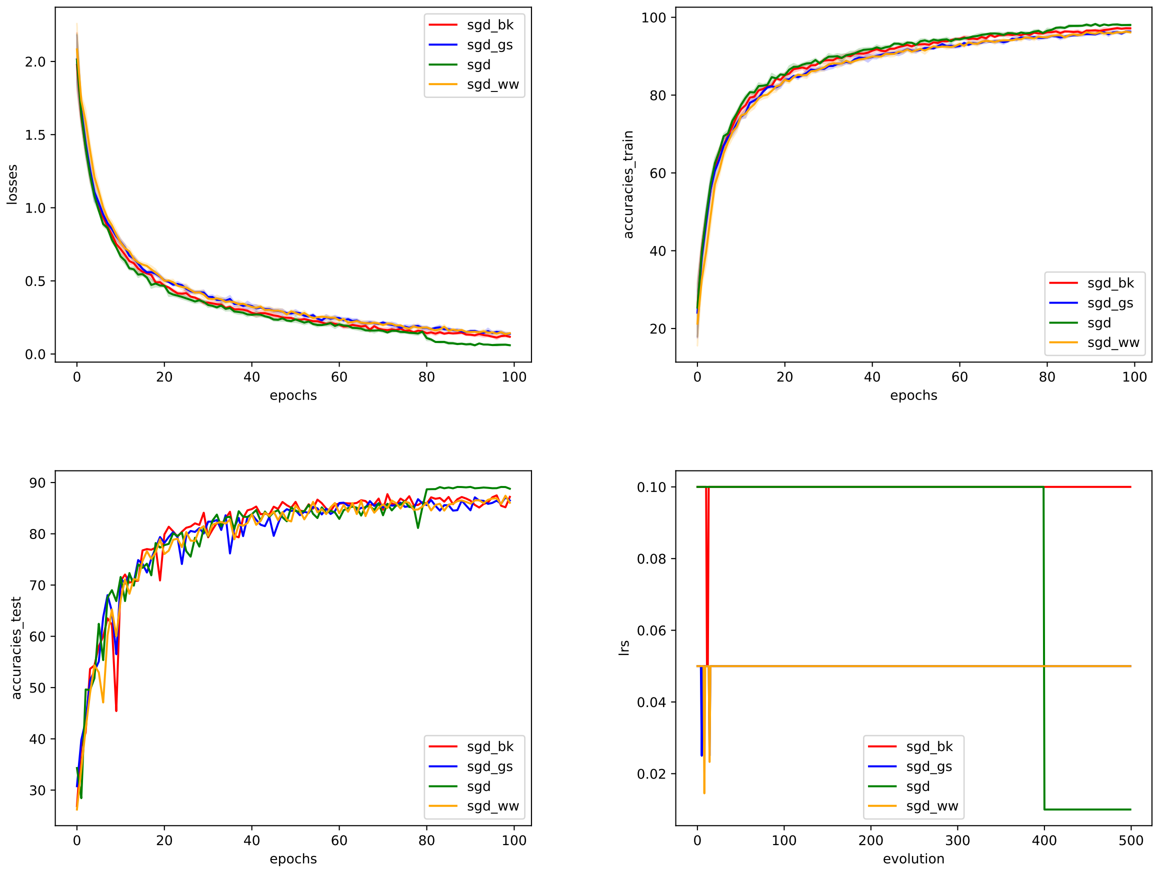

Figure 5.10: *Loss evolution / Train accuracy / Test accuracy / Learning rates.*

Table 5.5: *Non-convex problem L-BFGS - all line search methods.*

Model	ResNet 20
Dataset	`CIFAR-10` 50k – 10k
Loss function	Cross Entropy
Optimizer	`L-BFGS`
Epochs	30
Batch size	2048
Learning rate L-BFGS	0.1
α_1 for L-BFGS LS	0
α_2 for L-BFGS LS	0.1

