UNIVERSITÀ DI PISA

Optimization for Data Science

Andrea Frasson
Jacopo Gneri

March 2024

Contents

Chapter 1

Introduction

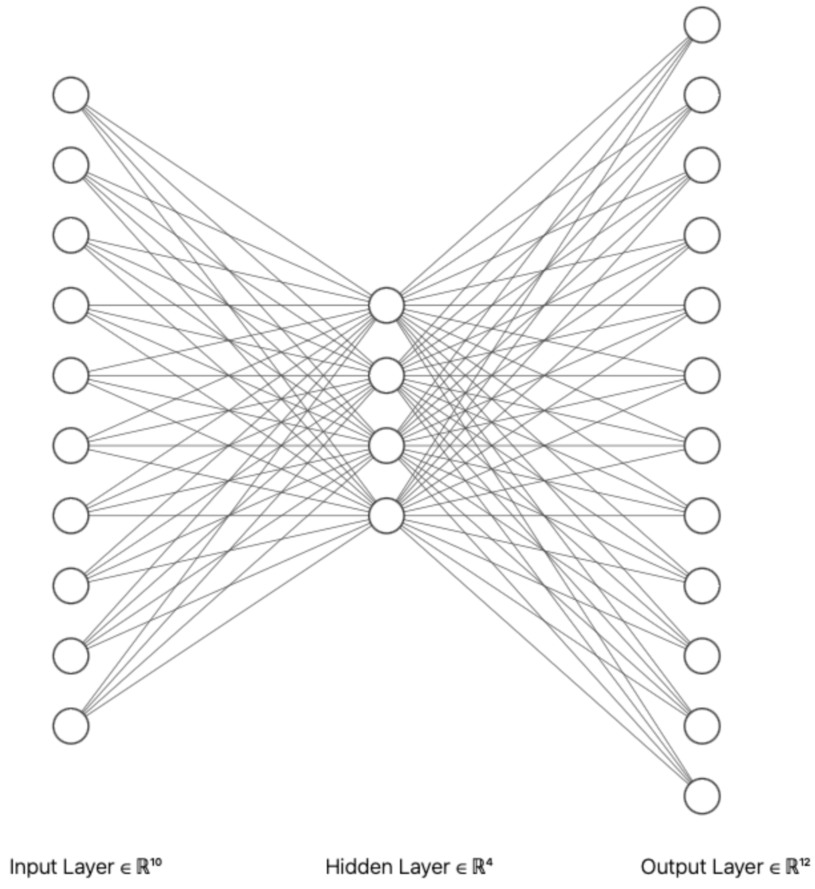
Neural Networks and their variants are powerful models that achieve high performance on difficult problems, like pattern recognition problems in vision and speech.
Modern deep-learning algorithms provide very powerful frameworks for supervised learning. By adding layers and more units within each layer, a deep network can represent functions of increasing complexity. In general, a neural network defines a mapping function $\mathbf{y} = f(\mathbf{x}; \theta)$ and learns the value of the parameters θ that results in the best approximation. Due to the nonlinearity of neural networks, iterative optimizers, particularly the stochastic gradient descent (SGD), are preferred over linear solvers for training the models. The problem is that SGD applied to non-convex loss functions has no convergence guarantee and is sensitive to the values of the initial parameters.
In this project we study two different algorithms: the classical momentum descent (HB), or heavy ball, approach and the deflected subgradient approach (DS). We apply these algorithms to a basic neural network to understand their behavior.
In the first section we define the structure of the neural network used for the implementation of the algorithms and the data adopted. Additionally, we present our target function. In the second section, we present the algorithms and their theoretical properties. The last section describes the experiments and shows the results.
Since the focus of the project is on optimization, we only report the training errors obtained in the experiments. It's worth noting that test error is heavily influenced by overfitting in these scenarios. While our cost function incorporates a regularization term, its importance is marginal in this study.

1.1 Dataset

The task required a multi-class classification dataset, so we decided to use the 'Avila Dataset', which has been extracted from 800 images of the 'Avila Bible', a XII century giant Latin copy of the Bible [9]. The prediction task consists of associating each pattern to one of the 12 possible copyists. The data was divided into two data sets: a training set containing 8344 samples and a test set containing 2086 samples. In this study, we decided to take into consideration only the training set, which still guarantees a discrete number of samples. We will call $\mathbf{X}$ the features in the training set and $\mathbf{y}$ the labels. Before the learning phase, the data were standardized, using the MinMax Scaler, so the final features contain values inside the interval $[0, 1]$.

1.2 Network Structure

The dataset was intentionally kept small since the final goal of this analysis is the implementation and study of different optimization algorithms. Obviously, if the dimension of the data becomes extremely high then more problems, like sparsity, may arise, which is something that in real situations needs to be taken into consideration.
Once the data were ready, the structure of the network needed to be formally declared. In this section we will introduce some of the notation regarding the neural network. As we said, in general, a neural network defines a mapping function, $\hat{\mathbf{y}} = g(\mathbf{x}; \theta)$. To maintain continuity, $\mathbf{x}$ will be a 10-dimensional input vector, $\hat{\mathbf{y}}$ will be the output obtained and θ will represent all parameters in the network. Later, for clarifying some of the passages, the notation of the weights and the biases will be complicated, for the general results we will try to keep the notation light. While in machine learning we would try a variety of different models to find the one that appears to generalize better, the objective of our project is analyzing the behavior of different optimization algorithms and therefore we will set a specific architecture for the network and investigate the resulting function.



Input Layer $\in \mathbb{R}^{10}$
Hidden Layer $\in \mathbb{R}^{4}$
Output Layer $\in \mathbb{R}^{12}$

Figure 1.1: Neural Network

As shown in Figure 1.1, the neural network is structured in 3 layers. An input vector $\mathbf{x}$ is given in the input layer, where it is linearly transformed using the weights of the hidden layer, a 10×4 matrix called $\mathbf{W}^1$, and the relative biases, $\mathbf{b}^1$, a 4-dimensional vector.

$$\mathbf{Z}^1 = \mathbf{x}^T \mathbf{W}^1 + \mathbf{b}^1 \tag{1.1}$$

This is the unactivated value for the first hidden layer, for the activation function we chose *tanh*, the hyperbolic tangent activation function. So passing Z^1 in the activation function, we obtain the activated vector for the first hidden layer, which will be used as the input vector for the output layer.

$$\mathbf{A}^1 = tanh(\mathbf{Z}^1) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{1.2}$$

Again, the input vector is linearly combined with the matrix of weights and vector of biases, then the vector is activated using once again the *tanh* activation function.

$$\begin{gathered} \mathbf{Z}^2 = \mathbf{A}^1 \mathbf{W}^2 + \mathbf{b}^2 \\ \hat{\mathbf{y}} = tanh(\mathbf{Z}^2) \end{gathered} \tag{1.3}$$

