Computational Mathematics for Learning and Data Analysis

Implementation of a Neural Network optimized through Stochastic Gradient Descent and Conjugate Gradient Descent

Sabrina Briganti - 465214 - sabrinabriganti@gmail.com
Gianmarco Ricciarelli - 555396 - gianmarcoricciarelli@gmail.com

April 30, 2019

Contents

Chapter 1

The Artificial Neural Network

In this first Chapter, we provide some informations about the Artificial Neural Network, i.e. a fully connected Multilayer Perceptron, we implemented from scratch. We'll describe both the network's structure and the algorithm we used in order to make our network *learn* from the data used during the testing and validation phases. Finally we'll present the loss function we have chosen for our network, and we'll provide and explanation on how it is differentiable. We'll use the notation proposed in [9].

1.1 The ANN's structure

Since we have to write from scratch an *Artificial Neural Network*, ANN for short, we have considered some alternatives before choosing the network's final structure. We agreed on a structure composed by:

- one *input layer*;
- two *hidden layers*;
- one *output layer*;

As convention, the number of units in the input layer is egual to the number of features of the dataset that is used for the learning, validation and testing phases. The two hidden layers contain, respectively, four and eight *hidden neurons*, following the convention of putting an increasing series of powers of two as number of hidden units per layer. The number of neurons for the output layer depends on the kind of task the network is trying to fullfil. In the case of a *classification task*, like the MONKS dataset [5], we have decided to put one unit in the output layer, while in the case of a *regression task*, like the CUP dataset, we have decided to put two units in the output layer. As we have seen studying the papers and books for gathering the necessary knowledge for the project, as [9, 11, 14], choosing to consider the network's structure as an *hyperparameter*, that is, a variable, could lead to a series of difficult choices during the validation phase, so we have decided to fix the ANN structure to the one described for both the task we have to fullfil, changing only the number of units in the output layer from task to task. In figure 1.1 you can see our ANN's graph for the classification task.

1.2 Initializing the Network

As we know from [9, 11, 14], an ANN is composed by a set of weights $\mathbf{W}_i$, and a set of biases $\mathbf{b}_i$, $i \in \{1, \dots, l\}$, with l representing the ANN's number of layers. Although it is common practice to initialized the network's weights and biases to random, small, values,

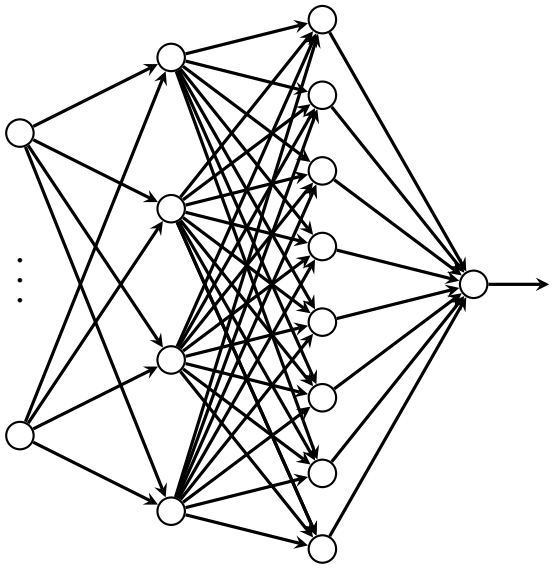


Figure 1.1: The ANN's structure for the classification task, the majority of the input nodes are omitted because they vary from dataset to dataset.

we have decided to follow the *normalized initialization*, as described in [8, 9], which defines the initial values for the weights and the biases for each layer in the uniform distribution taken in the range

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{m+n}},\ \frac{\sqrt{6}}{\sqrt{m+n}}\right]$$

with m and n, representing the number of inputs and outputs for each layer. This heuristic is designed to compromise between the goal of initializing all layers to have the same activation variance and the goal of initializing all layers to have the same gradient variance. The formula is derived using the assumption that the network consists only of a chain of matrix multiplications, with no nonlinearities. Other than this kind of initialization, we also make available the standard *random initialization* for creating a network, described at the beginning of this §, which initialize the weights and the biases for each layer in the uniform distribution taken in the range

$$W \sim U\left[-0.7,\ 0.7\right].$$

1.3 The back-propagation algorithm

The learning procedure for our ANN essentialy consist in two distinct phases:

1. compute the network's *gradient*, that is, the derivative of the cost function $\nabla_\theta J(\theta)$, with θ representing the ANN's hyperparameters, with respect to every network's unit using the well known *back-propagation algorithm*, described by algorithm 1 and 2;

2. optimize the information gathered during the first phase using a distinct optimizer, chosen among the *Stochastic Gradient Descent* and the *Conjugate Gradient Method*, as described in Chapter 2;

For computing the gradient we have chosen to use the well known *backpropagation algorithm*, firstly introduced in [20] and described in [9, 11, 14]. This algorithm is also

composed by two phases, a first phase, that is, the *forward propagation*, in which the feature vector $\mathbf{x}$ given in input has to flow from the input layer through the hidden layers and, finally, the output layer, giving the approximation $\hat{\mathbf{y}}$ as output, and a second one, that is, the *back-propagation*, which allows the information to flow backward through the network in order to compute the gradient by applying the Chain Rule of Calculus, that is, a technique for computing the derivative of a composition of functions.