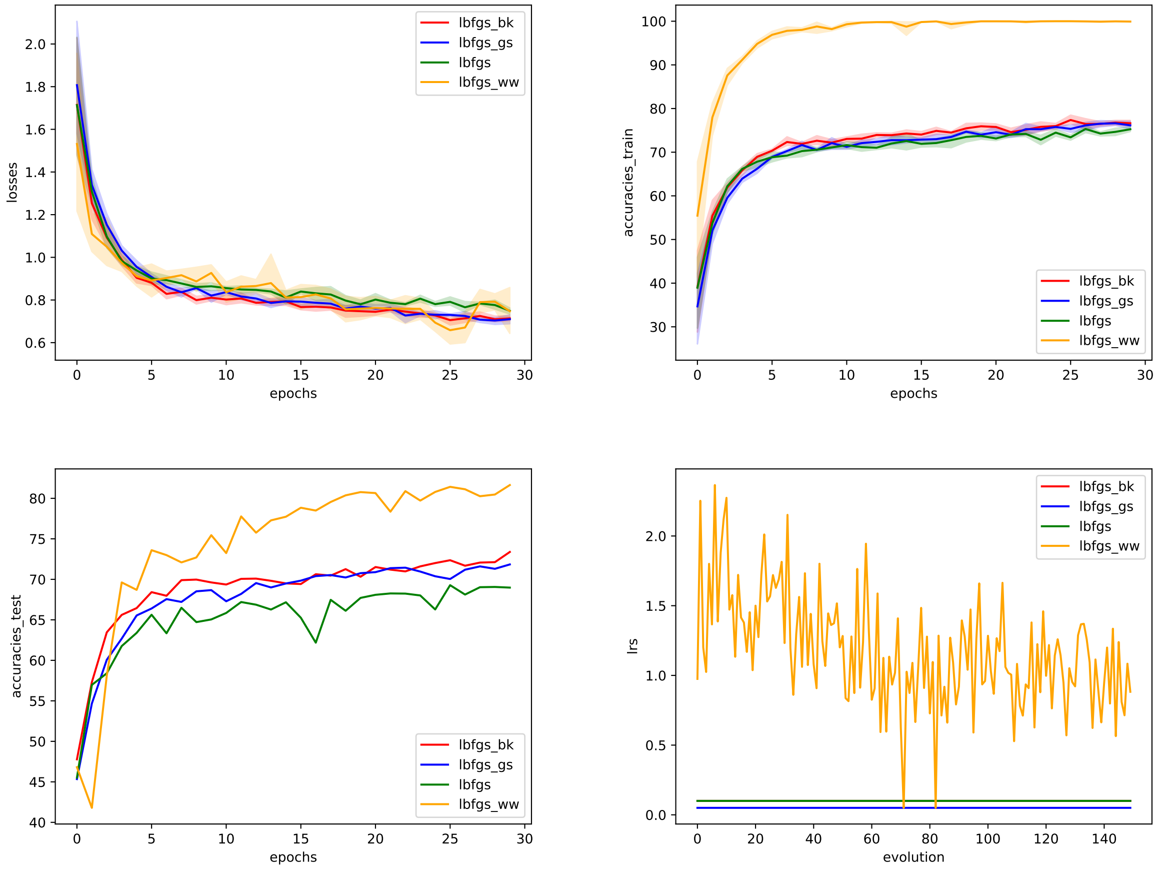

Figure 5.11: *Loss evolution / Train accuracy / Test accuracy / Learning rates.*

Table 5.6: *Non-convex problem L-BFGS - gs and ww line search methods.*

Model	ResNet 20
Dataset	`CIFAR-10` 50k – 10k
Loss function	Cross Entropy
Optimizer	`L-BFGS`
Epochs	30
Batch size	2048
Learning rate L-BFGS	-
α_1 for L-BFGS LS	0
α_2 for L-BFGS LS	∞