The activated vector for the output layer is the predictions of the model, $\hat{\mathbf{y}}$. In the implemented algorithm, the neural network performs a batch learning phase, so all the training samples are used to obtain the predictions at once. The process used does not change the formulas, but of course, the predictions, the activated and unactivated values, which were stated as vectors, are matrices.
At the end of the forward propagation phase, as we said, the model returns a matrix $n \times 12$ of predictions, $\hat{\mathbf{y}}$, where n is the number of training samples and all the values are inside the interval $[0, 1]$. In this matrix, the i^{th} row represents the prediction for the i^{th} observation in the training set.
At this point the loss is computed by using the Mean Squared Error (MSE). We obtain

$$J(\theta) = \frac{1}{2n} ||\hat{\mathbf{y}} - \mathbf{y}||_2^2 \tag{1.4}$$

A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs. Many strategies used in machine learning are explicitly designed to reduce test error, possibly at the expense of increased training errors. To generalize well the model we used the L1 regularization. Regularizing the bias parameters can introduce a significant amount of underfitting [7], so we decided to regularize only the weights. We will use $\mathbf{w}$ to indicate only the weights in the network, and $\mathbf{b}$ for the biases.
Formally, L1 regularization on the model parameter $\mathbf{w}$ is defined as the sum of the absolute values of the elements. L1 weight decay controls the strength of the regularization by scaling the penalty using a positive hyperparameter λ. The final loss function is given by

$$\tilde{J}(\theta) = J(\theta) + \lambda ||\mathbf{w}||_1 \tag{1.5}$$

With the hyper-parameter λ that regulates the strength of the regularization.
In general, the choice of the cost function is extremely important, the purpose of loss

functions is to compute the quantity that a model should seek to minimize during training. With the computation of the cost function, we can now explicitly define the optimization problem through a set of variables and an objective function. The goal, in this case, is to find the values for the variables that lead to the minimum value of the function

$$\min_{\mathbf{w},\mathbf{b}} f(\mathbf{w},\mathbf{b}) = \min_{\mathbf{w},\mathbf{b}} \tilde{J}(\mathbf{w},\mathbf{b}) = \min_{\mathbf{w},\mathbf{b}} \frac{1}{2n}||g(\mathbf{w},\mathbf{b},\mathbf{x}) - \mathbf{y}||_2^2 + \lambda||\mathbf{w}||_1 \tag{1.6}$$

Then we implemented a back-propagation algorithm that allows the information from the cost to flow backward through the network, to compute the gradient. In learning algorithms, the gradient we most often require is the gradient of the cost function concerning the parameters. To achieve this goal, we used the chain rule to compute the derivatives of functions formed by composing other functions whose derivatives are known.
Starting from the output layer, the first things we need to compute are the gradient concerning W^2 and b^2, the weights and the biases for this level.

$$\begin{aligned} \frac{\partial \tilde{J}(\theta)}{\partial W^2} &= \frac{\partial \tilde{J}(\theta)}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial Z^2} \frac{\partial Z^2}{\partial W^2} + \lambda \frac{\partial ||\mathbf{W}||_1}{\partial W^2} \\ \frac{\partial \tilde{J}(\theta)}{\partial b^2} &= \frac{\partial \tilde{J}(\theta)}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial Z^2} \frac{\partial Z^2}{\partial b^2} \end{aligned} \tag{1.7}$$

Since we decided to regularize only the weights, the derivative of the regularization term appears only in the weights expressions.
At this point, we simply apply the chain rule as we proceed backward through the layers of the network.
Again, we incorporated the first two terms in (1.7), calling them as $\frac{\partial \tilde{J}(\theta)}{\partial Z^2}$, to improve readability. At last, for the first hidden layer, we obtain

$$\begin{aligned} \frac{\partial \tilde{J}(\theta)}{\partial W^1} &= \frac{\partial \tilde{J}(\theta)}{\partial Z^2} \frac{\partial Z^2}{\partial A^1} \frac{\partial A^1}{\partial Z^1} \frac{\partial Z^1}{\partial W^1} + \lambda \frac{\partial ||\mathbf{W}||_1}{\partial W^1} \\ \frac{\partial \tilde{J}(\theta)}{\partial b^1} &= \frac{\partial \tilde{J}(\theta)}{\partial Z^2} \frac{\partial Z^2}{\partial A^1} \frac{\partial A^1}{\partial Z^1} \frac{\partial Z^1}{\partial b^1} \end{aligned} \tag{1.8}$$

At this point, the back-propagation scheme is completed, since the gradient for each parameter is obtained. In the most general case, at this point, the learning process is built, and implementing a gradient descent algorithm allows the neural network to adapt and discover the best model for the data.
Now that the structure of the network is declared, the following sections present the algorithms that we will study, their theoretical properties, and if they are verified in our function.

Chapter 2

Algorithms

In this section, we describe the implemented algorithms from a theoretical perspective. Before delving into the required methods, we briefly examine the classical gradient approach adopted in machine learning. Let θ be the parameter matrix of our neural network, representing some function $g(\theta)$. The goal is to minimize a cost function $C(g, D)$ where D is the set of available data representing the problem's true data distribution. With gradient descent, we take small steps on the cost function's surface using information from the current location. The step is an update of the parameter matrix θ defined as

$$\theta_i = \theta_{i-1} - \alpha \nabla C(\theta_{i-1}) \tag{2.1}$$

Where α denotes the step size along the gradient direction, in machine learning this is typically referred to as learning rate. Various methods exist to determine the step size, ranging from fixed values to dynamic searches like line search. The method assumes the cost function is differentiable, with established functions like Mean Squared Error and Cross Entropy commonly used. However, basic gradient descent can converge slowly, leading to interest in improving and speeding up the method while maintaining basic assumptions. Although optimization theory provides guarantees for convex functions, machine learning problems often don't fit this framework, posing challenges for theoretical guarantees. Subsequent sections will explore two methods that try to improve basic gradient descent.

2.1 Heavy Ball Method

The Heavy Ball (HB) approach, or classical momentum, is a powerful technique for solving optimization problems. It is an iterative method that uses the concept of momentum to accelerate the convergence of the solution to the global optimum. This approach has been widely used in machine learning, artificial intelligence, and data mining applications.
Given a function $f(x)$, referred to as the objective function, the classical momentum schema is given by:

$$x_{i+1} = x_i - \alpha_i \nabla f(x_i) + \beta_i (x_i - x_{i-1}) \tag{2.2}$$

In the neural network context, the HB method is commonly written as

$$\begin{aligned} d_{i+1} &= \mu d_i - \alpha_i \nabla f(x_i) \\ x_{i+1} &= x_i + d_{i+1} \end{aligned} \tag{2.3}$$

The equivalence proof between the expressions can be found at [A.1].
Here, α denotes the learning rate and μ is a single momentum parameter for all model variables. Adding the scaled previous step, the algorithm reduces the overall greediness of the SGD by taking into account the exponentially weighted average of the previous directions. When $\mu = 0$, HB reduces to the standard gradient descent algorithm. Momentum's main appeal is its established ability to accelerate convergence, more concrete results can be shown when we consider the convex quadratic setting where the function to optimize is of the form $f(x) = x^T Qx + qx$.
Given an unconstrained optimization problem,

$$\min_{x \in X} f(x) \tag{2.4}$$

With f strongly τ-convex and L-smooth with condition number $\kappa = L/\tau$, the optimal convergence rate of gradient descent without momentum is $O(\kappa - 1/\kappa + 1)$.
Fixing appropriate values for $\alpha, \mu > 0$, the HB algorithm converges to the solution of the problem (2.4) faster than the trajectories of the gradient descent method [3]. Moreover, under these assumptions, the convergence rate is a lot better than the gradient.