Algorithm 1 Forward propagation through a typical (deep) neural network and the computation of the cost function. Here $L(\hat{\mathbf{y}}, \mathbf{y})$ represents the loss function evaluated using both $\mathbf{y}$ and $\hat{\mathbf{y}}$ as inputs, more details about that will be provided in §1.5. The function f applied on line 5 represents the layer's *activation function*, while $\lambda\Omega(\theta)$ represents the network's regularization term, with θ representing the ANN's hyperparameters.

1: **procedure** FORWARD PROPAGATION(l, $\mathbf{W}_i$ $i \in \{1, \dots, l\}$, $\mathbf{b}_i$ $i \in \{1, \dots, l\}$, $\mathbf{x}$, $\mathbf{y}$)
2: $\quad \mathbf{h}_0 = \mathbf{x}$
3: $\quad$ **for** $k = 1, \dots, l$ **do**
4: $\quad\quad \mathbf{a}_k = \mathbf{b}_k + \mathbf{W}_k \mathbf{h}_{k-1}$
5: $\quad\quad \mathbf{h}_k = f(\mathbf{a}_k)$
6: $\quad \hat{\mathbf{y}} = \mathbf{h}_l$
7: $\quad J = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda\Omega(\theta)$

Since each one of the ANN's layers has its own *activation function*, that is, a function that has to be applyied to the output of every layer's neuron, it is particularly usefull to think of the ANN like a composition of functions, and, for this reason, the Chain Rule of Calculus play a decisive role in the gradient's computation by back-propagation. It is import to note that with the term back-propagation we mean only the method for computing the gradient, not the whole learning algorithm. We now provide the pseudocode for the forward propagation and the back-propagation phases, as shown in algorithms 1 and 2.

Algorithm 2 Backward computation for the (deep) neural network of algorithm 1. Here, the $\odot$ symbol represents the element-wise (Hadamard) product, while $\nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y})$ represents the gradient of the loss function computed with respect to the output $\hat{\mathbf{y}}$. $\nabla_{\mathbf{b}_k} J$, $\nabla_{\mathbf{W}_k} J$ and $\nabla_{\mathbf{h}_{k-1}} J$ represents the gradient of the loss function computed with respect to, respectively, $\mathbf{b}_k$, $\mathbf{W}_k$ and $\mathbf{h}_{k-1}$, and finally $\nabla_{\mathbf{b}_k}\Omega(\theta)$ and $\nabla_{\mathbf{W}_k}\Omega(\theta)$ represents the gradient of the ANN's hyperparameters computed with respect to, respectively, $\mathbf{b}_k$ and $\mathbf{W}_k$.

1: **procedure** BACKWARD PROPAGATION
2: $\quad \mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y})$
3: $\quad$ **for** $k = l, l-1, \dots, 1$ **do**
4: $\quad\quad \mathbf{g} \leftarrow \nabla_{\mathbf{a}_k} J = \mathbf{g} \odot f'(\mathbf{a}_k)$
5: $\quad\quad \nabla_{\mathbf{b}_k} J = \mathbf{g} + \lambda \nabla_{\mathbf{b}_k}\Omega(\theta)$
6: $\quad\quad \nabla_{\mathbf{W}_k} J = \mathbf{g}\mathbf{h}_{k-1}^T + \lambda \nabla_{\mathbf{W}_k}\Omega(\theta)$
7: $\quad\quad \mathbf{g} = \nabla_{\mathbf{h}_{k-1}} J = \mathbf{W}_k^T \mathbf{g}$

1.4 The activation functions

As we have mentioned in §1.3, each one of the ANN's layers has an *activation function*, that is, a function that is applied to $\mathbf{a}_k$ in order to obtain $\mathbf{h}_k$, with $k \in \{1, \dots, l\}$. We can say that an activation function of a node defines the output of that node, and maps $\mathbf{a}_k$ into the desired range. For being chosen as an activation function, a function has to possess a series of properties, such as

- *nonlinearity*: when the activation function is non-linear, then a two-layer neural network can be proven to be a universal function approximator;
- *range*: when the range of the activation function is finite, gradient-based training methods, such as the ones described in chapter 2, tend to be more stable, because pattern presentations significantly affect only limited weights;
- *continuously differentiable*: since the functions have to be envolved in the Chain Rule of Calculus during the gradient's computation in the back-propagation algorithm;

There are many functions that can be used as activation functions for an ANN; we have chosen to utilize the well known *logistic function*, i.e. the *sigmoid function*, which is computed as

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$
$$f'(x) = \sigma'(x) = f(x)(1 - f(x)),$$

and is defined in the range (0, 1). As introduced in §1.3, we can think the ANN as a *composition* of activation functions, that is, representing it as

$$\sigma_l \circ \sigma_{l-1} \circ \cdots \circ \sigma_1$$

with l representing the total number of layers in the network and σ representing the logistic function.

