Computational Mathematics for Learning and Data Analysis

Project report

Track

- (M) is a neural network with topology and activation function of your choice, provided it is differentiable; only smooth regularization (say, L_2) is allowed.
- (A1) is a standard momentum descent approach.
- (A2) is an algorithm of the class of limited-memory quasi-Newton methods

Gabriele Pisciotta
596217
g.pisciotta1@studenti.unipi.it

Marco Sorrenti
619615
m.sorrenti4@studenti.unipi.it

Group 7

Contents

Chapter 1

Introduction

The aim of this project is to develop an Artificial Neural Network simulator implementing different optimization techniques. Recalling the project track (Project 2), we must implement:

- *standard momentum descent approach*;
- *an algorithm of the class of limited-memory quasi-Newton methods*;

Regarding the second point, we chose to investigate the usage of L-BFGS [7].

1.1 Neural Network

A Neural Network (NN), or Multi-Layer Perceptron (MLP), is a computational model which aim is, given a specific input, to produce the desired output (according to the task, for example *classification* or *prediction*), after a specific *learning* phase in which its internal parameters are tuned by involving a learning algorithm.
More in details, the NN can be seen as a fully-connected set of layers, each one having a set of weights ($\mathbf{w}$) according to the size of neurons (or computing elements) plus a bias (b). A layer take an input ($\mathbf{x}$) that is a sample if the first layer or the output generated by the previous layer and finally returns an output by combining the input with the weights (leading to the so called net) and passing it through an *activation function* which aim is to introduce a non-linearity. More formally, for a single layer, be $\mathbf{x}, \mathbf{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$:

$$\boldsymbol{net} = \mathbf{w}^T\mathbf{x} + b \tag{1.1}$$

$$\boldsymbol{o} = f_\sigma(net) \tag{1.2}$$

The output of an hidden layer (o_ℓ, with ℓ specifying the proper layer w.r.t. Eq. 1.2) is the input of the successive layer's net (w.r.t. Eq. 1.1). As consequence, it's possible to write the full equation of a NN given its architecture, e.g.: the equation describing a NN with 1 hidden layer and 1 output layer, where f_o and f_h two activation functions related to the output layer and hidden layer respectively, is:

$$\boldsymbol{h}(\boldsymbol{x}) = f_o(\boldsymbol{w}_o^T * (f_h(\boldsymbol{w}_h^T\boldsymbol{x} + b_h)) + b_o) \tag{1.3}$$

In order to distinguish between the output of a hidden layer and the result of the prediction, also called *hypothesis*, the latter will be denoted as $\boldsymbol{h(x)}$ as stated in the previous equation.

1.2 Activation Functions

An activation function is a fundamental component of the NN architecture, being it responsible for the kind of output emitted by hidden layers and the output layers. According to the particular task we would need to consider different kind of activation functions: in the specific for this project we took into account the *linear* activation function and the *sigmoid*.
The first is the following:

$$f_\ell(\boldsymbol{x}) = \boldsymbol{x} \tag{1.4}$$

and doesn't add any non-linearity to the input that will be fed into it and will be outputted without any change. This is the typical choice for the output layer in a NN involved for regression tasks.

The sigmoid function, also called logistic function, is a function that introduces non-linearity and is one of the most used functions for both the hidden layer and output layer, the last if we're in presence of a classification task where the output must be clamped between two values (e.g.: $\{0, 1\}$). It can be defined as:

$$f_\sigma(\boldsymbol{x}) = \frac{1}{1 + e^{-\alpha \boldsymbol{x}}} \tag{1.5}$$

where α is a smoothing factor. Its derivative can be defined as:

$$f'_\sigma(\boldsymbol{x}) = f_\sigma(\boldsymbol{x})(1 - f_\sigma(\boldsymbol{x})) \tag{1.6}$$

making it easy and computational convenient to be implied. Without introducing non-linearity, so having k hidden layer with a linear activation function, all the hidden layers would be equivalent to a single hidden layer. Furthermore, non-linearities are needed for the Universal Approximation Theorem to be valid [1], with the NN being a composition of non-linear activation functions. Another interesting characteristing related to the sigmoid is that it's twice continuously differentiable, a requested feature for applying a certain class of algorithms.

1.3 Learning

To what concerns the *learning*, we must recall that we're in a supervised setting, so our model will learn exploiting a dataset: a set of samples with specific labels describing the expected outcome. More formally:

$$\{(\boldsymbol{x}_i, \boldsymbol{d}_i)\}_{i=1}^n \tag{1.7}$$

where n is the amount of samples, and the label d_i can have assume values according to the specific task (e.g.: can be $\{0, 1\}$ for a binary classification task while for a simple regression of a univariate function can be $d \in \mathbb{R}$, and so on). To properly learn we need to assess the performance of the model at any moment, and the loss function is responsible of this. It describes the distance of the model from the objective function to be learned. The loss functions that has been considered for this project is the Mean Squared Error (MSE):

$$\mathscr{E} = \frac{1}{n} \sum_i^n (\boldsymbol{d_i} - \boldsymbol{h(x)_i})^2 \tag{1.8}$$

where d is the label so the output that should be produced, $\boldsymbol{h(x)}$ is the one produced by the model (recalling Eq. 1.2).

The learning, so the model's parameters tuning (in our case, the *weights* and *bias*), will be addressed as a *minimization problem* in which we aim to find the minimum of the *loss function*: the more the loss function become small, the better our model performs.

The *backpropagation* is the technique used to properly find the parameters (w) in an iterative way such that the error is minimized. For convenience we'll omit the bias (b), as it can be considered absorbed by the weights. The algorithm is the 1.

Algorithm 1 Backpropagation Algorithm

$w \leftarrow initialize()$
compute output h(x) and loss $\mathscr{E}$
while not converged **do**
  for each $layer \in NN$ (from tail to head) **do**
    $\Delta w_\ell \leftarrow -\frac{\partial \mathscr{E}}{\partial w_\ell}$
    $w_\ell \leftarrow w_\ell + \alpha \Delta w_\ell + ...$
  compute output h(x) and loss $\mathscr{E}$

Until the convergence is reached, in each *epoch* (also known as *training iteration*) the algorithm first computes the output of the model by propagating the input through the layers in a forward step and the error (evaluating the loss function). Then the error is propagated back into each layer from the tail to the head, and the weights are updated according to the weight correction δw that is proportional to the partial derivative $\frac{\partial E_{tot}}{\partial w}$.
To what concerns the following notation, for consistency reasons we'll exploit the notation involved in [4] and accordingly the derivations described in the book.
Having already specified that our loss function is the MSE (Eq. 1.8), we can consider the *error signal* produced at the output of neuron j having fed the NN with an input sample $\boldsymbol{x}$. The *error signal* represents the difference between the desired response and the actual response of the network, defined by:

$$e_j = d_j - o_j \tag{1.9}$$

Having computed the output of the network, and having evaluated the *loss function*, these information will be exploited to change the weights in order to achieve a better predicted value by minimizing the loss. The correction applied to the weights is, in fact, proportional to the partial derivative $\frac{\partial \mathscr{E}}{\partial w_{ji}}$.
According to the chain rule of calculus, we may express this gradient as [4]:

$$\frac{\partial \mathscr{E}}{\partial w_{ji}} = \frac{\partial \mathscr{E}}{\partial e_j} * \frac{\partial e_j}{\partial o_j} * \frac{\partial o_j}{\partial net_j} * \frac{\partial net_j}{\partial w_{ji}} \tag{1.10}$$

which, differentiating each component, is equal to:

$$\frac{\partial \mathscr{E}}{\partial w_{ji}} = -e_j * f'_{\sigma_j}(net_j) * o_i \tag{1.11}$$