Theorem 1. *For f strongly τ-convex and L-smooth, with the optimum choice of parameters the Heavy Ball method converges as*

$$O\left(\frac{\sqrt{L} - \sqrt{\tau}}{\sqrt{L} + \sqrt{\tau}}\right) \tag{2.5}$$

Polyak used a local analysis to derive optimal step-size parameters and to show this convergence. [1]
Although the algorithm doesn't guarantee that the directions taken are descent directions, meaning that in theory, x_{i+1} can be worse than x_i, still with the optimum choice of α and μ, the algorithm converges with optimal rate.
The main assumptions made in the theorem imply that the function has to be strong τ-convex and L-smooth to reach the optimal rate.
Starting from the convexity of the function, the assumptions required it to be τ-convex. Recalling that a function is convex if it satisfies $f \in C^2$, so continuous differentiable two times, and its Hessian matrix $\nabla^2 f(x) \geq 0$ for all the x in the domain. A function is said strongly τ-convex function (or strongly convex modulus τ) with $\tau > 0$ on a set S, if the function is convex on S and the following inequality holds for all x, y $\in$ S and all α in the closed interval [0,1]:

$$\alpha f(x) + (1 - \alpha) f(y) \geq f(\alpha x + (1 - \alpha) y) + \frac{\tau}{2} \alpha (1 - \alpha) ||y - x||^2 \tag{2.6}$$

The modulus tau controls the "strength" of the convexity. A larger value of tau implies stronger convexity, while a smaller value of tau implies weaker convexity.
However, in our case, the use of non-linear and non-convex activation functions in the neural network, precisely the *tanh*, introduces non-linearities into the output, which makes the resulting objective function non-convex. It can be seen, although we do not do it explicitly here, that the hessian of our problem is not positive semi-definite.
Proving that the neural network is not convex, we prove that it is not strongly convex.

For the L1 norm, to show its convexity we compute the second derivative of $f(x) = |x|$, which is 0 for all the domain of the function. Since it is greater or equal than 0, then the absolute value is convex. The combination of convex and non convex function is a non convex function.
Recalling that a differentiable function $f : R^n \to R^m$ is L-smooth with respect to a norm $||.||$ if, for any $x, z \in \textbf{int dom}(f)$,

$$||\nabla f(x) - \nabla f(z)|| \leq L||x - z|| \tag{2.7}$$

We decided to study the objective function one piece at a time. Starting from the MSE, it can be written as

$$f(x) = \frac{1}{2n}||x - y||_2^2 \tag{2.8}$$

The resulting gradient is

$$\nabla f(x) = \frac{1}{n}(x - y) \tag{2.9}$$

Then, we rewrite the inequality in (2.7), for $x, z \in R^n$

$$\begin{aligned} ||\nabla f(x) - \nabla f(z)|| &= ||\frac{1}{n}(x - y) - \frac{1}{n}(z - y)|| = \\ &= \frac{1}{n}||x - z|| \leq L||x - z|| \end{aligned} \tag{2.10}$$

We can state that the function is L-smooth, for all L $\geq 1/n$.
For the L1 norm, which is the sum of the absolute values of its components, the absolute value function is not differentiable at 0. Therefore, the gradient is not defined at points where one or more vector components are zero. This non-differentiability makes it impossible to find a Lipschitz constant L for the gradient. The result is that the L1 norm is not L-smooth.
Turning our attention to the case of interest, the non-convex one, we mention the convergence analysis done in [6] where the proposed iPiano algorithm can reach an optimal value for the problem:

$$\min_{x \in R^n} f(x) = h(x) + g(x) \tag{2.11}$$

where h is a L-smooth possibly non-convex function and g is convex possibly non-smooth. In particular, the algorithm uses the update scheme (2.12)

$$x_{i+1} = (I + \alpha \partial g)^{-1}(x_i - \alpha_i \nabla f(x_i) + \beta_i(x_i - x_{i-1})) \tag{2.12}$$

In this schema we have two distinct parts, $(I + \alpha \partial g)^{-1}$ is referred as the *proximal step* while the rest is the usual HB schema.
When $g \equiv 0$, the proximal step is the identity and the update scheme is exactly the Heavy-ball method.
In section 4 of [8], the authors proved the global convergence for iPiano algorithm, for the class of problems defined before, using three assumptions:

- The function $h : U \subset R^n \to R$ is assumed to be C^1-smooth (possibly non convex) with L-Lipschitz continuous gradient, L>0.

- The function $g : U \subset R^n \to R$ is proper lower semi-continuous, possibly non-smooth, simple and prox-bounded.
- The function f restricted to U is coercive and bounded from below by $\underline{f} > -\infty$.

In Corollary 4.1, they prove the global convergence for iPiano, and they state that the results apply to all the versions of the algorithm. In particular, they provide ranges for the parameters in different scenarios. For the HB method they set

$$\alpha \in (0, \tfrac{2(1-\beta)}{L}) \quad \beta \in [0, 1)$$

In our case it's easy to find a relation, our h is the MSE and our g is the L1 regularization. Starting from the MSE, we already demonstrated that it is L-smooth for all L $\geq 1/n$.
Recalling that a function f is *proper lower semi-continuous* at a point $x_0 \in X$ if and only if

$$\liminf_{x \to x_0} f(x) \geq f(x_0)$$

A function is proper lower semi-continuous if it is semi-continuous in all points. The L1 regularization involves absolute values, which are continuous functions. Since continuity implies lower semi-continuity, we can conclude that L1 regularization is lower semi-continuous.
Lastly, in our setting, it's easy to see that the MSE is always ≥ 0 and the same is true for the L1 regularization, so even if we assume $\underline{f}(x) = 0$, the property is satisfied.
With that, we proved that in our case the convergence assumptions are verified.
In their work [6], the convergence rate for the iPiano algorithm can be defined as

Theorem 2. *Given h, g, and f as before. If f satisfies the KL property, the proposed algorithm has a convergence rate for the squared l2 norm of the error*

$$O\left(\frac{1}{k}\right) \tag{2.13}$$

Where the KL property is the Kurdyka-Łojasiewicz property.

2.2 Deflected Subgradient Method

Subgradient methods (SM) are iterative methods for solving convex minimization problems born in the 60s, these algorithms are the extension of the gradient method. These methods have long been preferred for solving non-differentiable optimization problems. To declare the notation, g is called subgradient of a convex function $f : R^n \to R$ at $x \in R^n$ if

$$f(z) \geq f(x) + < g, z - x > \forall z \in Dom(f) \tag{2.14}$$

In practice, g defines a lower approximation of f. However, in a point, there can exist many subgradients, even if the point is a local or global minimum. For non-differentiable convex function, $\partial f(x)$ is called subdifferential and is the set of all the possible subgradients in x

$$\partial f(x) = \{g \in R^n : g \text{ is a subgradient in } x\} \tag{2.15}$$

A subdifferential is a generalization of the gradient. If f is differentiable at x, then $\partial f(x) = \{\nabla f(x)\}$, and vice-versa.
Given an unconstrained optimization problem,

$$f(x) \to \min_{x \in X} \tag{2.16}$$

The SM employs a simple recurrence formula

$$x_{i+1} = x_i - v_i g_i \tag{2.17}$$

With v_i an appropriate stepsize and g_i a subgradient of f in x_i.
Unlike the ordinary gradient method, the subgradient method is not a descent method; the function value can (and often does) increase. To study the convergence properties, the simplest case is to choose a generic minimization problem and assume f convex and L-c for some L. Given v_i the stepsize and x_* the minimum, the fundamental relationship is

$$\begin{aligned} ||x_{i+1} - x_*||^2 &= ||x_i - v_i g_i - x_*||^2 = \\ &= ||x_i - x_*||^2 + 2v_i\langle g_i, x_* - x_i\rangle + (v_i)^2||g_i||^2 \\ &\leq ||x_i - x_*||^2 + 2v_i(f_* - f(x_i)) + (v_i)^2||g_i||^2 \end{aligned} \tag{2.18}$$

(2.18) shows how the descent is influenced by two parts: the first is $2v_i(f_* - f(x_i))$ (negative), while the second is $(v_i)^2||g_i||^2$ (positive). As v_i tends to 0, the negative part dominates the recurrence, and the function decreases.
In practice, there's no control over individual stepsizes, only on "long term". The resulting rate is not good, and it's not feasible in practice.
If we knew f_*, with f convex and L-c, we can show that the optimal value for the stepsize is

$$v_* = \frac{(f(x_i) - f_*)}{||g_i||^2} \tag{2.19}$$

That is the Polyak Stepsize. Also, the convergence is now better, although not good. In theory, for the stepsize we require only very simple rules:

$$\sum_{i=1}^{\infty} v_i = \infty \quad \sum_{i=1}^{\infty} v_i^2 < \infty \tag{2.20}$$

The crucial point is that with an appropriate step along a (anti) subgradient, the algorithm converges. The stepsize can be chosen without any knowledge of the specific function to be minimized: any choice of the stepsize satisfying the diminishing/square summable conditions (2.20) leads to a convergent algorithm, for example, $1/i$ (although it's very inefficient in practice).