1.5 The Loss function

It §1.3 we have referred to the *Loss function* as to a function that takes as input the ANN's output vector $\hat{\mathbf{y}}$ and the *ground truth* vector $\mathbf{y}$, that is, the vector containing the desired output for the network. But what essentially is a Loss function? As a matter of fact, the Loss function can be considered like one way of measuring the performance, or equivalently the error, of the model that utilizes it, an ANN in this case. There are various types of Loss functions, for our network we have decided to use a well known function: the *Mean Squared Error*, MSE for short, for both the classification and the regression tasks. The MSE is obtained with the formula

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^{m} (\hat{\mathbf{y}} - \mathbf{y})_i^2 ,$$

where m represents the number of samples for which we computed the output $\hat{\mathbf{y}}$. We can represent this functions also as a composition of the square function and the Euclidean norm by writing the latter formula as

$$\text{MSE} = \frac{1}{m} ||\hat{\mathbf{y}} - \mathbf{y}||_2^2.$$

Since this formula is used for computing the performances of our models, our primary goal is to minimize its output as much as possible by computing the gradient during the backward propagation step of the back-propagation algorithm, as described in §1.3. Of course, in order to do so, it must be a *differentiable* function. The gradient for the MSE with respect to $\hat{\mathbf{y}}$ is defined as:

$$\nabla_{\hat{\mathbf{y}}} \text{MSE}\,(\hat{\mathbf{y}},\ \mathbf{y}) = \hat{\mathbf{y}} - \mathbf{y}.$$

1.5.1 Properties of the Loss function

Here we discuss the relevant properties of the Loss function we have chosen for our model.

- **Continuity**. As we introduced in this section, our Loss function is a function in two variables, $\hat{\mathbf{y}}$ and $\mathbf{y}$, respectively. A function f in two variables is L-Lipschitz continuous if there exists a constant L such that

  $$||f(\mathbf{x}_1, \mathbf{y}_1) - f(\mathbf{x}_2, \mathbf{y}_2)|| \leq L(||\mathbf{x}_1 - \mathbf{x}_2|| + ||\mathbf{y}_1 - \mathbf{y}_2||) \quad \forall (\mathbf{x}_1, \mathbf{y_1}), (\mathbf{x}_2, \mathbf{y_2}) \in \textbf{dom } f.$$

  Being the MSE a composition of the square function and the Euclidean norm, because of the squaring, the function is Lipschitz continuous only if we restrict its domain to a bounded set. Since, as described in §1.4, each network's layer uses the sigmoid function, which is a squashing function that bounds every layer's output in the range $(0,\ 1)$, the MSE can be considered as a continous function.

- **Differentiability**. Again, being the sigmoid function the one used by the ANN, and since the sigmoid function is continuously differentiable with bounded Lipschitz continuous derivative, we can consider the MSE as a differentiable function, being the network a composition of continuously differentiable functions.

- **Convexity**. We recall that, for $h : \mathbb{R}^k \to \mathbb{R}$, $g_i : \mathbb{R}^n \to \mathbb{R}$, the function:

  $$f(\mathbf{x}) = h(g_1(\mathbf{x}), \ldots, g_k(\mathbf{x}))$$

  is convex if $\textbf{dom } h = \mathbb{R}^k$, $\textbf{dom } g = \mathbb{R}^n$, h is convex, h is non-decreasing in each argument and g_i are convex. For the aforesaid result, the squared Euclidean distance is convex, as any distance is non-negative and the square function is convex and increasing for non-negative values; however, when composed with another function it may not be convex. The functions that compose the network, that is, the sigmoid functions, are not convex functions, hence we can conclude that, for our network, the MSE is a not convex function.

Chapter 2

Optimizers

2.1 Stochastic Gradient Descent

When choosing an optimizer, the *Stochastic Gradient Descent*, SGD for short, is a quite common choice. It is not the best though, since, as proved by the last developments in the machine learning field, its convergence's rate is quite slow. Said that, it is also true that it allows to finds a very low value of the cost function quickly enough.
The algorithm 3 shows the standard SGD version implemented, as described in [9], supporting both *momentum* and *regularization*. [13]
SGD is an extension of the *Gradient Descent Algorithm* (GD). It is an iterative first-order optimization technique, usefull to minimize the objective Loss function.
The main goal is, indeed, to minimize the Loss function, in order to obtain a neural network with good generalization property:

$$\min_{\mathbf{W} \in \mathbb{R}^n} \quad L(\mathbf{W}), \tag{2.1}$$

where L is the Loss function and $\mathbf{W}$ are the synaptic weights of the network.
The way to achieve this result, is to identify and compute a local minimum, moving along the direction of the steepest descent of the function, that is the negative gradient $-\nabla L(\mathbf{W})$.
Of course, in order to find a local minimum, it is necessary to update the synaptic weights $\mathbf{W}$ at each iteration as $\mathbf{W} = \mathbf{W} + \Delta\mathbf{W}$.
The correction applied to the each one of the weights is defined as follows, taking a step in the opposite direction of the cost gradient:

$$\Delta w_{ji} = -\eta \frac{\partial L}{\partial w_{ji}}, \tag{2.2}$$

where η is the learning rate. The latter one is a fundamental hyperparameter which has to been chosen wisely, since it represents how much is right to move along the descent direction: too much and the procedure will be vain, missing the minimum; too little and the converge will be slow.
Unlike the GD algorithm, that could get a longer time to converges to the minimum because of the potentially big number of training examples, the SGD algorithm requires only the evaluation of one example for epochs (on-line mode).
Here, however, we refer to SGD even when using the entire or just a subset of training dataset (batch or mini batch).
The name stochastic derives from the fact that the samples are randomly selected, bringing to an approximation of the true gradient, estimated using a small set of samples. This is also the reason of the typical zig-zag pattern in the path towards the minimum of the Loss function, as visible in Fig.2.1.