At this point the correction Δw_{ji} can be addressed as:

$$\Delta w_{ji} = \alpha \delta_j o_i \tag{1.12}$$

where δ_j is the local gradient described by:

$$\delta_j = e_j * f'_{\sigma_j}(net_j) \tag{1.13}$$

Now, we need to distinguish between two cases:

1. if the neuron j is an output node, so it's in the output layer, δ_j equals the equation 1.13

2. if the neuron j is a hidden node, δ_j it's equal to the product of the associated derivative $f'_{\sigma_j}(net_j)$ and the weighted sum of the *delta*s computed for the neurons in the next hidden or output layer that are connected to neuron j as follows:

$$\delta_j = f'_{\sigma_j}(net_j) * \sum_k \delta_k w_{kj} \tag{1.14}$$

As it's possible to see from the Algorithm 1, the final update rule have "..." in it: this means that it can be extended to address different things that could change the behavior of the learning algorithm, such as adding penalty terms (regularization) and momentum.

To what concerns the convergence of the algorithm, this can be tackled in different ways such as considering a fixed maximum amount of epochs, checking the norm of the gradient or even considering to compute a specific performance metric (e.g.: accuracy) on a different dataset and stop when the performance stops getting better.
Regarding the α parameter in the equation 1.12, it's a scalar that defines the magnitude of parameter updates during gradient descent: it's commonly defined as *learning rate*, but it could also be addressed as a *step length*. For the Machine Learning (ML) project we considered it as an hyperparameter of the model to be seek through a grid search, trying to find a balance between speed of convergence and stability.
A different strategy, that we'll address to involve in our project, is to find the optimal α for each of the update by involving a Line Search algorithm (see Section 1.5).

1.4 Regularization

When dealing with ML algorithms, we should always care to what concerns the complexity of the models, which needs to be controlled in order to avoid to fall in cases of overfitting / underfitting.
Let's consider a slighty different definition for the concept of training in ML: *the training of a network by means of examples, designed to retrieve an output pattern when presented with an input pattern, is equivalent to the construction of a hypersurface (i.e., multidimensional mapping) that defines the output pattern in terms of the input pattern* [4]. In real-life, training samples could suffer from many problems such as noise, missing labels, and more in general to not being sufficient by themselves to reconstruct the unknown input–output mapping uniquely. This is important because for these reasons we violate the three Hadamard's rules for problem *well posedness*, which means that even large datasets could have a small amount of desired information about the desired solution [4]. To being successful in dealing with the training which is an ill-posed problem, and so to avoid overfitting, we rely on Tikhonov's regularization theory, which suggest to add a penalty term to the cost function in order to regularize it.

We decided to involve the L_2 regularization, also informally known as "weight decay", which consists in adding to the loss function the term:

$$\lambda\Omega(\boldsymbol{w}) = \frac{\lambda}{2}\|\boldsymbol{w}\|_2 \tag{1.15}$$

where λ is an hyperparameter to be tuned. The effect of the L_2 regularization is to keep the weights of the network smaller, implying the fact that they won't be altered too much from specific input samples during the learning, so making more difficult to learn noise in

the data. Furthermore, the regularization is driven by the λ hyperparameter: if it's high the training can become slower and we could even go in underfitting due to the parameters penalty being too high. Similarly, if it's too low and under certain condition, it's possible to fall in overfitting. For the chose of the best λ value, we will rely on a proper search of the hyperparameters.

From now on, when refering to to *loss function*, we will refer to the following combination of the MSE plus the regularization term, with a slight overload of the notation used until now:

$$\mathscr{E} = \frac{1}{n}\sum_{i}^{n}(\boldsymbol{d_i} - \boldsymbol{h(x)_i})^2 + \frac{\lambda}{2}\|\boldsymbol{w}\|_2 \tag{1.16}$$

1.5 Line Search

A line search algorithm is responsible to find the right *step length* which minimize the loss function. The step length α_k is a positive scalar, which represent the learning rate. Each iteration of a line search method computes a search direction p_k which tells how far is right to move along a given direction. The iteration is given by:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha_k \boldsymbol{p}_k \tag{1.17}$$

So, in the line search strategy, fixed the descent direction p_k and the w_k, we are able to find the right α_k by approximately solving the following onedimensional minimization problem:

$$\min_{\alpha>0} \mathscr{E}(\boldsymbol{w}_k + \alpha_k \boldsymbol{p}_k) \tag{1.18}$$

Different kinds of line search algorithm are available, but, since we are looking for a substantial reduction of $\mathscr{E}$ without spending too much time computing α_k, we decide to use an *inexact line search* to identify a step length that achieves adequate reductions in $\mathscr{E}$ at minimal cost.
Line search is performed in two phases [6]:

- A *bracketing phase* which finds an interval containing desirable step lengths;
- An *interpolation phase* which computes a good step length within this interval;

To avoid the insufficient reduction in $\mathscr{E}$ at each step, which causes it to fail to converge to the minimizer, we need to enforce a *sufficient decrease condition*, which states that:

$$\phi(\boldsymbol{w}_k + \alpha_k \boldsymbol{p}_k) \leq \phi(\boldsymbol{w}_k) + c_1 \alpha \nabla \phi_k^T \boldsymbol{p}_k \tag{1.19}$$

where $c_1 \in (0, 1)$ and the reduction in ϕ should be proportional to both the step length α_k and the directional derivative $\nabla \phi_k^T p_k$. This condition is also called *Armijo condition.* c_1 is often chosen to be quite small, say $c_1 = 10^{-4}$.
Since the sufficient decrease condition is not enough by itself to ensure that the algorithm makes reasonable progress we introduce the so called *curvature condition*:

$$\nabla \phi(\boldsymbol{w}_k + \alpha_k \boldsymbol{p}_k)^T \boldsymbol{p}_k \geq c_2 \nabla \phi_k^T \boldsymbol{p}_k \tag{1.20}$$

where $c_2 \in (c_1, 1)$. Should be taken into account the fact that when $\boldsymbol{p}_k$ is obtained by a Newton or quasi-Newton method, c_2 is often chosen to be 0.9, while $c_2 = 0.1$ when p_k is obtained from a nonlinear conjugate gradient method.
The sufficient decrease and curvature conditions together are known as the *Wolfe Conditions.*

However, by modifying the curvature condition we can ensure to choose steps whose size is not too small. This could be guaranteed by the *Strong Wolfe Conditions*, which states that:

$$\phi(\boldsymbol{w}_k + \alpha_k \boldsymbol{p}_k) \leq \phi(\boldsymbol{w}_k) + c_1 \alpha \nabla \phi_k^T \boldsymbol{p}_k, \tag{1.21a}$$

$$|\nabla \phi(\boldsymbol{w}_k + \alpha_k \boldsymbol{p}_k)^T \boldsymbol{p}_k| \geq c_2 |\nabla \phi_k^T \boldsymbol{p}_k| \tag{1.21b}$$

The Strong Wolfe condition doesn't allow the derivative to be positive, so we are sure that α found next to a stationary point of the function.
Line search algorithm follows:

Algorithm 2 Line Search Algorithm [6]

Require: $\alpha_{max} > 0$, $\alpha_1 \in (0, \alpha_{max})$, max_iter
  $\alpha_0 \leftarrow 0$
  $i \leftarrow 1$
  while $i \leq max\_iter$ **do**
    Evaluate $\phi(\alpha_i)$
    if $\phi(\alpha_i) > \phi(0) + c_1 \alpha_i \phi^{'}(0)$ or $[\phi(\alpha_i) \geq \phi(\alpha_{i-1})$ and $i > 1]$ **then**
      $\alpha_* \leftarrow$ **zoom**(α_{i-1}, α_i);
      return α_*;
    Evaluate $\phi^{'}(\alpha_i)$
    if $|\phi^{'}(\alpha_i)| \leq -c_2 \phi^{'}(0)$ **then**
      $\alpha_* \leftarrow \alpha_i$;
      return α_*;
    if $\phi^{'}(\alpha_i) \geq 0$ **then**
      $\alpha_* \leftarrow$ **zoom**(α_i, α_{i-1});
      return α_*;
    Choose $\alpha_{i+1} \in (\alpha_i, \alpha_{max})$;
    $i \leftarrow i + 1$

Algorithm 3 Zoom Algorithm [6]

Require: α_{low}, α_{hi}
  while True **do**
    $\alpha_j \leftarrow$ Interpolate (using quadratic and cubic)
    Evaluate $\phi(\alpha_j)$
    if $\phi(\alpha_j) > \phi(0) + c_1 \alpha_j \phi^{'}(0)$ or $\phi(\alpha_j) \geq \phi(\alpha_{low})$ **then**
      $\alpha_{hi} \leftarrow \alpha_j$;
    else
      Evaluate $\phi^{'}(\alpha_j)$;
      if $|\phi^{'}(\alpha_j)| \leq -c_2 \phi^{'}(0)$ **then**
        $\alpha_* \leftarrow \alpha_j$;
        return α_*;
      if $\phi^{'}(\alpha_j)(\alpha_{hi} - \alpha_{low}) \geq 0$ **then**
        $\alpha_{hi} \leftarrow \alpha_{low}$;
      $\alpha_{low} \leftarrow \alpha_j$;

Chapter 2

Optimizers

In this Chapter, we will talk about some optimization algorithms. First of all, the goal to achieve is to search and find $\boldsymbol{w}$ (set of weights) in order to minimize a specified loss function, recalling the definition of learning given in Sec. 1.3.

2.1 Momentum descent approach

The momentum descent approach is a gradient-based optimization algorithm that allows to speed up the gradient descent through a velocity vector in directions of persistent reduction in the objective across iterations.
The main idea of this technique is to update weights through a velocity vector $\boldsymbol{v}$ that stores the gradient elements and that takes account of the *momentum coefficient* $\mu \in [0, 1]$.

$$\boldsymbol{v}_k = \mu \boldsymbol{v}_{k-1} + \nabla \mathscr{E}(\boldsymbol{w}_k) \tag{2.1a}$$

$$\boldsymbol{w}_k = \boldsymbol{w}_{k-1} - \alpha \boldsymbol{v}_k \tag{2.1b}$$

When μ is 0, it's like saying that we're operating without the momentum, falling back in the classic case of the (Stochastic) Gradient Descent.

Algorithm 4 Momentum Descent Algorithm

Require: $\mu \in [0, 1]$, $\alpha > 0$
  w $\leftarrow$ initialize()
  v $\leftarrow$ initialize()
  compute output h(x) and loss $\mathscr{E}$
  while not converged **do**
    for each *layer* $\in NN$ (from tail to head) **do**
      $v_k = \mu v_{k-1} - \alpha \frac{\partial \mathscr{E}}{\partial w_\ell}$
      $w_\ell \leftarrow w_\ell + \boldsymbol{v}_k$
    compute output h(x) and loss $\mathscr{E}$

Let's considerate the effect of the inclusion of the Momentum term in the weight update rule, following [4]:

1. when the partial derivative $\frac{\partial \mathscr{E}}{\partial \boldsymbol{w}_{ji}}$ has the same sign on consecutive iterations, the weights are updated by a large amount, meaning that the momentum has an acceleration effect to what concern the descent of the backpropagation

2. when the previously described partial derivative has opposite signs on consecutive iterations, the weights are updated by a small amount, meaning that it has a stabilizing effect in directions that oscillate in sign

and these previously described effects basically lead to a better optimization and are highly desirable characteristic: for this improvement we have only a small modification cost to the classic backpropagation algorithm (see. Algorithm 1)

Momentum Descent convergence

Let's recall the fact that we're dealing with a not convex setting, due to the fact that our loss function is basically a combination of non convex functions, i.e.: sigmoid).
Regarding the non convex setting [5], we consider to satisfy the following:

- The gradient of the loss function is L-Lipschitz continuous, that is to assume that for some constant $L > 0$ we have:

$$\|\nabla\mathscr{E}(\boldsymbol{x}) - \nabla\mathscr{E}(\boldsymbol{y})\| \leq L\|\boldsymbol{x} - \boldsymbol{y}\|, \forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n \tag{2.2}$$

- Because of the convergence rate of this algorithm is strictly connected to the choice of α, in order to ensure the convergence we consider:

$$\sum_{k}^{\infty} \alpha_k = \infty \tag{2.3a}$$

$$\sum_{k}^{\infty} \alpha_k^2 < \infty \tag{2.3b}$$

Before demontrate the Lipschitz continuity of the gradient of the loss, let's consider the fact that the loss function is made of the composition of the output function (i.e..: sigmoid) which *has a Lipschitz constant equal to 1* [8] and the Euclidean norm which is again Lipschitz continuous. At this point, we can involve the following theorem took from [2] to state that the the loss is Lipschitz continuous:

Theorem 1 *Let f_1 be Lipschitz continuous on a set h with Lipschitz constant L_1 and f_2 be Lipschitz continuous on I_2 with Lipschitz constant L_z such that $f_1(I_1) \subset I_2$. Then the composite function $= f_2 \circ f_1$ is Lipschitz continuous on I_1 with Lipschitz constant L_1L_2.*

Due to the fact that we're using a sigmoid activation function which has its output bounded in [0, 1], and that the Euclidean norm takes this in input, we can say that our loss function is Lipschitz continuous. *We are anyway interested in the gradient of the loss.*

We can recall other two theorems from [2] which states:

Theorem 2 *If f_1 and f_2 are Lipschitz continuous on a bounded interval I, then $f_1 * f_2$ is Lipschitz continuous on I.*

Theorem 3 *Suppose that f_1,..., f_n are Lipschitz continuous on a bounded interval I, with Lipschitz constants L_1,...,L_n respectively. Then the linear combination $c_1f_1 + ... + c_nf_n$ is Lipschitz continuous on I with Lipschitz constant $|c_1|L_1 + ... + |c_n|L_n$*

Recalling Eq. 1.11, and expanding it with the regularization term, we have:

$$\frac{\partial\mathscr{E}}{\partial w_{ji}} = -e_j * f'_{\sigma_j}(net_j) * o_i - \lambda 2 * w_{ji} \tag{2.4}$$