Theorem 3. *Let f be a Lipschitz continuous and convex function with domain R^n and Lipschitz constant $L > 0$, the best convergence rate for the subgradient method is*

$$O\left(\frac{1}{\epsilon^2}\right) \tag{2.21}$$

The proof can be found at [A.2].
An efficient rule adopted in our project is the Polyak-type target value stepsize rule [2] of the form:

$$vi = \frac{\beta_i(f_i - f_i^{lev})}{||d_i||^2} \tag{2.22}$$

Where f_i^{lev} is an approximation of f_*, the real minimum. The main problem is how f_i^{lev} is computed. There are two possible alternatives.
The first approach exploits an available lower bound, $\bar{l}$, of the function. Then use $\bar{l}$ as the approximation, also the lower bound can be used as a stopping condition.
The second approach is called *target following*, here $f^{lev} = f^{rec} - \delta_i$. In this approach $f^{rec} = min\{f_j : j = 1, ..., i\}$ is the record value, while $\delta_i > 0$ is chosen based on whether one intends to utilize the vanishing or non-vanishing rules, meaning $\delta_i \to 0$ as $i \to \infty$.
It is known that choosing the subgradient direction leads to the zigzagging phenomenon that might cause the procedure to progress slowly towards optimality. To overcome this situation, we can adopt a direction search that deflects the subgradient pure direction. These techniques are called *deflected subgradients*, this leads to replace (2.17) with

$$x_{i+1} = x_i - v_i d_i \tag{2.23}$$

Where the direction d_i is obtained by some linear combination of the current subgradient g_i and the previous direction d_{i-1}. For our experiments, we decided to use the standard version of the deflection to compute the next iterate, for the deflection parameter $\alpha_i \in [0, 1]$.

$$d_i = \alpha g_i + (1 - \alpha) d_{i-1} \tag{2.24}$$

The choice of the deflection parameter α can have a significant impact on the convergence properties of the deflected subgradient descent method with Polyak-type stepsize. In general, larger values of α can introduce more randomness and help escape from poor local optima, but may also slow down convergence. Smaller values of α can help converge faster to local optima, but may also get stuck in poor local optima.
Now that all the theoretical aspects are explained, we give the details for our specific case.
We tackle individual components of the function, beginning with Mean Squared Error (MSE). Since this function is differentiable, the subgradient at any given point is the gradient.
The L1 norm, instead, is the sum of the absolute value for every element in the vector. In this case, we know that the function is not differentiable in 0. The subgradients for a single term in the regularization can be determined by setting $x = 0$

$$\begin{aligned} f(z) &\geq f(0) + g(z - 0), \forall z \in Dom(f) \\ \lambda|z| &\geq \lambda|0| + g(z), \forall z \in Dom(f) \\ \lambda|z| &\geq g(z), \forall z \in Dom(f) \\ \text{if } z > 0\text{: } \lambda|z| &\geq g(z) \to g \leq \lambda \\ \text{else: } \lambda|z| &\geq g(z) \to g \geq -\lambda \end{aligned} \tag{2.25}$$

Starting from this inequality, we have that the subgradient of the L1 norm can be characterized as

$$\begin{cases} \lambda, & \text{if } x > 0 \\ -\lambda, & \text{if } x < 0 \\ [-\lambda, \lambda]. & \text{if } x = 0 \end{cases} \tag{2.26}$$

we decided to use the subgradient $0 \in \partial f(x)$ and use the function sign of NumPy to set the correct value.
To satisfy the theoretical properties, our function should be convex and L-c. Now, for the convexity part, we already discussed in the previous section how we can prove the non-convexity of our function, we don't need to discuss any further.
For the L-c property, the strategy takes into account the different parts of our function. Recalling that a function is L-c if

$$||f(x_1) - f(x_2)|| \leq L||x_1 - x_2||, \forall x_1, x_2 \tag{2.27}$$

To prove that MSE is L-c, we state that

Theorem 4. *Let f be a convex function and let K be a closed and bounded set contained in the relative interior of the domain Dom f of f. Then f is Lipschitz continuous on K.*

We know that the MSE is a convex function, the possible input values of the MSE are constrained by the output of the neural network. The potential output values in our case are contained within $[-1, 1]$, a closed and bounded interval. Therefore, the theorem holds true, and we can conclude that the MSE is locally L-c in the interval.
For the L1 norm, it is easy to see that is Lipschitz by simply exploiting the triangle inequality. Specifically, we know that

$$\begin{aligned} ||f(x_1) - f(x_2)|| &= ||\lambda||x_1|| - \lambda||x_2|||| \\ &\leq ||\lambda x_1 - \lambda x_2|| = ||\lambda|||x_1 - x_2||| \end{aligned} \tag{2.28}$$

Taking for example $L \geq ||\lambda||$ we can clearly satisfy the condition.
In the end, the final overall function is composed by two L-c function, so their composition is L-c.

Chapter 3

Experiments and Results

The first task is initializing the network. Getting the starting values of weights and biases right is crucial for optimization algorithms when training feedforward neural networks. In artificial neural network, the activation function and the weight initialization method play important roles in training and performance of a neural network. We decided to use Xavier normalized initialization [5], which advocates that the method helps in the training phase when the network is not deep. For each weight, the initial value is extracted from a Uniform distribution with range

$$[-\frac{\sqrt{6}}{\sqrt{n_i + n_j}}, \frac{\sqrt{6}}{\sqrt{n_i + n_j}}]$$

where n_i is the number of incoming network connections, n_j is the number of outgoing network connections from that layer.
Before delving into the required implementations, we needed to fix our model. So far, we had not explicitly defined the regularization parameter λ. For our purposes, we assumed the model to be fixed, focusing instead on comparing algorithm performance.
To determine an appropriate value for λ, we initially set it to 1×10^{-2} and observed the behavior of the algorithms using this fixed value. Subsequently, we plan to adjust λ to 1 in order to assess how varying the non-differentiability of the function impacts the algorithms. We have made the decision to maintain this parameter constant as we proceed with the actual experiments, during which we will compare the discussed algorithms.