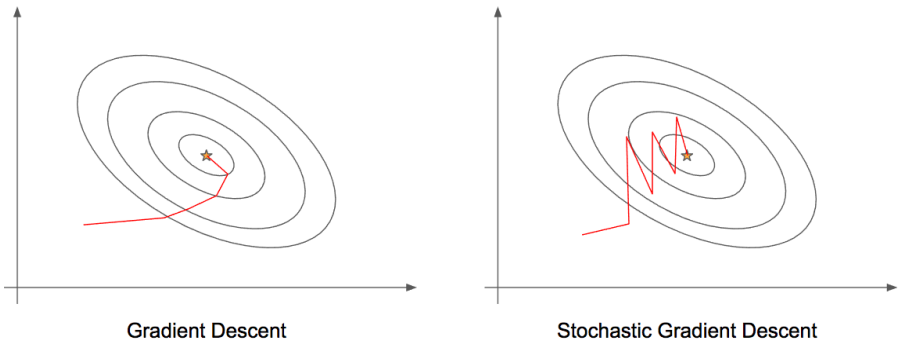


Figure 2.1: The different trajectories from GD and SGD.

For what concernes the convergence of the stochastic gradient descent algorithm, it depends on the choice of η: it is necessary to gradually decrease the learning rate over the epochs for ensure convergence. A sufficient condition to guarantee convergence of SGD is that the sequence of decreasing learning rates satisfy:

$$\sum_{t=1}^{\infty} \eta_t = \infty \quad \text{and} \quad \sum_{t=1}^{\infty} \eta_t^2 < \infty \tag{2.3}$$

Futhermore, if we assume that the Hessian matrix of the Loss function is strictly positive definite at the optimum, which means that the Loss function is strogly convex, the convergence rate is $O(\frac{1}{k})$, with k as the number of epochs in the training. Otherwise, relaxing this assumption in presence of a convex problem, the convergence rate becomes $O(\frac{1}{\sqrt{k}})$[9, 15, 21].
Unfortunatly, as we showed in §1.5.1, our objective function is not convex, so we can't rely on what has been written before. Anyway, this issue seems to be overcomed: as proved in [2], the SGD is convergent even in presence of nonconvex functions. If the algorithm is run with a stepsize sequence satisfying Eq.2.3 and the following assumptions hold:

Assumption 1 - *If the level set $\mathcal{L} = \{w : L(\boldsymbol{W}) \leq L(\boldsymbol{W}_1)\}$ is bounded, in some neighborhood $\mathcal{N}$ of $\mathcal{L}$ the objective function L is continuously differentiable, and its gradient is Lipschitz continuous, there exist a constant $B > 0$ s.t. $\|\nabla L(\boldsymbol{W}) - \nabla L(\widetilde{\boldsymbol{W}})\| \leq B\|\boldsymbol{W} - \widetilde{\boldsymbol{W}}\|, \forall \boldsymbol{W}, \widetilde{\boldsymbol{W}} \in \mathcal{N}$;*

Assumption 2 - *The Loss function L and SGD satisfy:*

- *The sequence of weights $\boldsymbol{W}_k$ is contained in an open set over which L is bounded below by a scalar value L_{inf}, that means the function is bounded below over the region explored by the algorithm;*
- *There exist scalars $\mu_G \geq \mu > 0$ s.t., $\forall k \in \mathbb{N}$,*

$$\nabla L(\boldsymbol{W}_k)^T \mathbb{E}_{\xi_k}[\boldsymbol{g}(\boldsymbol{W}_k, \xi_k)] \geq \mu \|\nabla L(\boldsymbol{W}_k)\|^2 \text{ and} \tag{2.4}$$

$$\|\mathbb{E}_{\xi_k}[\boldsymbol{g}(\boldsymbol{W}_k, \xi_k)]\| \leq \mu_G \|\nabla L(\boldsymbol{W}_k)\|, \tag{2.5}$$

  that means the vector $\boldsymbol{g}$ (the extimate of the real gradient vector) is a direction of sufficient descent for L with a norm comparable to the norm of the gradient;

- *There exist scalars $M \geq 0$ and $M_G \geq 0$ s.t.:*

$$\|\mathbb{V}_{\xi_k}[\boldsymbol{g}(\boldsymbol{W}_k, \xi_k)]\| \leq M + M_G \nabla L(\boldsymbol{W}_k)\|^2, \forall k \in \mathbb{N}, \tag{2.6}$$

  that restricts the variance of $\boldsymbol{g}$.

then the following property is garanteed: $\lim_{k\to\infty} \mathbb{E}[\|\nabla L(\mathbf{W}_k)\|^2] = 0$, that is the convergence of the algorithm.
A property of SGD is that the computation time per update does not grow with the number of training examples, allowing convergence even in presence of a large dataset.

Algorithm 3 Stochastic Gradient Descent Algorithm. The learning rate η, the α term and the maximum number of epochs are given.

procedure STOCHASTIC GRADIENT DESCENT
  Initialize $\mathbf{W}$ and $\mathbf{v}$
  $k \leftarrow 0$
  while $k < max\_epochs$ **do**
    Sample a minibatch of m training examples $\{(x_0, y_0), (x_1, y_1), ..., (x_m, y_m)\}$
    if Nesterov Momentum **then**
      $\mathbf{W} \leftarrow \mathbf{W} + \alpha\mathbf{v}$
    Compute gradient estimate: $\mathbf{g} \leftarrow \frac{1}{m}\nabla \sum_i L(\mathbf{W})$
    Compute velocity update: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta\mathbf{g}$
    Apply update: $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$