Looking at the previous equation, we can see that for the first part related to the gradient of the MSE, we have that the theorems previously recalled are not a-priori satisfied, considering also the fact that weights are not in a bounded set. Considering that the derivative of a

sigmoid, Eq. 1.2, is Lipschitz continuous, and that the derivative of the MSE loss function is made also of composition of product of sigmoids, we could reason about the fact that in the first iteration of the algorithm it could be ensured the Lipschitz continuity of the gradient of the MSE loss having considered the initialization of the weights from a uniform distribution in in [-1, +1] (or using the Xavier initialization [3]). To what concerns the subsequent iterations, we could try to enforce the property to our loss function by looking at the regularization term of the Eq. 2.4, so by considering the gradient of the full "MSE + regularization" loss. We need at this point to consider the monotonicity of the descent achieved by the algorithm involved. In general, it's not guaranteed to be monotonic, but as stated in [6]: *Line search methods may use search directions other than the steepest descent direction. In general, any descent direction – one that makes an angle of strictly less than* $\frac{\pi}{2}$ *radians with -*∇f_k *– is guaranteed to produce a decrease in* f*, provided that the step length is sufficiently small.* For the sake of prove the Lipschitz continuity property and a theoretical convergence of the algorithm, having considered the fact that a monotonous decrease of the loss values it's possible, supposed that $0 < \lambda \leq 1$:

$$\mathscr{E}(\boldsymbol{w}_i) \leq \mathscr{E}(\boldsymbol{w}_0), \forall i \in (1, ..., k)$$

where k is the number of iterations of the learning algorithm.

$$\mathscr{E}(\boldsymbol{w}_i) + \lambda ||\boldsymbol{w}_i||_2 \leq \mathscr{E}(\boldsymbol{w}_0)$$

The first term of the loss function, the MSE, is positive and so we can write

$$\lambda ||\boldsymbol{w}_i||_2 \leq \mathscr{E}(\boldsymbol{w}_0)$$

$$||\boldsymbol{w}_i||_2 \leq \frac{\mathscr{E}(\boldsymbol{w}_0)}{\lambda}$$

$$||\boldsymbol{w}_i|| \leq \sqrt{\frac{\mathscr{E}(\boldsymbol{w}_0)}{\lambda}}$$

Proving that the weights are bounded. Having selected the proper λ value, the regularzation terms in fact has the effect to keeping the weights limited in their magnitude, a solution that has the effect of avoiding the unlimited growth of their values and to achieve a "smoother" descent. Due to the presence of the regularization term in our loss, and due to the fact that we're involving only sigmoid activation functions, we can state that the gradient of our loss function is Lipschitz continuous. In our project, anyway, for the momentum descent we're not planning to involve a line search strategy but we will rather rely on the hyperparameter optimization considering the step length as a fixed value to be optimized in a small range (trying to keep it as small as possible), or even involving a "decay" (so, keeping reducing it until a specific iteration is reached). For this last reason we don't have the the full theoretical guarantee that it will converge, but in practise small enough values will be selected to achieve a solution that works.

Regarding the convergence rate of the algorithm using the Momentum, citing [9], we can recall the following theorem:

Theorem 4 *Suppose* $\mathscr{E}$ *is a non-convex and L-smooth function,* $\boldsymbol{E}[||\boldsymbol{G}(\boldsymbol{w};\xi) - \nabla\mathscr{E}(\boldsymbol{w})||^2] \leq \delta^2$ *and* $||\nabla\mathscr{E}(\boldsymbol{w})|| \leq \boldsymbol{G}$ *for any* $\boldsymbol{w}$*. After* k *iterations, with* $\boldsymbol{G}(\boldsymbol{w}_k;\xi_k)$*, by setting* $\alpha = min\{\frac{1-\mu}{2L}, \frac{C}{\sqrt{(k+1)}}\}$ *we have*

$$\min_{k=0,...,t} E[||\nabla\mathscr{E}(\boldsymbol{w}_k)||^2] \leq \frac{2(\mathscr{E}(\boldsymbol{w}_0) - \mathscr{E}_*)(1-\mu)}{k+1} max\left\{\frac{2L}{1-\mu}, \frac{\sqrt{k+1}}{C}\right\} +$$

$$+ \frac{C}{\sqrt{k+1}} \frac{L\mu^2((1-\mu)s-1)^2(\boldsymbol{G}^2+\sigma^2) + L\sigma^2(1-\mu)^2}{(1-\mu)^3} \tag{2.5}$$

where $\mathbf{G}$ is the estimate of the true full-batch gradient, w.r.t. the fact that $\boldsymbol{G}_k = \boldsymbol{G}(\boldsymbol{w}_k; \xi_k)$, is a stochastic gradient of $\mathscr{E}(\boldsymbol{w})$ at $\boldsymbol{w}_k$ depending on a random variable ξ_k such that $\boldsymbol{E}[\boldsymbol{G}(\boldsymbol{w}_k; \xi_k)] = \nabla\mathscr{E}(\boldsymbol{w}_k)$
Considering these assumptions and the previous theorem, always citing [9] we can consider a convergence rate of $O(1/\sqrt{k})$.

2.2 Limited-memory quasi-Newton methods

The second derivatives of the objective functions are important because gives us a better understanding of the function topology, that leads to a more efficient descent direction [7]. Their usage it's the basis of the the so-called Newton's method, in which we exploit a second-order Taylor series expansion of the loss function:

$$\Delta\mathscr{E}(\boldsymbol{w}_k) = \mathscr{E}(\boldsymbol{w}_{k+1}) - \mathscr{E}(\boldsymbol{w}_k) \approxeq$$

$$\approxeq \nabla\mathscr{E}_k^T \Delta\boldsymbol{w}_k + \frac{1}{2}\Delta\boldsymbol{w}_k \boldsymbol{H}_k \Delta\boldsymbol{w}_k \tag{2.6}$$

We now wish to minimize the loss function $\mathscr{E}$ with respect to Δw, and this leads to the following update rule that expands the Eq. 1.17:

$$\Delta\boldsymbol{w}_{k+1} = -\alpha_k \boldsymbol{p_k} = -\alpha_k [\boldsymbol{H}_k]^{-1} \nabla\mathscr{E}_k \tag{2.7}$$

where, as stated before, α is a parameter that defines the magnitude of the parameter changing, while $\mathbf{H}$ is the Hessian matrix that's defined as $\nabla^2\mathscr{E}$, and its evaluation requires of course that $\mathscr{E}$ is twice continuously differentiable with respect to the parameters $\boldsymbol{w}$ [4]. Regarding this fact, our loss function defined in Eq. 1.8 it's a composition of twice continuous differentiable activation functions as defined in Sec. 1.2 and so this constraint is satisfied. Under the right assumptions (see Chapter 3, section "Newton's Method" from [6]), the rate of convergence of the Newton's method is *quadratic*. Always citing Haykin's work [4], it's stated that *Newton's method converges quickly asymptotically and does not exhibit the zigzagging behavior that sometimes characterizes the method of steepest descent*, and this is something that we desire in our learning process, because we're invoking a quadratic approximation of the error surface around the current point $\boldsymbol{w}_k$ and supposed that the loss function is quadratic, Newton's method converges to the optimum solution in one iteration [4], while when it's not there's no guarantee for convergence. This method, anyway, has the disadvantage that the Hessian has to be recomputed at each iteration and it's a computationally intensive task: even disregarding its cost, the inverse of the Hessian to be computable requires the fact that $\mathbf{H}$ has to be positive definite, but citing Haykin *there isn't any guarantee that the error surface of the NN always fits this description.*

The solution to this is to exploit *quasi-Newton* methods such as BFGS, in which the inverse of the Hessian is approximated. Furthermore, we'll involve an algorithm of the family of the *Limited-Memory quasi-Newton*, in which the inverse of the Hessian is not only estimated but also in a memory-efficient way by storing a limited amount of information needed to construct its approximation.