3.1 Experiments

Our primary objective is to assess the descent capabilities of each algorithm. In the context of neural networks, finding the global minimum presents a significant challenge due to the high dimensionality and non-convex nature of the function, so generally it's acceptable and common, in this scenario, to stop in a local minimum or at least in a sufficiently low value that results in satisfactory model performance. To allow for meaningful comparison of the results generated by various algorithms, we decided to initialize each model only once, saving the initial values of the weights and biases. Each algorithm will start from the same point, aiming to converge to the same local minimum.
Each algorithm has is own set of parameters, the choices on what values are taken in consideration will be discussed in the next sections. The primary objective of the initial

tests was to evaluate how well the algorithms minimize the function. We created multiple sets of parameter combinations, treating each combination as an individual 'experiment'. After testing all possible combinations, we decided to use the best one among them, which was the set of parameters that led to the lowest value, and increase the number of iterations from 5000 to 1 million. In this way, we obtained an estimate of f_*.
This process was repeated independently for each value of lambda, ultimately yielding two approximations, one for each model.
In addition to visually observing how the function's value changes throughout the iterations, we decided to look upon different numerical measures to assess the effectiveness of the optimization process. To achieve this, we explore different measures aimed at capturing various aspects that signify a successful optimization progression:

- final value achieved: which is the last value obtained during after the execution of an algorithm.
- variation: which is the absolute difference between the starting value of the function and the final value, obtained after meeting one of the termination conditions.
- convergence speed: which we computed as the number of iterations needed to reach a value within a predetermined range of the final value, assuming the last iteration is denoted as "n". This metric offers insights into how rapidly the optimization process converges to a satisfactory solution.

  $$cs = \arg\min_i \{f(x_i) \leq f(x_n) + \rho; i = 1, ..., n\} \tag{3.1}$$

  In this way, we can measure the point after which the convergence of the algorithm slows.
- relative gap: the absolute difference between the last value in the descent and the optimum, divided by the optimum.

With the experimental setup laid out, we will now delve into our implementation of the algorithms and elaborate on the parameters we study to investigate in our experiments.

3.1.1 Heavy Ball algorithm

Starting from the Heavy Ball schema, in the pseudo-code, the regularization parameter is not explicitly presented, but it was used in the forward and backward propagation, line 3. The learning rate α was chosen between $\{0.05, 0.01, 0.005, 0.001, 5e-4, 1e-4,$ $5e-5, 1e-5\}$. The parameter μ was chosen from $\{0.1, 0.3, 0.5, 0.9, 0.995, 0.999\}$. The schedule was partially motivated by Nesterov [4] who advocates that this procedure is appropriate when the function is not strongly convex, and partially by our decision to explore the behavior of the algorithm with different values of the momentum.
The algorithm follows the code reported in **Algorithm 1**.

Algorithm 1: Heavy Ball pseudo-code

```
1: $d_0 = 0$, $i \leftarrow 1, \mu$ and $\alpha$ chosen from the schedules, network assumed initialized
2: while $True$ do
3:     compute $f_i = f(w_i)$ and $\nabla f(w_i)$        ▷ forward and backward propagation
4:     If any stopping condition is true, exit
5:     $d_{i+1} = \mu d_i - \alpha \nabla f(w_i)$
6:     $w_{i+1} = w_i + d_{i+1}$
7:     $i = i + 1$
8: end while
```

The algorithm follows directly what we reported in Section 2.1, and since the momentum term is always greater than 0, the previous direction is always taken into account before computing the new direction.
Here, we implemented three different stopping criteria:

- Norm of the Gradient: the algorithm stops when the norm of the gradient is below a certain threshold.
- Convergence Threshold: the algorithm stops when the change in the function's value between iterations is below a predefined threshold, indicating that further improvements will be minimal.
- Number of Iterations: to prevent the algorithm from running indefinitely, a maximum number of iterations is set, and the algorithm stops when this limit is reached.

With that, all the necessary elements for the heavy ball algorithm have been established, now we will present the deflected subgradient algorithm.

3.1.2 Deflected Subgradient algorithm

For the deflected subgradient algorithm we need to discuss several strategies. Starting from the stepsize, we decided to implement the Polyak-type target rule, that defines the stepsize as

$$vi = \frac{\beta_i(f_i - f_i^{lev})}{||d_i||^2} \tag{3.2}$$

For β, stepsize rule parameter, we decided to keep a constant value during the iteration, and modify the value only if $f_i^{lev} < f(x_i)$. The possible values for β are $\{0.1, 0.3, 0.5, 0.7, 0.9, 1\}$. Then, we have the target f^{lev}, computed as

$$f_i^{lev} = f_i^{rec} - \delta_i$$

Where $f_i^{rec} = min\{f(x_j), j = 1, ..., i\}$. Here, we implemented a rule for computing the approximation in different cases.
If $f(x_i) < f_i^{lev} - \delta_i$, δ_i is reset using the product of the default value, called a_*, and the maximum between the current value of the function and 1. The schedule for the possible values of a_* was $\{1e^{-3}, 1e^{-4}, 1e^{-5}, 1e^{-6}\}$.

Instead, if the condition is false, δ_i is decreased, provided only that it does not vanish. In the implemented solution, we decided to limit delta

$$\delta_{i+1} \in \max \begin{cases} \delta_i * \tau \\ \epsilon * \max(\min(f(x_i), f_i^{rec}), 1) \end{cases} \tag{3.3}$$

Where ϵ was used as a parameter in the tuning phase, the possible values were $\{0.01, 0.001, 1e-3, 1e-4, 1e-5\}$. The possible values for τ were $\{0.9, 0.95\}$.
Before the update step, we need to compute the direction, using a fixed value for α, the deflection parameter, selected between $\{0.1, 0.2, 0.5, 0.7, 0.9, 1\}$, where 1 is the case where the subgradient is not deflected.
Now that we have all the building blocks for the deflected SM procedure, we report the pseudo-code of the algorithm implemented, see **Algorithm 2**.

Algorithm 2: Deflected Subgradient pseudo-code

```
1: $d_0 = 0$, $i \leftarrow 1$, parameters chosen from the schedules, network assumed initialized
2: **while** $True$ **do**
3:     compute $f_i = f(w_i)$, $g_i \in \partial f(w_i)$
4:     If any stopping condition is true, exit
5:     $\delta_i = updateDelta()$
6:     $f^{rec} = min\{f_l; l = 1, ..., i\}$
7:     $f^{lev} = f^{rec} - \delta_i$                    ▷ Update target value
8:     $d_i = Deflection(g_i, d_{i-1}, \alpha)$
9:     $v_i = Stepsize(f_i, f^{lev}, d_i, \beta)$
10:    $w_{i+1} = w_i - v_i d_i$
11:    $i = i + 1$
12: **end while**
```

In the subgradient method, there is not a formal stopping criterion due to its slow theoretical convergence. In our implementation, we've opted for two specific conditions:

- Norm of the Subgradient: the algorithm stops when the norm of the subgradient is below a certain threshold, it's unlikely to happen in the case of the subgradient.
- Number of Iterations: to prevent the algorithm from running indefinitely, a maximum number of iterations is set, and the algorithm stops when this limit is reached.

3.1.3 Testing Correctness

In order to assess correct functioning of our algorithms and evaluate their performances, both algorithms have been tested on the Matyas function, which is quadratic, convex and has no local minima except the global one, $f(x_*) = 0$ in $x_* =$ (0,0). The starting point has been set to $x_0 =$ (-2.4,4.5), this point is not meaningful in any way, it was selected for graphical reason. We implemented the same algorithms, adding the gradient descent to have a third view. For all the algorithms, we decided to limit the iterations to 1000 and see the results.

Gradient descent was tested setting $\alpha = 0.3$ and it produced a final value equal to 8.30e-18.
Heavy Ball approach has been tested with the parameters α and μ set respectively to 0.3 and 0.8, and returns a final value of the function equal to 1.89e-21.
Lastly, the deflected subgradient algorithm has been tested with the following parameters: $\alpha = 0.9, \beta = 1, \epsilon = 0.001$, astart $= 0.1$ and $\tau = 0.99$ and gives as final value 1.73e-10.
In all cases the final value is really close to 0, that is the minimum of the function. We decided to reduce the number of iterations to 75 and plot the different trajectories, to see the differences in a test environment.