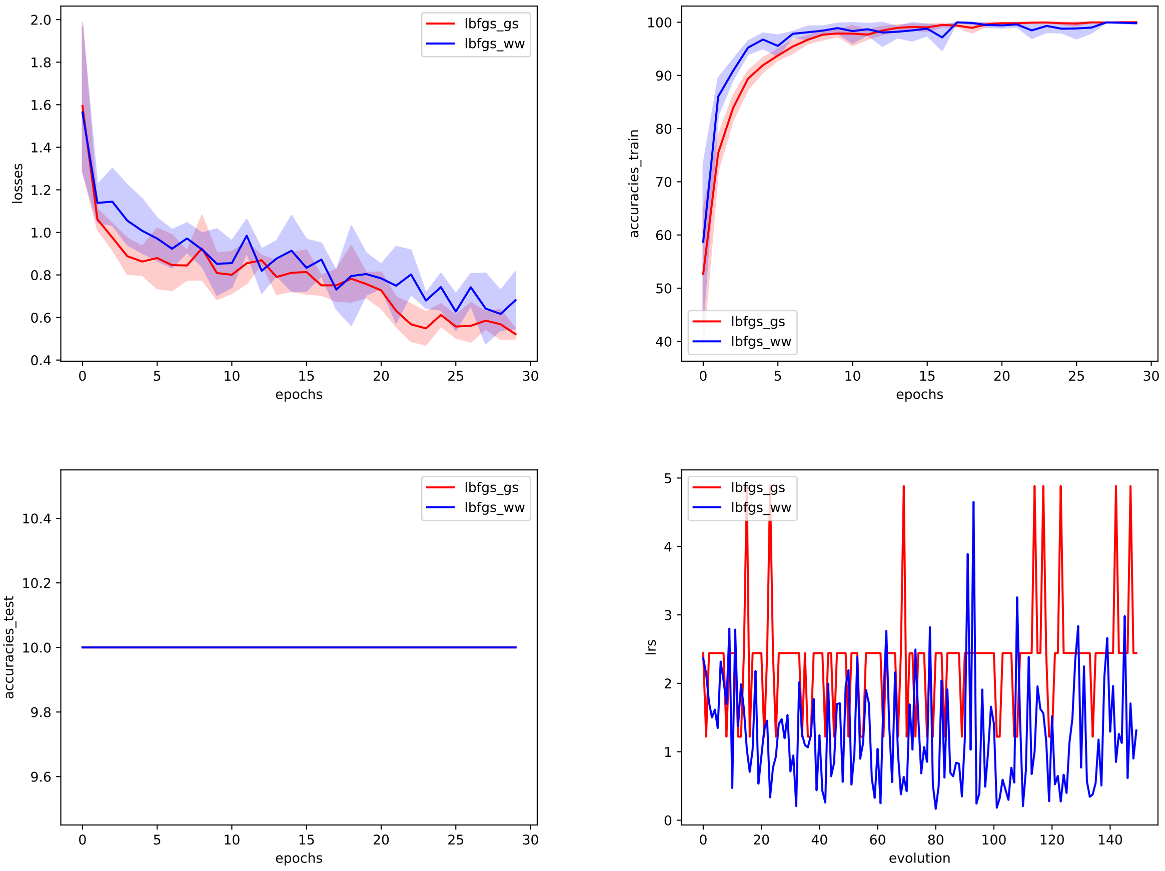

Figure 5.12: *Loss evolution / Train accuracy / Test accuracy / Learning rates.*

From the experiment presented in Table 5.4 and the plots presented in Figure 5.10 we can observe that the training process of the non-convex problem using SGD optimizer is performing quite similar using either fixed step length for the update or line search methods for step length. The loss function is decreasing for all methods, but SGD using fixed learning rate which is decaying after 80 epochs seems to be a better option.

From the experiment presented in Table 5.5 and the plots presented in Figure 5.11 we can observe that the training process of the non-convex problem using L-BFGS optimizer is performing better when the step size is chosen with line search methods. The accuracy of the model is reaching 100% with weak wolfe method on training set, but the accuracy on test set is not performing good, so the neural network does not generalize well. Also, the value of learning rate using weak wolfe method is changing at every update step because the conditions are stricter than the conditions of other line search methods.