2.1.1 Momentum

When computing the adjustment of the synaptic weights $\Delta\mathbf{W}$ as in Eq. 2.3, the choice of the learning rate η influences the convergence of the SGD algorithm. The smaller is η, the smaller will be the changes in the matrix of weights and the rate of learning, but the smoother will be the trajectory.
On the contrary, a bigger η will bring to a faster convergence, but also to an oscillatory behaviour.
A way to accelerate the SGD is the use of the *Classical Momentum* (CM), a first order optimization technique which accelerate gradient descent, and so the learning rate of the final training.
It consists in the adjustment of the new weigths through a velocity vector $\mathbf{v}$ that accumulates the gradient elements in the directions of reduction of the Loss function, and in a momentum coefficient $\alpha \in [0, 1]$: the larger is α, the more the previous gradients affect the current direction.
In this case, the classical momentum is given by:

$$\mathbf{v}_k = \alpha\mathbf{v}_{k-1} + \eta\nabla L(\mathbf{W}_k). \tag{2.7}$$

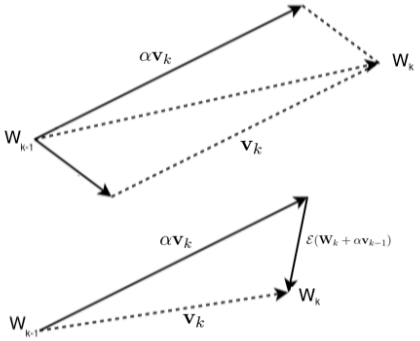
The new synaptic weights are then updated as:

$$\mathbf{W}_k = \mathbf{W}_{k-1} + \mathbf{v}_k. \tag{2.8}$$

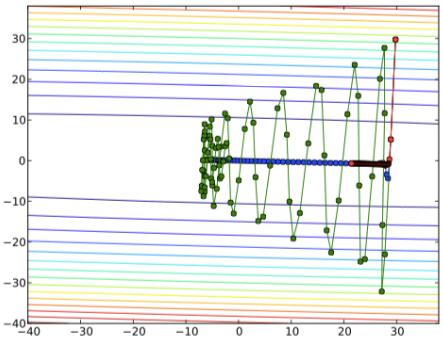
A variant of the CM algorithm, is the *Nesterov's Accelerated Gradient* (NAG), which allows to avoid the oscillatory behaviour in the trajectory computed with CM, as visible in Fig.2.2b
The velocity vector $\mathbf{v}$ is computed as:

$$\mathbf{v}_k = \alpha\mathbf{v}_{k-1} + \eta\nabla L(\mathbf{W}_k + \alpha\mathbf{v}_{k-1}). \tag{2.9}$$

The update of the weights $\mathbf{W}$ follows the one described in Eq. 2.8.
The variant respect Eq. 2.7, shown in Fig.2.2a, is given by the fact that the gradient ∇L is evaluated after the current velocity is applied: NAG first update the $\mathbf{W}_k$, making a jump in the direction of the previous accumulated gradient, and then evaluate the gradient in



(a) The classical momentum on top and the Nesterov Accelerated gradient on bottom.



(b) The trajectory from CM (in green) and the one from NAG (in blue).

that point and makes a correction. This procedure allows it to change velocity vector $\mathbf{v}_k$ in a faster way.
It's worth to underline that, in case of batch mode and convex functions, Nesterov momentum brings the rate of convergence from $O(\frac{1}{k})$ (after k steps) to $O(\frac{1}{k^2})$[17].

2.1.2 Regularization

In order to garantee a tradeoff between goodness and complexity of the model, the regularization is allowed in the network. This choice is important to ensure that the model doesn't grow too much in complexity.
Regularization, in fact, adds a penalty as the model's complexity increases, forcing some weights to take values close to zero, if they have little influence on the newtork performance and so resulting in poor generalization.
It basically modifies the objective Loss functions as follows:

$$\min_{\mathbf{W} \in \mathbb{R}^n} \quad L(\mathbf{W}) + \lambda \Omega(\mathbf{W}) \tag{2.10}$$

where $\Omega(\mathbf{W})$ is a complexity penalty term based of the weights, and λ is a parameter which tells how much importance must have the complexity penalty term.
Two kinds of regularization are tipically used:

- *L1*: also called Lasso Regression, which results in $\Omega(\mathbf{W}) = \sum_{i=1}^{k} |w_i| = \|\mathbf{W}\|_1$;
- *L2*: also known as Ridge Regression, which results in $\Omega(\mathbf{W}) = \sum_{i=1}^{k} w_i^2 = \|\mathbf{W}\|_2^2$.

The addition of squared terms to the Loss function of Eq.1.5 in the Ridge Regression, returns again a smooth continous and differentiable function.
However, the Lasso penalty, which pushes several elements of $\mathbf{W}$ to be exactly zero, making the result a sparse parameter vector, is not differentiable. So, since when applied to the Loss function, it returns a non-smooth function, it hasn't been tested on SGD[22].

2.2 Nonlinear Conjugate Gradient

An intresting optimization ables to lead to an improvement of the performances of the neural network, is the use of high-order information during the training phase: this brings to a more accurate choice of the search direction and of the step size, by using information from the second order approximation.