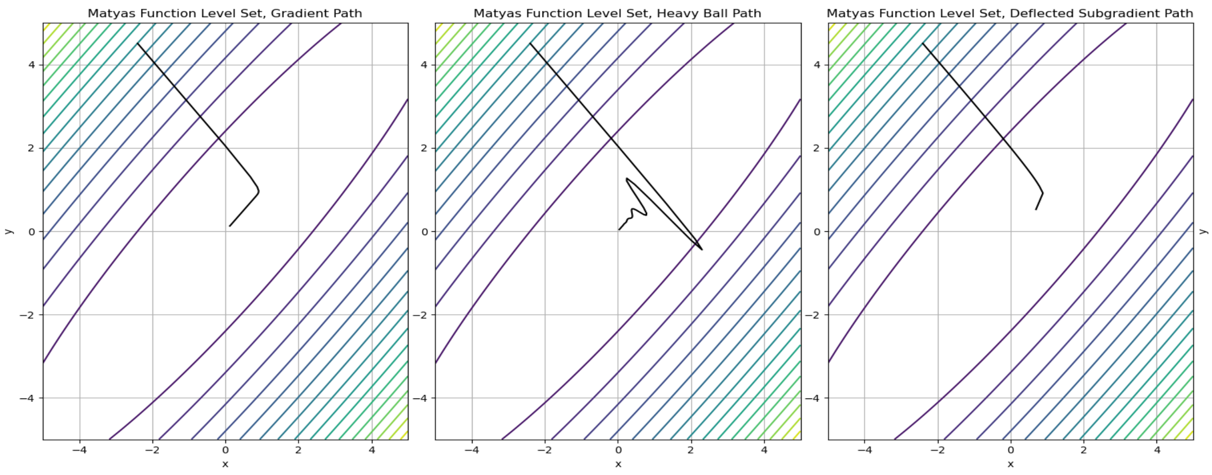


Figure 3.1: Comparison of SGD (left), momentum (center), and deflected subgradient (right). It can be seen how the the higher momentum term accelerates the convergence but oscillates more as it takes longer to decelerate once the direction changes.

Then, we decided to test our neural network implementation. We decided to build the exact same network using keras (https://github.com/keras-team/keras), a reliable high level library that provides a broad range of optimizers.
Keras has many optimizers implemented, including Heavy Ball. After constructing our neural network using Keras, we made a slight modification in our version to employ the classical MSE function. Then, we used the same initialization for the weights and biases, the Xavier normal initialization, called GlorotNormal in Keras.
Our training process involved limiting iterations to 1000 and repeating it 5 times. Each time we computed the absolute difference between the final losses, and in the end we averaged the results. Due to the stochastic nature of initialization, the final values varied across repetitions. However, the average difference between our model and the Keras model consistently remained equal to or lower than 2e-5.
The difference in this comparison with Keras, coupled with the results obtained on the test functions, seemed like a sufficient confirmation of the correctness of our implementation.

3.2 Results

We will now present the results for both algorithms with different values of the parameter lambda (1e-2 and 1).
Initially, during our testing phase, we observed some unusual behaviors. Despite starting each experiment from the same point, we often found that the values between configurations weren't comparable, leading to inadequate results.
To address this, we investigated further. Firstly, we removed all stopping criteria from the algorithms except for the number of iterations, then we set a limit of 100,000 iterations to approach the local minima as closely as possible. Lastly, we computed the norm of the gradient at the final descent point.
We noticed that both norms were relatively small, ranging between 1e-3 and 1e-5 for the model with $\lambda = 1e-2$. However, despite this, the final losses still differed significantly. In particular, the heavy ball algorithm reached always a higher value. The algorithms converge to different minima, so in the experiments we will compute different approximations for the local minima for each best configurations, one for the heavy ball algorithm and one for the deflected subgradient one.

The first model we studied was the case $\lambda = 1e-2$. We initialized the network and saved our initial weights and biases, to use them as fixed starting point in all of the experiments.
As reference, we implemented a standard gradient descent algorithm and set the iteration limit to 10,000. After a quick tuning phase the output value was 0.0322.
Starting with the heavy ball algorithm, we decided to run all of the experiments with 1000 iterations as limit, Table 3.1 display the 5 best results in terms of final value. Before delving into the results, it's important to highlight that we computed the approximation of the optimal value after the tuning phase of both the heavy ball and the deflected subgradient (in particular, we used a gridsearch for Heavy Ball and a random search for the Deflected Subgradient). Essentially, we opted to employ the most effective configurations, one for each method, and re-run them, increasing the iteration limit to 100,000. Then we used those values to get an approximation of the local minima.
It is noticeable how the momentum term has a positive impact on the descent: even with fewer iterations the final value, obtained with the best configuration, improves compared to the gradient descent.
Recalling that the convergence speed is the value of i, the number of iterations, such that

$$f(x_i) \leq f(x_n) + \rho$$

we decided to set $\rho = 0.001$ and observe the results. We can notice how the convergence speed is very different among the configurations: the values range from 152 to 987. Although it is not the project's primary focus, this result suggests potential applications in classical machine learning problems where efficiency can be as crucial as precision.

Parameters	Final Value	Relative Gap	Difference	Speed
$\alpha : 0.01, \mu : 0.9$	0.0320	1.9e-4	0.4031	693
$\alpha : 0.05, \mu : 0.5$	0.0321	3.6e-3	0.4030	713
$\alpha : 0.05, \mu : 0.0$	0.0323	9.5e-3	0.4026	152
$\alpha : 0.05, \mu : 0.9$	0.0332	3.6e-2	0.4019	960
$\alpha : 0.05, \mu : 0.1$	0.0436	3.5e-1	0.3916	987

Table 3.1: 5 Best results (final loss) reached with heavy ball.

Now that the general results are exploited, we dig a little more into the descent phase. The expected convergence for the squared error, if the function satisfies all the assumptions, is stated in Theorem 2, and it is

$$O\left(\frac{1}{k}\right) \tag{3.4}$$

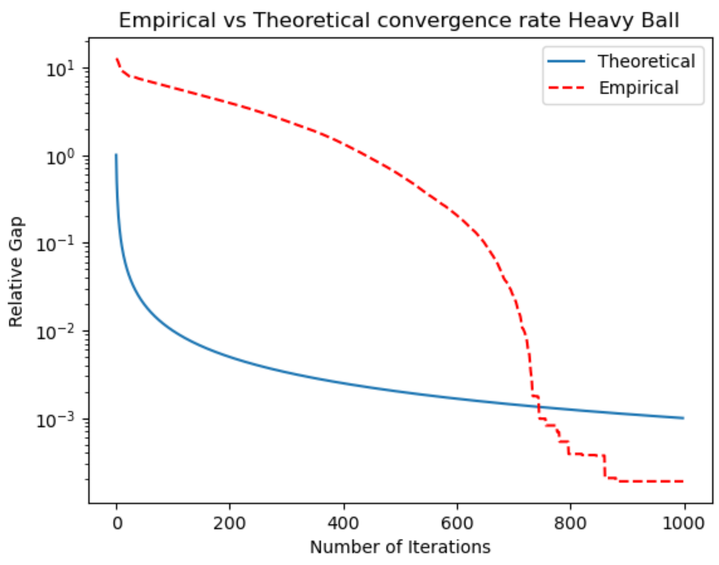


Figure 3.2: Convergence for the squared error, here we compare the theoretical results obtained with the iPiano algorithm and our experiment using the neural network.