From the experiment presented in Table 5.6 and the plots presented in Figure 5.12 we can observe an interesting behaviour of the neural network. The model is converging to 100% accuracy on training set, but it is not able to generalize, giving 10% accuracy on test set. We can conclude that using $\alpha_1 = 0$ and $\alpha_2 = \infty$ for goldstein and weak wolfe methods the model learns the data too well, resulting an overfitted neural network. For future work, we can implement some methods for selecting optimal values for α_1 and α_2.

5.8 Technology

During the development process we used different technologies, mainly specialized in building and training neural networks and special architectures of deep neural networks, like Resnet, VGG, AlexNet, LeNet and many others.

Python [30] is a high-level programming language, where the code is interpreted and object-oriented. It has a simple syntax reducing the cost of maintenance. The library of modules and packages is used to build modular programs and to reuse the code. In order to ease the development process we used **Pytorch** [31], a specialized machine learning framework, built on the top of **Torch** library. Multidimensional arrays which represent the structure of a neural network are called Tensors. Using this framework the computational operations (e.g. addition, multiplication) of Tensors can be done on GPU that supports CUDA. Pytorch framework has some important modules for automatic differentiation of functions (**autograd**), for optimization algorithms (**optim**), for defining neural networks using computational graphs (**nn**). Different deep neural network architectures are defined in **torchvision** module, in **torchvision.models** package. Also, in order to use some existing datasets (e.g. MNIST, CIFAR-10), **torchvision.datasets** package is available. Our GPU-based experiments were run on a compute cluster with GeForce RTX 2080 GPUs (supported by ERC Grant ScaleML 805223). In order to plot the results we used **Matplotlib** [32], which is a Python 2D plotting module used to produce quality figures like plots, histograms, bar charts.

Chapter 6

Conclusions and future work

During the training of neural networks an important point is hyperparameter optimization, in order to build a model that performs well. Line search methods are specialized algorithms that try to find an optimal learning rate for each update step using a directional vector. We need to take care of the generalization property of the model, because it needs to perform well not only on training data, but also on unseen data. There are some open questions for our future work:

- do line search strategies help with SGD and L-BFGS on large batch size?
- do the MNIST results from the previous experiments (convex problem) generalize to other architectures (e.g. AlexNet, VGG, LeNet) and datasets (CIFAR-10/100, ImageNet)?
- how large is the update of the model from an epoch to another (implement a progress measure based on the gradients)?
- does early stopping strategy helps the training with L-BFGS optimizer in order to avoid overfitting?
- do other second order methods (e.g. SQN [33], Neumann [34]) produce similar results as L-BFGS?
- how do we get better results using Residual neural network architecture?
- how can we parallelize line search methods in order to perform faster?

The computational time necessarily for line search methods is much higher than using fixed step length during the optimization process, because we need to check the performance of the model, by validating the conditions, for every value of α_k. The key point is that L-BFGS method is working much better using weak wolfe method than using a fixed learning rate (from the above plots we can conclude that second order methods are using a different value of learning rate for every update step, building a better model). For SGD method we can build a good neural network that performs well using a fixed step length, by running multiple times the training process (grid search).

In ”Numerical Optimization”, Nocedal and Wright present a line search algorithm with Wolfe conditions using interpolation methods (e.g. cubic, quadratic, bisection) for selecting the value of α_k between some bound values α_1 and α_2. In their work they conclude that Wolfe conditions are the most broadly applicable conditions. In our experimental results we prove that Wolfe conditions perform better than other line search methods, but we need to be careful that our model is able to generalize. Also, an important observation is that a line search algorithm must include a stopping criteria in case that it cannot find a lower function value after a given number of steps (typically, ten). Ortega and Rheinboldt in their work [35], present an extensive discussion about the termination conditions of line search methods, which can be used in our implementation. Also, some line search methods (see Goldfarb [36]; More and Sorensen [37]) select the direction of negative curvature, in order to avoid non-minimizing points that are stationary. In our future work we should try these methods (optimization step will combine the direction of negative curvature with steepest descent direction).