In order to avoid the expensive computation of the inverse of the Hessian, we can use the *Conjugate Gradient* methods, which are a class of iterative second-order optimization methods, derived from the steepest-descent algorithm, that ensure low memory requirements.
In this way, the adjustment to the synaptic weights of the network is computed as:

$$\Delta \mathbf{W} = \alpha \mathbf{d}, \tag{2.11}$$

where α is the learning rate and $\mathbf{d}$ is the new direction found.
In our case, the nonlinear conjugate gradient methods are designed to solve the minimization problem defined in Eq.2.1.
As showed in the pseudocode 4, the iterative formula generates a sequence of weights $\{W_k\}$, for every epoch of training k, as:

$$\mathbf{W}_{k+1} = \mathbf{W}_k + \alpha_k \mathbf{d}_k, \quad k = 0, 1, ..., \tag{2.12}$$

where α_k is a learning rate and $\mathbf{d}_k$ is a descent direction. These are the new synaptic weights computed with the adjustment of Eq.2.11.

Algorithm 4 Nonlinear Conjugate Gradient Algorithm. The maximum number of epochs and the tolerance are given.

procedure NONLINEAR CONJUGATE GRADIENT
  Initialize $\mathbf{W}_0$ and L_g
  $k \leftarrow 0$, $g_0 \leftarrow 0$
  while $k < max\_epochs$ **do**
    Evaluate the Loss function L_k and its gradient $\mathbf{g}_k$
    if $L_k < L_g$ or $\|\mathbf{g}_k\| > tolerance$ **then**
      return Error goal reached
    Compute the β with one of the methods HS, MHS, FR, PR
    Compute the direction: $\mathbf{d}_k \leftarrow -\mathbf{g}_k + \beta \mathbf{d}_k$
    Compute the learning rate α_k with a Line Search
    Update the weights: $\mathbf{W}_{k+1} \leftarrow \mathbf{W}_k + \alpha_k \mathbf{d}_k$
    $k \leftarrow k + 1$

2.2.1 Search Direction

The direction $\mathbf{d}_k$ holds the sequent property:

$$\mathbf{d}_k^T \mathbf{H} \mathbf{d}_{tk-1} = 0, \tag{2.13}$$

that means it is conjugate to the previous direction $\mathbf{d}_{k-1}$. Furthermore, it doesn't need to know all the previous directions, but it only needs the last one, which is why it requires very little storage and computation.
When dealing with quadratic functions, this method keeps the progress obtained so far in the minimization of the Loss function, by ensuring that the gradient along the previous direction does not increase. Anyway, it's worth to underline that this method can also be applyed with nonlinear functions: in this case, it should be necessary to restart the process, since there is no assumption that the conjugate directions previously found are still at the minimum of the function.
Each new direction it's a linear combination of the steepest descent -**g** and the previous direction $\mathbf{d}_{k-1}$, and it is defined as:

$$\mathbf{d}_k = \begin{cases} -\mathbf{g}_0, & \text{if } k = 0; \\ -\mathbf{g}_k + \beta_k \mathbf{d}_{k-1}, & \text{otherwise,} \end{cases} \tag{2.14}$$

where β_k is a scalar, to be determined, that says how much of the previous direction should be added to the newest one. When applied to minimize a strictly convex quadratic function, it ensure that the directions $\mathbf{d}_k$ and $\mathbf{d}_{k-1}$ are conjugate with respective to the Hessian of the objective function, that is the property 2.2.1 holds.
Of course, the first search direction when $k = 0$ is defined as the steepest descent direction at the initial weight $\mathbf{W}_0$ while, for $k > 1$, a minimization along each of the search direction is performed.
Since it may be that the direction found is not a descent direction of the objective function, another modified search direction, proposed by Zang et al.[12], has been tested in the project. It ensures sufficient descent $g_k^T = -\|g_k\|^2$, indipendent of the line search used or the convexity of the objective function, and is defined as follows:

$$\mathbf{d}_k^+ = \begin{cases} -\mathbf{g}_0, & \text{if } k = 0; \\ -(1 + \beta_k \frac{\mathbf{g}_k^T \mathbf{d}_k}{\|\mathbf{g}_k\|})\mathbf{g}_k + \beta_k \mathbf{d}_{k-1}^+, & \text{otherwise.} \end{cases} \tag{2.15}$$

2.2.2 Beta

What really makes the difference in the computation of the conjugate gradient algorithm, is the choice of the method used to compute the β coefficient.
In fact, there has been proposed various choices for computing it, each one giving different efficiency and properties.
The formulas tested in our implementation are three: the Polak-Ribierère (PR), the Hestenes-Stiefel (HS) and a Modified Hestenes-Stiefel (MHS^+).
One of the properties that must be garanteed, is the global convergence of the method. When the function to be minimized is convex and quadratic, indeed, the Conjugate Gradient algorithm ensures the convergence and the detection of the global minimum in at most N iterations, that is the number of dimensions. Often, expecially when N is very large, there are great chances that the algorithm terminates in less than N epochs. [4]. However, since in our network we are dealing with a nonquadratic Loss function, as showed in §1.5.1, the direction computed as in Eq. 2.14 could not be a descent direction. In order to avoid this issue, all the methods have been modified as follows, ensuring the global convergence [6]:

$$\beta^+ = max\{\beta, 0\}. \tag{2.16}$$

This change provides a sort of restart of the algorithm, in case the β found is negative. This is equivalent to forget the last search direction and start again the search from the steepest descent direction. The use of β in Eq. 2.16 is similar to adopt the strategy of restarting the algorithm after N steps, initializing d_k to the current steepest descent direction [16, 7].

$$\beta_k^{PR} = \frac{\mathbf{g}_k^T(\mathbf{g}_k - \mathbf{g}_{k-1})}{\|\mathbf{g}_{k-1}\|^2}, \ \beta_k^{HS} = \frac{\mathbf{g}_k^T(\mathbf{g}_k - \mathbf{g}_{k-1})}{(\mathbf{g}_k - \mathbf{g}_{k-1}^T \mathbf{d}_{k-1})}. \tag{2.17}$$