In the graph, we observe how the heavy ball algorithm behaves in comparison to the theoretical expectation. We can see that even tough the relative gap does not follow the theoretical line, in the end the precision is better. This is, of course, and approximated value and it is hard to make other considerations.
The table 3.2 reports the 5 best results for the deflected subgradient. In this case, the measures taken into consideration are the same as before, while the parameters are alpha, beta, astart, tau and eps. The ρ and L1 are still fixed to $1e{-}3$ and $1e{-}2$ respectively.
Before presenting the obtained results, it is important to specify that since the subgradient algorithm does not guarantee a reduction of the function between one iteration and the following, we created two data structures: one to collect the sequence of values obtained and another to gather the 'best' value obtained up to that point. The results were

analyzed by looking at the second data structure. From the results, we can see that the performances of the deflected subgradient are better, in terms of relative gap, than to the ones obtained with the heavy ball, the opposite of what the theory suggested to us.

Parameters	Final Value	Relative Gap	Difference	Speed
$\alpha : 1, \beta : 0.1, a_* : 0.01, \tau : 0.9, \epsilon : 0.1$	0.0326	5.8e-4	0.4025	769
$\alpha : 0.9, \beta : 1.7, a_* : 0.1, \tau : 0.9, \epsilon : 0.01$	0.0330	1.1e-2	0.4022	406
$\alpha : 0.3, \beta : 2, a_* : 0.2, \tau : 0.95, \epsilon : 1e-4$	0.0330	1.2e-2	0.4021	334
$\alpha : 1, \beta : 0.9, a_* : 0.2, \tau : 0.95, \epsilon : 0.01$	0.0337	3.2e-2	0.4015	525
$\alpha : 0.9, \beta : 0.1, a_* : 1e-4, \tau : 0.99, \epsilon : 0.1$	0.0338	3.7e-2	0.4013	737

Table 3.2: 5 Best results (final loss) reached with deflected subgradient.

Then, we repeated the study done for the Heavy Ball algorithm. We ran the algorithm with more iterations and used the minimum as the optimum approximation.
Theorem 3 states that if a function is convex and Lipschitz then the subgradient method has a convergence rate of

$$O\left(\frac{1}{\sqrt{k}}\right) \tag{3.5}$$

We exhibit our result, obtained with the best configuration, in Figure 3.3, where we can see that the empirical relative gap is better than the expected one. Again, we can not say a lot more than that because we used an approximation of the minimum, not the true minimum.

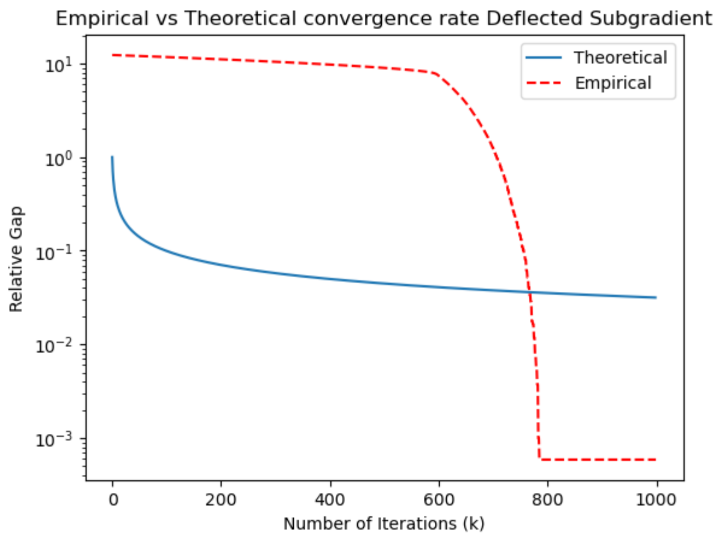


Figure 3.3: Convergence for error, here we compare the theoretical results derived for the subgradient algorithm and our experiments.

To get an idea of the different performances we decided to plot the results in the same scale.

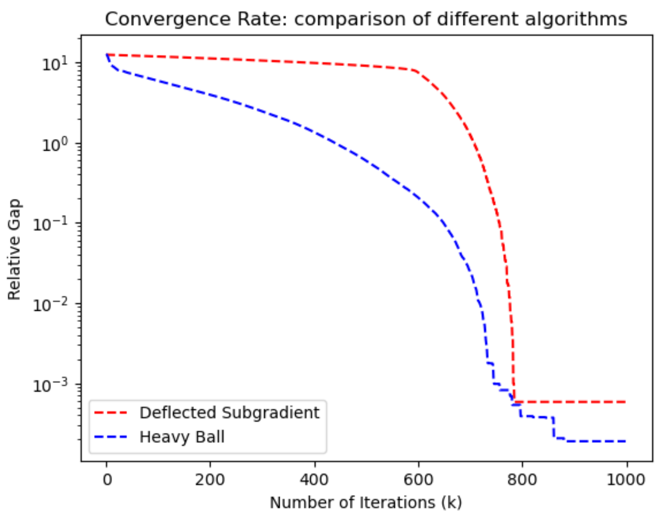


Figure 3.4: Convergence for error, comparison between classical momentum and deflected subgradient.

Figure 3.4 illustrates that between the heavy ball approach and the deflected subgradient, the gap favors the first but there is no significant difference.
After having conducted the analysis fixing $\lambda = 1e{-2}$ we repeated the entire procedure with the regularization parameter set to 1. For the other models we did again the search for the best configurations, starting from the same point in the space, and, once we obtained the best parameters, we computed again different approximations of the local minima. In this way we were able to compare the descent properties for each model, as the non-differentiability of the function changes.
Figure 3.5 illustrates the results for $\lambda = 1$. The heavy ball is, in this case, worse than the subgradient method. In this case the difference is more marked than before, but the overall performance of both models are worse when compared to the ones obtained in the first model.

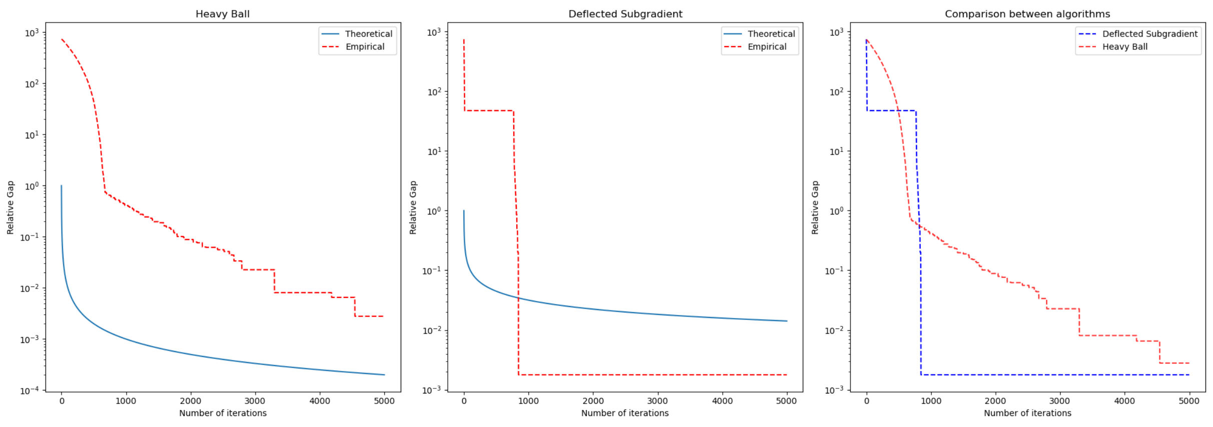


Figure 3.5: Results obtained with the same model and λ =1. From left to right, heavy ball, deflected subgradient and both.