Recalling the update rule in Eq. 1.17 latter expanded with Eq. 2.7, we can focus on the estimate of the $\boldsymbol{p}_k$ term that tell us the *direction*. This will be rewritten as:

$$\boldsymbol{p}_k = \boldsymbol{S_k} \nabla\mathscr{E}_k \tag{2.8}$$

where $\boldsymbol{S}_k$ will be a positive definite matrix that's adjusted at each iteration in order to make $\boldsymbol{p}_k$ approximate the classic Newton's direction defined in Eq. 2.7, substituting the inverse of the Hessian with:

$$\boldsymbol{S}_{k+1} = \boldsymbol{V}_k^T \boldsymbol{S}_k \boldsymbol{V}_k + \boldsymbol{\rho}_k \boldsymbol{s}_k \boldsymbol{s}_k^T \tag{2.9}$$

where, taking the equations from [6], we have that:

$$\boldsymbol{\rho}_k = \frac{1}{\boldsymbol{y}_k^T \boldsymbol{s}_k} \tag{2.10}$$

$$\boldsymbol{V}_k = \boldsymbol{I} - \boldsymbol{\rho}_k \boldsymbol{y}_k \boldsymbol{s}_k^T \tag{2.11}$$

$$\boldsymbol{s}_k = \boldsymbol{w}_{k+1} - \boldsymbol{w}_k \tag{2.12}$$

$$\boldsymbol{y}_k = \nabla \mathscr{E}_{k+1} - \nabla \mathscr{E}_k \tag{2.13}$$

At this point, the idea is to involve a small number (m) of pairs $\boldsymbol{s}_k$, $\boldsymbol{y}_k$ that let us evaluate the *direction* $\boldsymbol{p}_k$ (and so the product $\boldsymbol{H}_k \nabla \mathscr{E}_k$) by performing a series of inner products and vector summations involving these two information we're able to easily compute [6]. To what concerns the Eq. 2.9, this is estimated by means of the following in the L-BFGS method:

$$\begin{aligned} \boldsymbol{S}_k = (\boldsymbol{V}_{k-1}^T \cdots \boldsymbol{V}_{k-m}^T) \boldsymbol{S}_k^0 (\boldsymbol{V}_{k-m} \cdots \boldsymbol{V}_{k-1}) \\ + \boldsymbol{\rho}_{k-m} (\boldsymbol{V}_{k-1}^T \cdots \boldsymbol{V}_{k-m+1}^T) \boldsymbol{s}_{k-m} \boldsymbol{s}_{k-m}^T (\boldsymbol{V}_{k-m+1} \cdots \boldsymbol{V}_{k-1}) \\ + \boldsymbol{\rho}_{k-m+1} (\boldsymbol{V}_{k-1}^T \cdots \boldsymbol{V}_{k-m+2}^T) \boldsymbol{s}_{k-m+1} \boldsymbol{s}_{k-m+2}^T (\boldsymbol{V}_{k-m+2} \cdots \boldsymbol{V}_{k-1}) \\ \cdots \\ + \boldsymbol{\rho}_{k-1} \boldsymbol{s}_{k-1} \boldsymbol{s}_{k-1}^T \end{aligned} \tag{2.14}$$

To effectively compute the direction $\boldsymbol{p}_k$ involving the $\boldsymbol{S}$ from the Eq. 2.14, it's possible to follow a two loop algorithm that takes in input the gradient $\nabla \mathscr{E}$. To do so, anyway, we need first initialization of the $\boldsymbol{S}_k^0$. In literature [6] it's suggested to initialize it by considering it as equals to $\boldsymbol{\gamma}_k \boldsymbol{I}$, where $\boldsymbol{\gamma}_k$ is:

$$\boldsymbol{\gamma}_k = \frac{\boldsymbol{s}_{k-1}^T \boldsymbol{y}_{k-1}}{\boldsymbol{y}_{k-1}^T \boldsymbol{y}_{k-1}} \tag{2.15}$$

In practise, we're initializing the approximation of the inverse of the Hessian by involving the latest curvature information, and this happen to be a good heuristic being them the latest research direction of our optimization process.
At this point $\boldsymbol{p}_k$ can be evaluated with Algorithm 5, while the whole algorithm for the optimization is described in Algorithm 6.

Algorithm 5 L-BFGS direction evaluation [6]

$q \leftarrow \nabla \mathscr{E}$
for `i = k-1, k-2, ..., k-m` **do**
    $\alpha_i \leftarrow \rho_i s_i^T q$
    $q \leftarrow q - \alpha_i y_i$
$r \leftarrow S_k^0 \mathrm{q}$
for `i = k-m, k-m+1, ..., k-1` **do**
    $\beta_i \leftarrow \rho_i y_i^T r$
    $r \leftarrow r + s_i(\alpha_i - \beta_i)$
return r

Algorithm 6 L-BFGS [6]

w ← initialize
m ← initialize
k ← 0
while not converged **do**
  $S_k^0 \leftarrow$ initialize
  $p_k \leftarrow S_k \nabla \mathscr{E}_k$
  $w_{k+1} \leftarrow w_k - \alpha_k p_k$, where α has to satisfy the Wolfe conditions
  if k > m **then**
    Discard the vector pair $\{s_{k-m}, y_{y-m}\}$ from storage
  save $s_k \leftarrow w_{k+1} - w_k$ and save $y_k \leftarrow \nabla \mathscr{E}_{k+1} - \nabla \mathscr{E}_k$
  k ← k + 1

The total memory cost for L-BFGS is 2mn +4n, where m defines the amount of pairs of vector to be stored as already said and n is the size of each of them; and from an implementation point of view, we will have a matrix $\boldsymbol{S_k}$ for each of layers present in the NN. With the limited memory quasi-Newton methods, we're neglecting some second-order interactions making the number of the elements of the Hessian proportional to the number of neurons in each layer, instead of the square of the total number of weights in the network. Anyway, even if sparing memory, neglecting these second-order information we will have a less efficient descent direction, which will increase the number of epochs needed to converge [7]. The parameter m is strictly related to the amount of memory implied for the direction estimation and the CPU usage, so a trade-off between memory usage and precision achieved and the CPU must be done. To chose it, being it problem dependent, we plan to involve a grid search across a small set of values (i.e.: {3, 5, 10, 15, 30}), taking into account the fact that when m approaches n (specifically, m > $\frac{n}{2}$) the memory limited approach could be more costly in terms of computer time and storage than the classic BFGS. The parameter α_k will be computed involving a line search method (see Sec. 1.5), ensuring the satisfiability of the Strong Wolfe conditions.