The HS and the PR methods in Eq. 2.17 have very similar performances and they are two of the most efficient conjugate gradient methods, but they are not globally convergent for nonlinear function[10]. That's why the modification of Eq. 2.16 has been adopted[18], since it enforces the descent property of the algorithm. Both the methods are formulated in such a way that, when occurring small steps, the search direction found is close to the negative gradient direction, getting a final step of decent size[7]. Moreover, the HS method is considered superior to other methods when applied to nonquadratic functions.
If we assume the Assumption 1 and we use a β HS or PR, modified as in Eq.2.16 and a Line Search as the one described in §2.2.3, then it holds the convergence of the algorithm as follows[7]:

$$\lim_{k\to\infty} \|\mathbf{g}_k\| = 0, \tag{2.18}$$

a weaker result with respect to the one involved with strongly convex functions, $\lim_{k\to\infty} \mathbf{g}_k = 0$.
The last method tested is the MHS^+, a modified version of the Hestenes-Stiefel one [13]. It garantees sufficient descent with inexact line search and is based on a modified secant equation which approximates the second order information of the Loss function with high order accuracy. Moreover, it is globally convergent.
It is defined as follows:

$$\beta_k^{MHS} = \frac{\mathbf{g}_k^T \widetilde{\mathbf{y}}_{k-1}^*}{\mathbf{d}_{k-1}^T \widetilde{\mathbf{y}}_{k-1}^*}. \tag{2.19}$$

In order to better understand the formula 2.19, it's important to describe all the components involved in its definiton.
When dealing with quasi-Newton methods, an approximation $\mathbf{B}_{k-1}$ of the Hessian of the Loss function $\nabla^2 L_{k-1}$ is update such that $\mathbf{B}_k$ satisfies the secant condition:

$$\mathbf{B}_k(\mathbf{W}_k - \mathbf{W}_{k-1}) = \mathbf{y}_{k-1}, \tag{2.20}$$

where $\mathbf{y}_{k-1}$ is defined as $\mathbf{g}_k - \mathbf{g}_{k-1}$.
Wei et al. [19] derived a class of modified secant condition:

$$\mathbf{B}_{k-1}(\mathbf{W}_k - \mathbf{W}_{k-1}) = \widetilde{\mathbf{y}}_{k-1}, \tag{2.21}$$

$$\widetilde{\mathbf{y}}_{k-1} = \mathbf{y}_{k-1} + \frac{\theta_{k-1}}{(\mathbf{W}_k - \mathbf{W}_{k-1})^T \mathbf{u}} \mathbf{u}, \tag{2.22}$$

with $\mathbf{u}$ a vector satisying $(\mathbf{W}_k - \mathbf{W}_{k-1})^T \mathbf{u} \neq 0$ and θ_{k-1} defined as:

$$\theta_{k-1} = 2(L_{k-1} - L_k) + (\mathbf{g}_k + \mathbf{g}_{k-1})^T (\mathbf{W}_k - \mathbf{W}_{k-1}). \tag{2.23}$$

Since for $\|(\mathbf{W}_k - \mathbf{W}_{k-1})\| > 1$ the standard secant Eq.2.20 better approximates $\nabla^2 L_{k-1}(\mathbf{W}_k - \mathbf{W}_{k-1})$ than the modified version in Eq.2.22, Livieris and Pintelas[13] proposed a modification of the equation in this way:

$$\mathbf{B}_{k-1}(\mathbf{W}_k - \mathbf{W}_{k-1}) = \widetilde{\mathbf{y}}_{k-1}^*, \tag{2.24}$$

$$\widetilde{\mathbf{y}}_{k-1}^* = \mathbf{y}_{k-1} + \rho_{k-1} \frac{max\{\theta_{k-1}, 0\}}{(\mathbf{W}_k - \mathbf{W}_{k-1})^T \mathbf{u}} \mathbf{u}, \tag{2.25}$$

where $\rho_{k-1} \in \{0, 1\}$ is a parameter that switch between the standard secant equation Eq.2.20 and the modified one Eq.2.24, setting ρ_{k-1} as:

$$\rho_{k-1} = \begin{cases} 1, & \text{if } \|(\mathbf{W}_k - \mathbf{W}_{k-1})\| \leq 1; \\ 0, & \text{otherwise.} \end{cases} \tag{2.26}$$

Since the following assumptions are satisfied by the Loss functions in Eq.1.5:

Assumption 3 - *The level set* $\mathcal{L} = \{w \in \mathbb{R}^n | L(\boldsymbol{W}) \leq L(\boldsymbol{W}_0)\}$ *is bounded, there exist a constant* $B > 0$ *s.t.* $\|\boldsymbol{W}\| \leq B, \forall \boldsymbol{W} \in \mathcal{L}$*;*

Assumption 4 - *If in some neighborhood* $\mathcal{N} \in \mathcal{L}$*, L is differentiable and its gradient* $\boldsymbol{g}$ *is Lipschitz continuous, then there exists a constant* $L > 0$ *such that*

$$\|\boldsymbol{g}(\boldsymbol{W}) - \boldsymbol{g}(\widetilde{\boldsymbol{W}})\|, \forall \ \boldsymbol{W}, \widetilde{\boldsymbol{W}} \in \mathcal{N}, \tag{2.27}$$

if the conjugate gradient algorithm is performed using MSH^+ and computing the search direction $\mathbf{d}^+$ as defined by Eq.2.15, we have:

$$\lim_{k\to\infty} \|\mathbf{g}_k\| = 0, \tag{2.28}$$

that is the convergence of the algorithm.