In this case the number of iterations in the tuning phase was set to 5,000. This choice was made because as the non-differentiability of the function increases, we noticed that our algorithms struggled to keep the same precision. For the results, we refer A.1, A.2

Chapter 4

Conclusions

In this report, we have taken a straightforward approach by randomly sampling the training dataset and consistently using the same fixed subset to train our network. It is worth highlighting that machine learning theory offers more reliable techniques, such as cross-validation and nested cross-validation, which ensure the appropriate generalization properties of the model.
Our objective was to assess the behavior and characteristics of two distinct optimization algorithms, classical momentum and deflected subgradient, while keeping the model predetermined. Additionally, we simplified our problem by selecting a loss function that, while generally unsuitable for classification tasks, holds favorable properties for our specific task, such as differentiability. Consequently, the predictive performance of our trained models was limited in practice. However, it is important to note that achieving high predictive accuracy was not the primary aim of this project.
Our experiments show two different results: in the first case, with the parameter λ of the model set to $1e{-}2$, the Heavy Ball algorithm presents slightly better performances. In the second, with the model having a parameter λ equal to 1, the Deflected Subgradient is better than the Heavy Ball, but the overall performances are worse than the ones of the previous case.

Bibliography

[1] B.T. Polyak. "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17. ISSN: 0041-5553. DOI: https://doi.org/10.1016/0041-5553(64)90137-5. URL: https://www.sciencedirect.com/science/article/pii/0041555364901375.

[2] B.T. Polyak. "Minimization of unsmooth functionals". In: *USSR Computational Mathematics and Mathematical Physics* 9.3 (1969), pp. 14–29. ISSN: 0041-5553. DOI: https://doi.org/10.1016/0041-5553(69)90061-5. URL: https://www.sciencedirect.com/science/article/pii/0041555369900615.

[3] S. K. Zavriev and F. V. Kostyuk. "Heavy-Ball Method In Nonconvex Optimization Problems". In: *Comput. Math. Model.* 4 (1993), pp. 336–341. URL: https://link.springer.com/content/pdf/10.1007/BF01128757.pdf.

[4] Yurii Nesterov. *Introductory Lectures on Convex Optimization.* Springer New York, NY, 2003. URL: https://link.springer.com/book/10.1007/978-1-4419-8853-9.

[5] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics.* Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[6] Peter Ochs et al. *iPiano: Inertial Proximal Algorithm for Non-Convex Optimization.* 2014. arXiv: 1404.4805 [cs.CV].

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

[8] Peter Ochs. "Local Convergence of the Heavy-Ball Method and iPiano for Non-convex Optimization". In: *Journal of Optimization Theory and Applications* 177.1 (2018), pp. 153–180. ISSN: 1573-2878. DOI: 10.1007/s10957-018-1272-y. URL: https://doi.org/10.1007/s10957-018-1272-y.

[9] C. De Stefano et al. "Reliable writer identification in medieval manuscripts through page layout features: The "Avila" Bible case". In: *Engineering Applications of Artificial Intelligence* 72 (2018), pp. 99–110. ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2018.03.023. URL: https://www.sciencedirect.com/science/article/pii/S0952197618300721.

Appendix A

A.1 Heavy ball and Momentum are the same thing

To see that 2.2 and A.1 are the same, we first expand 2.2

$$x_{i+1} = x_i - \alpha_i \nabla f(x_i) + \mu d_i$$

Since

$$x_i = x_{i-1} + d_i$$
$$d_i = x_i - x_{i-1}$$

We can replace d_i and say that

$$x_{i+1} = x_i - \alpha_i \nabla f(x_i) + \mu(x_i - x_{i-1})$$

With that, we have exactly the other form.

A.2 Convergence rate of subgradient method

Let f be a Lipschitz continuous convex function with domain R^n and Lipschitz constant $L > 0$

$$||f(x) - f(y)|| \leq L||x - y||$$

From the definition of the subgradient, calling g any subgradient of f, v_i the stepsize and x_* the minimum of the function

$$||x_i - x_*|| \leq ||x_{i-1} - x_*|| - 2v_i(f(x_{i-1}) - f(x_*)) + v_i^2||g_{i-1}||$$

Rewriting it in terms of x_0

$$||x_i - x_*|| \leq ||x_0 - x_*|| - 2\sum_{i=1}^{k} v_i(f(x_{i-1}) - f(x_*)) + \sum_{i=1}^{k} v_i^2||g_{i-1}||$$

Using $||x_i - x_*|| \geq 0$, letting $R = ||x_0 - x_*||$

$$0 \leq R^2 - 2\sum_{i=1}^{k} v_i(f(x_{i-1}) - f(x_*)) + G^2\sum_{i=1}^{k} v_i^2$$

Introducing $f(x_{best}) = \min_{i=0,\dots,k} f(x_i)$ and substituting for $f(x_{i-1}) - f(x_*)$ makes the right side larger, we can write

$$0 \leq R^2 - 2\sum_{i=1}^{k} v_i(f(x_{best}) - f(x_*)) + G^2 \sum_{i=1}^{k} v_i^2$$

Rearranging gives what's called the basic inequality:

$$f(x_{best}) - f(x_*) \leq \frac{R^2 + G^2 \sum_{i=1}^{k} v_i^2}{2\sum_{i=1}^{k} v_i}$$

If we assume that the stepsize is fixed we have that

$$f(x_{best}) - f(x_*) \leq \frac{R^2}{2kv} + \frac{G^2 k v^2}{2kv} = \frac{R^2}{2kv} + \frac{G^2 v}{2}$$

To force the right-hand side of the basic inequality to be less than ϵ, we can force both terms to be less than $\epsilon/2$.
Rearranging the second term and maximizing step size v, results in $v = \epsilon/G^2$. Rearranging the first term to solve for k, plugging in v and minimizing k gives

$$\frac{R^2 G^2}{\epsilon^2} \rightarrow O(\frac{1}{\epsilon^2})$$

Thus, for an ϵ optimal solution, we need $O(1/\epsilon^2)$ iterations.

A.3 Results for $\lambda = 1$

Parameters	Final Value	Relative Gap	Difference	Speed
$\alpha : 1e-4, \mu : 0.9$	0.0391	2.7e-3	28.845	2792
$\alpha : 5e-4, \mu : 0.5$	0.0492	2.6e-1	28.444	2412
$\alpha : 0.001, \mu : 0.5$	0.0679	7.4e-1	28.424	1150
$\alpha : 0.001, \mu : 0.3$	0.0682	7.4e-1	28.424	2294
$\alpha : 5e-4, \mu : 0.9$	0.0687	7.6e-1	28.423	1933

Table A.1: 5 Best results (final loss) reached with heavy ball, lambda = 1.

Parameters	Final Value	Relative Gap	Difference	Speed
$\alpha : 0.5, \beta : 1.7, a_* : 0.2, \tau : 0.95, \epsilon : 1e-5$	0.0383	1.7e-3	28.461	845
$\alpha : 0.5, \beta : 1.0, a_* : 0.2, \tau : 0.95, \epsilon : 0.01$	0.0395	5.2e-2	28.457	435
$\alpha : 1, \beta : 1.7, a_* : 0.2, \tau : 0.95, \epsilon : 1e-3$	0.0471	2.3e-1	28.453	1001
$\alpha : 0.5, \beta : 1.7, a_* : 0.1, \tau : 0.95, \epsilon : 1e-5$	0.0478	2.4e-1	28.452	157
$\alpha : 0.7, \beta : 2, a_* : 0.2, \tau : 0.9, \epsilon : 1e-5$	0.0493	2.8e-1	28.450	81

Table A.2: 5 Best results (final loss) reached with deflected subgradient, lambda = 1.