L-BFGS convergence

Let's recall the following theorem from [6]:

Theorem 5 *Suppose that $\mathscr{E} : \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable. Consider the iteration $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha_k \boldsymbol{p}_k$, where $\boldsymbol{p}_k$ is a descent direction and α_k satisfies the Wolfe conditions with $c_1 \leq \frac{1}{2}$. If the sequence $\{\boldsymbol{w}_k\}$ converges to a point $\boldsymbol{w}^*$ such that $\nabla \mathscr{E}(\boldsymbol{w}^*) = 0$ and $\nabla^2 \mathscr{E}(\boldsymbol{w}^*)$ is positive definite, and if the search direction satisfies*

$$\lim_{k \to \infty} \frac{||\nabla \mathscr{E}_k + \nabla^2 \mathscr{E}_k \boldsymbol{p}_k||}{||\boldsymbol{p}_k||} = 0 \tag{2.16}$$

then

1. *the step length $\alpha_k = 1$ is admissible for all k greater than a certain index k_0; and*
2. *if $\alpha_k = 1$ for all $k > k_0, \{\boldsymbol{w}_k\}$ converges to $\boldsymbol{w}^*$ superlinearly.*

Basically, even if $\boldsymbol{B_k}$ doesn't converge to $\nabla^2 \mathscr{E}(\boldsymbol{w}^*)$ we could reach a superlinear convergence rate, assumed that $\boldsymbol{B_k}$ is positive definite (a condition which is already satisfied) and that its accuracy approximation increases to $\nabla^2 \mathscr{E}(\boldsymbol{w}^*)$ progressively.

So, as stated previously, we could combine Eq. 2.16 and Eq. 2.8, to get the following *necessary* and *sufficient* condition for the superlinear convergence:

$$\lim_{k\to\infty} \frac{||(\boldsymbol{S}_k + \nabla^2 \mathscr{E}(\boldsymbol{w}^*))\boldsymbol{p}_k||}{||\boldsymbol{p}_k||} = 0 \tag{2.17}$$

which can be shown to be equivalent to

$$\boldsymbol{p}_k - \boldsymbol{p}_k^N = o(||\boldsymbol{p}_k||) \tag{2.18}$$

Then by combining Eq.2.17 and Eq. 2.18, we obtain:

$$||\boldsymbol{w}_k + \boldsymbol{p}_k - \boldsymbol{w}^*|| \leq ||\boldsymbol{x}_k + \boldsymbol{p}_k^N - \boldsymbol{w} * || + ||\boldsymbol{p}_k - \boldsymbol{p}_k^N|| = O(||\boldsymbol{w}_k - \boldsymbol{w}^*||^2) + o(||\boldsymbol{p}_k||) \tag{2.19}$$

that after some manipulations of the inequalities reveals that $||\boldsymbol{p}_k|| = O(||\boldsymbol{w}_k - \boldsymbol{w}^*||)$, leading to

$$||\boldsymbol{w}_k + \boldsymbol{p}_k - \boldsymbol{w}^*|| \leq o(||\boldsymbol{w}_k - \boldsymbol{w}^*||) \tag{2.20}$$

giving the superlinear convergence result (equations taken from [6]). The previously described properties are not only related to the L-BFGS method but also for any generic quasi-Newton method. On top of this, the condition 2.17 is satisfied by the L-BFGS and therefore we can state that the algorithm can have a superlinear convergence under the proper assumptions.
To what concerns this last point, always following [6], some further assumptions has to be made and has to be verified in sequence:

- due to the fact that our loss function is not convex (being it basically a combination of non convex functions, i.e.: sigmoid), we must say that the sequence $||\boldsymbol{w} - \boldsymbol{w}^*||$ has to converge in a way that
$$\sum_{k=1}^{\infty} ||\boldsymbol{w}_k - \boldsymbol{w}^*|| < \infty \tag{2.21}$$
- *The Hessian matrix $\boldsymbol{G}$ is Lipschitz continuous at $\boldsymbol{w}^*$, that is:*
$$||\boldsymbol{G}(\boldsymbol{w}) - \boldsymbol{G}(\boldsymbol{w}^*)|| \leq L||\boldsymbol{w} - \boldsymbol{w}^*|| \tag{2.22}$$
for all $\boldsymbol{w}$ near $\boldsymbol{w}^*$, where L is a positive constant.

Having ensured these, we can recall from [6] the following theorem:

Theorem 6 *Suppose that f is twice continuously differentiable and that the iterates generated by the BFGS algorithm converge to a minimizer $\boldsymbol{w}^*$, at which the previously assumptions hold. Then $\boldsymbol{w}_k$ converges to $\boldsymbol{w}^*$ at a superlinear rate.*

In our case, we can't verify if the assumptions needed are satisfied, and hence we can't say if we'll have to expect a superlinear convergence rate or not: however, we will investigate the convergence rate from an empirical perspective.

Chapter 3

Experiments

In this chapter we will describe the data we'll use for our experiments, the experimental setup, any configuration and in the end the results.

3.1 Datasets

To compare the optimizers, we chose to involve well-known datasets already used for benchmarking purposes, that are the "MONK's Problem datasets" (from now on MONKs) and the "CUP dataset" (from now on CUP) used in the internal competition of the Machine Learning course. In the specific, these datasets presents two kind of different objectives: the first is a classification problem and the last is a regression problem.

3.1.1 MONKs

MONK is a set of three datasets, namely *Monk1*, *Monk2* and *Monk3*, widely used to assess the capability of Machine Learning models with respect to a classification task. Without entering in the details of the semantics of the dataset, it's made of 432 instances and 6 categorical attributes, plus the category related to the label. Between the 3 MONK there are some slight differences (i.e.: introduction of noise). For these datasets we have performed some *preprocessing steps* we decided to use. Here are the steps we followed in order to reach the final states of our analysis.

1. As first step we encode MONKS datasets with a *1-of-k encoding* to convert categorical data to numerical data and obtaining 17 binary input features.
2. Then a value rescaling has been performed ($0 \rightarrow 0.1$; $1 \rightarrow 0.9$)

3.1.2 CUP

The CUP dataset consist in a set of 20 numerical features and 1765 examples: the task for this dataset is the regression with a bidimensional output. Due to the fact that the test set given in the Machine Learning course is blind, we involved only the training set.

3.2 Experimental setup

For both the optimizer, when experimenting on the same dataset, the topology of the model has been chosen a-priori with a manual exploration. Having set the topology we proceeded with an exploration of the possible hyperparameters.
To what concerns the experiments, those has been executed on a pc AMD Ryzen 5 3600 Processor @ 3.60 GHz with 6-Core, with 16 GB RAM.

The aims of the experiments are to compare the performances of the two optimizers in terms of speed and convergence rate. To what concerns the hyperparameters of both the model and the optimizers, those have been manually explored together with the network topology, fixing them for each dataset.

3.2.1 Stopping criteria

Methods to detect when to stop the optimization process is something widely tackled in literature. For our experiments, we chose the following three criteria:

- *loss threshold*: if the value of the loss computed at an iteration is less than or equal to it, the optimization will stop
- *norm of the gradient threshold*: having set a threshold value, if the norm of the gradient computed at the end of an iteration is less than or equal to it, the optimization will stop
- *max number of iterations*: as last condition to avoid to infinitely trying to reach a minimum

In this way, we can achieve our goal (in terms of loss minimum / norm of the gradient minimum) or stop anyway the optimization after a certain number of iterations.