Bibliography

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 3rd edition, 2009.

[2] Deep learning for image recognition: why it's challenging, where we've been, and what's next. `https://towardsdatascience.com/`.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[4] Tariq Rashid. *Make Your Own Neural Network*. CreateSpace Independent Publishing Platform, 2016.

[5] Facundo Bre, Juan Gimenez, and Víctor D. Fachinotti. Prediction of wind pressure coefficients on building surfaces using artificial neural networks. *Energy and Buildings*, 158, 11 2017.

[6] Learning process of a neural network. `https://towardsdatascience.com/how-do-artificial-neural-networks-learn-773e46399fc7`.

[7] Tensorflow. `https://www.tensorflow.org`.

[8] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[9] Non-convex optimization in machine learning. `https://cw.fel.cvut.cz/b181/_media/courses/xp36vpd/janmrkos-non-convex_optimization_in_ml.pdf`.

[10] Prateek Jain and Purushottam Kar. *Non-convex Optimization for Machine Learning*. IEEE, 2017.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, page 2012.

[12] A comprehensive guide to convolutional neural networks. https://towardsdatascience.com.

[13] Roberto Battiti. First- and second-order methods for learning: Between steepest descent and newton's method. *Neural Computation*, 4, 03 1992.

[14] *CS231N - Convolutional NN for Visual Recognition*. Stanford University, 2019. http://cs231n.github.io/.

[15] Stochastic vs batch gradient descent. https://medium.com/@divakar_239/stochastic-vs-batch-gradient-descent-8820568eada1.

[16] R. H. Byrd, S. L. Hansen, Jorge Nocedal, and Y. Singer. A stochastic quasi-newton method for large-scale optimization. *SIAM Journal on Optimization*, 26(2):1008–1031, 2016.

[17] Yu-Hong Dai. A perfect example for the bfgs method. *Mathematical Programming*, 138(1):501–530, Apr 2013.

[18] Setting the learning rate of your neural network. https://www.jeremyjordan.me/nn-learning-rate/.

[19] Wenyu Sun and Ya-Xiang Yuan. *Optimization Theory and Methods Nonlinear Programming*. Springer, 2016.

[20] Line search methods. https://optimization.mccormick.northwestern.edu/index.php/Line_search_methods.

[21] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[23] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[24] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[25] Linear class. https://pytorch.org/docs/stable/_modules/torch/nn/modules/linear.html.

[26] Zhilu Zhang and Mert R. Sabuncu. Generalized cross entropy loss for training deep neural networks with noisy labels. *CoRR*, abs/1805.07836, 2018.

[27] M. Vidyasagar. *A Theory of Learning and Generalization: With Applications to Neural Networks and Control Systems*. Springer-Verlag, 1997.

[28] Shaeke Salman and Xiuwen Liu. Overfitting mechanism and avoidance in deep neural networks. *CoRR*, abs/1901.06566, 2019.

[29] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1139–III–1147. JMLR.org, 2013.

[30] Python. `https://www.python.org/doc/`.

[31] Pytorch. `https://pytorch.org/docs/stable/index.html`.

[32] Matplotlib. `https://matplotlib.org/`.

[33] R H. Byrd, S L. Hansen, Jorge Nocedal, and Y Singer. A stochastic quasi-newton method for large-scale optimization. *SIAM Journal on Optimization*, 26, 2014.

[34] Shankar Krishnan, Ying Xiao, and Rif A. Saurous. Neumann optimizer: A practical optimization algorithm for deep neural networks. 2017.

[35] James M. Ortega and Werner C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[36] Donald Goldfarb. Curvilinear path steplength algorithms for minimization which use directions of negative curvature. *Mathematical Programming*, 18(1):31–40, 1980.

[37] Jorge J. More and Danny C. Sorensen. On the use of directions of negative curvature in a modified newton method. *Mathematical Programming*, 16(1):1–20, 1979.