3.2.2 Comparison features

Regarding the comparison of the optimization strategies, we took into account – for the same dataset – the same fixed Neural Network topology, the same loss threshold, the same norm of the gradient threshold, and the same max number of iterations. As is it known, also the starting point with reference to the weights of the model is an hyperparameter which effects the capability of the optimizers and, ideally, to find a good starting point can make the job easy. The best strategy in terms of effectiveness and simplicity is to sample a random vector for a uniform distribution in a given range: in the specific, we consider the range [-1, +1] for all the experiments.
For the purposes of our comparison, we will consider the followings information:

- f^*: the optimal value found by the optimization process
- $||g_k||$: the norm of the gradient at the end of the optimization process
- *number of iterations* until convergence
- *total time spent to converge*
- *mean time spent for each iteration*

Convergence rate

We're also interested in comparing these algorithm in terms of the rate of convergence. To compute it we consider the following equation:

$$\lim_{k\to\infty} \frac{|f(x_{k+1}) - f^*|}{|f(x_k) - f^*|^p} = R \tag{3.1}$$

where f^* has the same meaning previously expressed and $f(x_k)$ is the optimal value at the iteration k. If there exists $R > 0$, we can say that the algorithm has an order of

convergence p. The convergence rate p is said to be linear if $p = 1$, superlinear if $p > 1$ and quadratic if $p = 2$, and so on. To estimate p, we can use the following:

$$p = \lim_{k\to\infty} \frac{log|f(x_{k+1}) - f^*| - logR}{log|f(x_k) - f^*|} \approx \lim_{k\to\infty} \frac{log|f(x_{k+1}) - f^*|}{log|f(x_k) - f^*|} \tag{3.2}$$

We will show, for each dataset, the comparison of the convergence rate of the two optimizer by means of meaningful plots.

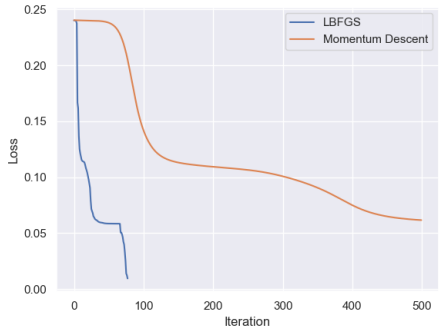
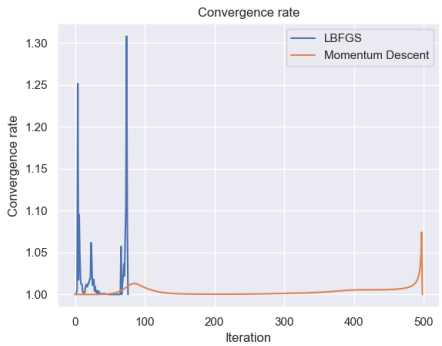
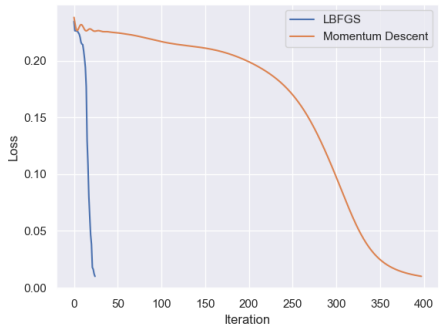
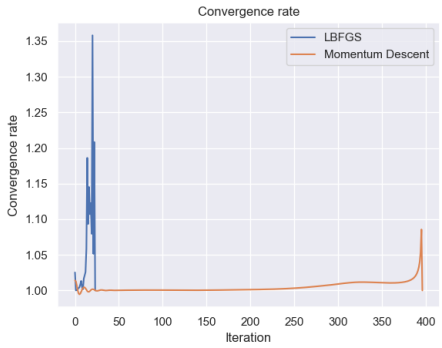
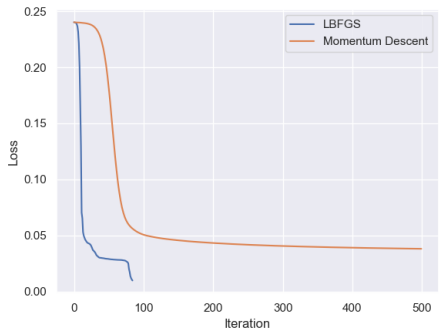
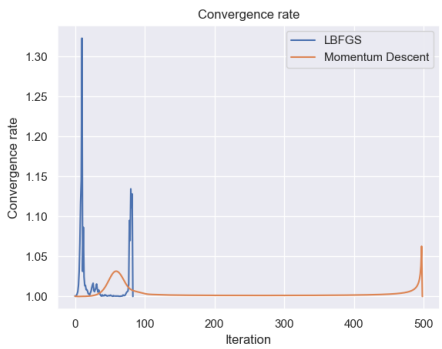
3.3 MONKs

To what concerns the MONK dataset, we used the following hyperparameters:

- Model: 1 input layer with 17 neurons, 1 hidden layer with 4 neurons, 1 output layer with 1 neuron
- Activation functions: sigmoid
- λ: 1e-5
- max number of iteration: 500
- gradient norm threshold: 1e-5
- loss threshold: 0.01

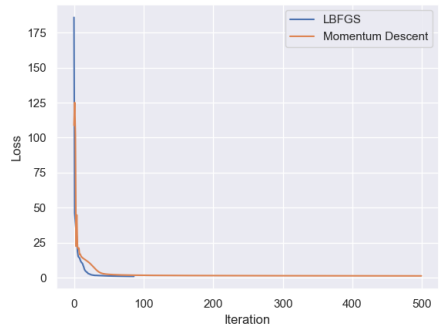
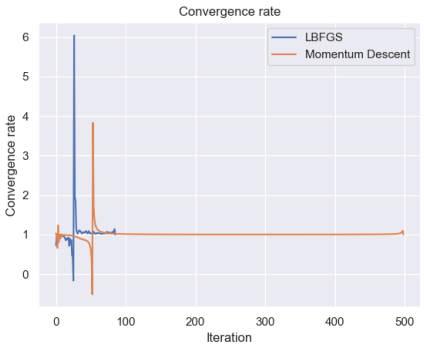
To what concerns the MGD optimizer, we set the step size to 0.9, and the μ parameter of the momentum to 0.9. For the L-BFGS, we used the suggested values for c_1 and c_2 according to [6], setted respectively to $1e-4$ and 0.9. The m parameter of L-BFGS, which states the number of previous s and y to consider, has been chosen by studying the f^* achieved and the number of iteration occurred varying $m \in \{3, 5, 7, 10, 15, 20, 25, 30\}$. According to this, we've chosen 10 for Monk1, 15 for Monk2 and 3 for Monk3.

We can see in the next page a summarizing table showing the results of the experiments on the three MONK dataset. The plots shown are related to the loss functions of the two optimization strategies involved and the rate of convergence.
L-BFGS achieves better results in terms of rate of convergence, with a lot of superlinear peaks. It's almost superlinear, and its behavior (that can be easily seen for example in MONK 2) confirms the theoretical dissertation about its convergence, even if we're in presence of a non-convex function. The Momentum Descent is much more linear, while also having an increasing of the convergence rate in the end. It's possible to see that the $||g_k||$ achieved by the L-BFGS is smaller than the one with the Momentum Descent. To what concerns the speed of convergence, we can see that the L-BFGS is able to achieve the loss threshold in typically less than 100 iterations, while the Momentum Descent keeps going until the maximum number of iteration (reaching also a *plateau* condition) unless in the MONK 2 dataset where it reaches the loss threshold. To what concerns the time spent, we can see clearly that the mean time spent for each iteration is more costly for the L-BFGS, but it can achieve in less number of iteration the desired results, so ending typically previously than the Momentum Descent. This is an interesting result, because its capability to achieve faster a (local) minimum compensate its being costly compared to the Momentum Descent. According to the $f*$ achieved, we can observe graphically from the plots, and numerically by the Table 3.1 that the minima achieved by optimizing with the L-BFGS are smaller than the one achieved with the Momentum Descent, demonstrating the capability of the first optimization algorithm involved both in terms of time and quality of the solution found, confirming the theoretical expectations.

Losses	Convergence rates
MONK 1	MONK 1
	
MONK 2	MONK 2
	
MONK 3	MONK 3
	

dataset	opt	f^*	$\|\|g_k\|\|$	# iterations	total sec	mean sec/it.	stop reason
Monk1	L-BFGS	0.0093	0.0233	77	0.1050	0.0009	f threshold reached
	MGD	0.0615	0.1807	500	0.2762	0.0001	max it. reached
Monk2	L-BFGS	0.0099	0.0035	24	0.0360	0.0008	f threshold reached
	MGD	0.0099	0.4261	397	0.2642	0.0001	f threshold reached
Monk3	L-BFGS	0.0097	0.0058	84	0.1080	0.0007	f threshold reached
	MGD	0.0380	0.0862	500	0.2682	0.0001	max it. reached

Table 3.1: Results on MONKs

Losses	Convergence rates
CUP	CUP
	

opt	f^*	$\|\|g_k\|\|$	# iterations	total sec	mean sec/it.	stop reason
L-BFGS	0.9937	0.2495	86	7.3162	0.0746	f threshold reached
MGD	1.3125	99.1451	500	14.6299	0.0202	max it. reached

Table 3.2: Results on CUP dataset

3.4 CUP

To what concerns the CUP dataset, we used the following hyperparameters:

- Model: 1 input layer with 20 neurons, 2 hidden layer with 100 neurons each, 1 output layer with 2 neuron
- Activation functions: sigmoid for input/hidden layers and linear for the output layer
- λ: 1e-4
- max number of iteration: 500
- gradient norm threshold: 1e-5
- loss threshold: 0.01

To what concerns the MGD optimizer, we set the step size to 0.06, and the μ parameter of the momentum to 0.6. For the L-BFGS, we used the suggested values for c_1 and c_2 according to [6], setted respectively to $1e-4$ and 0.9. The m parameter of L-BFGS, which states the number of previous s and y to consider, has been chosen by studying the f^* achieved and the number of iteration occurred varying $m \in \{3, 5, 7, 10, 15, 20, 25, 30\}$. According to this, we've chosen 25.

As we can see from the results in Tab. 3.2 and relative plots, the L-BFGS after a peak of the convergence rate it keeps being superlinear. This let us achieve the stop condition of the loss minimized below the threshold after 86 iterations of the optimization algorithm. The convergence rate of the Momentum Descent, despite a first superlinear peak, keeps being almost linear. In the end, the Momentum Descent is not able to achieve the same quality of the f^* found with respect to the one found by L-BFGS. As for the MONKs experiments, the $||g_k||$ observed at the end of the optimization process with the Momentum Descent is bigger than the one observable in the L-BFGS case. The mean iteration cost of L-BFGS is more than twice the one achieved by the Momentum Descent, but being the number of iteration smaller it's possible, in fact, to converge to a solution in half of the time that the Momentum Descent needs, as we can see from Table 3.2.

m	f *	||g_k||	# iterations	total time (s)	mean time per it. (s)	stop reason
3	0.009317	0.017387	112	0.142335	0.000922	f threshold reached
5	0.009398	0.010829	68	0.100690	0.001115	f threshold reached
7	0.008939	0.005282	121	0.166857	0.001036	f threshold reached
10	0.007829	0.006971	53	0.088047	0.001282	f threshold reached
15	0.009399	0.006646	73	0.121454	0.001302	f threshold reached
20	0.009842	0.002543	69	0.119888	0.001369	f threshold reached
25	0.009046	0.013704	124	0.201051	0.001265	f threshold reached
30	0.009675	0.003775	36	0.064213	0.001360	f threshold reached

Table 3.3: Results varying m on MONK 1

m	f *	||g_k||	# iterations	total time (s)	mean time per it. (s)	stop reason
3	0.993596	0.693900	142	34.692169	0.210517	f threshold reached
5	0.999889	0.284015	129	34.363672	0.229802	f threshold reached
7	0.997601	0.344468	108	26.693757	0.213679	f threshold reached
10	0.996379	0.563552	87	24.275142	0.242305	f threshold reached
15	0.998135	0.284607	85	22.013743	0.222510	f threshold reached
20	0.995864	0.301241	73	22.207128	0.260736	f threshold reached
25	0.998908	0.288784	67	22.199533	0.284502	f threshold reached
30	0.999584	0.335335	66	19.322641	0.250501	f threshold reached

Table 3.4: Results varying m on CUP

3.5 On the impact of the m parameter

As stated in the previous sections, we proceeded by testing different values of $m \in \{3, 5, 7, 10, 15, 20, 25, 30\}$.
Here we present some results in Tab 3.4 and 3.3. The first interesting thing that we can observe is that for increasing m the mean time per iteration tends to increase. This is obvious due to the mole of information to be took into account computing the direction. This trend can be seen on both the datasets took into account in this analysis. The total amount of time spent to achieve a minimum follows, instead, the inverse trend: the more m is high, the less we spent to converge (at the cost of a cost from the point of view of memory usage). To what concerns the number of iteration needed, we can see that more information we use, the less we need to iterate. While needing to chose the best m in our experiments, we tried to consider both the number of iterations needed and the f^* achieved to find a compromise between results and speed.

Chapter 4

Conclusions

This work let us investigate and experiment two optimization approaches involving the L-BFGS method and the Momentum Descent. We analyzed theoretical aspects of both, applying both methods on four differents dataset to understand their convergence behavior. As a result, we have observed the interesting qualities of the L-BFGS even in the presence of non-convex cases, demonstrating empirically its strengths in terms of effectiveness of the solution found and the number of iteration needed, while confirming the well known capability of the Momentum Descent that even if not able (with the current hyperparameters involved) to achieve the same results, keeps being a low cost optimization algorithm in terms of mean amount of time spent during its iterations.

Bibliography

[1] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989.

[2] Kenneth Eriksson, Donald Estep, and Claes Johnson. *Applied Mathematics: Body and Soul*. Springer Berlin Heidelberg, 2004.

[3] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[4] S.S. Haykin. *Neural Networks and Learning Machines*. Pearson International Edition. Pearson, 2009.

[5] Lam M. Nguyen, Phuong Ha Nguyen, Peter Richtárik, Katya Scheinberg, Martin Takáč, and Marten van Dijk. New convergence aspects of stochastic gradient algorithms, 2019.

[6] Jorge Nocedal and Stephen Wright. *Numerical Optimization*. Springer Science & Business Media, 2006.

[7] B. Robitaille, B. Marcos, M. Veillette, and G. Payre. Modified quasi-newton methods for training neural networks. *Computers & Chemical Engineering*, 20(9):1133–1140, 1996.

[8] Kevin Scaman and Aladin Virmaux. Lipschitz regularity of deep neural networks: analysis and efficient estimation, 2019.

[9] Tianbao Yang, Qihang Lin, and Zhe Li. Unified convergence analysis of stochastic momentum methods for convex and non-convex optimization, 2016.