2.2.3 Line Search

Once computed the new direction $\mathbf{d}$ involved in the new weights $\mathbf{W}+\alpha\mathbf{d}$, a line search has to be implemented in order to find the right step size which minimize the Loss function. The step size α is nothing more than a scalar: the learning rate for the conjugate gradient algorithm, which tells how far is right to move along a given direction.
So, fixed the values of the weights $\mathbf{W}$ and the descent direction $\mathbf{d}$, the main goal is to find the right value for α that is able to minimize the Loss function:

$$\min_{\alpha} \quad L(\mathbf{W} + \alpha\mathbf{d}). \tag{2.29}$$

Of course, we have to deal with a tradeoff: we want a good reduction, but we can't spend too much time computing the exact value for the optimum solution. So, the smarter way to get it is to use an inexact line search, that try some candidate step size and accepts the first one satisfying some conditions.
This search is performed in two phases:

- a *bracketing phase*, that finds an initial interval containing a minimizer;
- an *interpolation phase* that, given the interval, finds the right step length in it.

We decided to use one of the most popular line search condition: the *Armijo-Wolfe* condition.
The search for the better α is led by two condition:

- the *Armijo* one:
$$L(\mathbf{W}_k + \alpha_k\mathbf{d}_k) \leq L(\mathbf{W}_k) + \sigma_1\alpha\nabla L_k^T\mathbf{d}_k \tag{2.30}$$
which ensure that α gives a sufficient decrease of the objective function, being this reduction proportional to the step length α and the directional derivative $\nabla L_k^T d_k$.
The constant σ_1 has been set $\sigma_1 = 10^{-4}$, since it is suggested in literature to be quite small.

- the *Strong Wolfe* condition:
$$|\nabla L(\mathbf{W}_k + \alpha_k\mathbf{d}_k)^T\mathbf{d}_k| \leq L(\mathbf{W}_k) + \sigma_2|\nabla L_k^T\mathbf{d}_k| \tag{2.31}$$
which garantees to choose steps whose size is not too small.
It is also known as curvature condition and ensures that, moving of a step α along the given direction, the slope of our function if greater than σ_2 times the original gradient (if the slope is only slightly negative, the function cannot decrease rapidly along that direction, so it's better to stop the search).
In this case, the constant σ_2 is equal to 0.1, since a smaller value gives a more accurate line search. Futhermore, having choosen the strong condition, which doesn't allow the derivative to be too positive, we are sure that the α found lies close to a stationary point of the function.

Algorithm 5 Line Search satisfying the strong Wolfe conditions. $\alpha_1 > 0$ and α_{max} are given.

procedure LINE_SEARCH
  $\alpha_0 \leftarrow 0$;
  $i \leftarrow 1$;
  while $i \leq max\_iter$ **do**
    Evaluate$L(\alpha_i)$;
    if $[L(\alpha_i) > L(0) + \sigma_1 \alpha_i \nabla L_0^T d_0]$ or $[L(\alpha_i) \leq L(\alpha_{i-1})$ and $i > 1]$ **then**
      $\alpha_* \leftarrow$ **zoom**(α_{i-1}, α_i); **return** α_*;
    Evaluate ∇L_i
    if $|\nabla L_i| \leq -\sigma_2 \nabla L_0^T d_0$ **then**
      $\alpha_* \leftarrow \alpha_i$; **return** α_*;
    if $\nabla L_i \geq 0$ **then**
      $\alpha_* \leftarrow$ **zoom**(α_i, α_{i-1}); **return** α_*;
    if $(|L_i - L_{i-1}| \leq threshold$ **then**
      $\alpha_* \leftarrow \alpha_i$ **return** α_*;
    Choose $\alpha_{i+1} \in (\alpha_i, \alpha_{max})$;
    $i \leftarrow i + 1$;

Algorithm 6 Zoom

procedure ZOOM
  while True **do**
    $\alpha_j \leftarrow quadratic\_interpolation(\alpha_{lo}, \alpha_{hi})$;
    Evaluate $L(\alpha_j)$;
    if $\left[L(\alpha_j) > L(0) + \sigma_1 \alpha_j \nabla L_0^T d_0\right]$ or $[L(\alpha_j) \leq L(\alpha_{lo})]$ **then**
      $\alpha_* \leftarrow \alpha_j$;
      return α_*;
    else
      Evaluate $\nabla L_j^T d_j$;
      if $\left|\nabla L_j^T d_j\right| \leq -\sigma_2 \nabla L_0^T d_0$ **then**
        $\alpha_* \leftarrow \alpha_j$;
        return α_*;
      if $\nabla L_j^T d_j (\alpha_{hi} - \alpha_{lo}) \geq 0$ **then**
        $\alpha_{hi} \leftarrow \alpha_{lo}$;
      if $(|L_j - L_0| \leq threshold$ **then**
        $\alpha_* \leftarrow \alpha_j$
        return α_*;
    $\alpha_{lo} \leftarrow \alpha_j$;

The algorithm satisfing the Strong Wolfe conditions is implemented through the functions described in the pseudocodes 5, 6 and is based on the the algorithms presented in [3], chapter 3.
Since two consecutive values may be similar in finite-precision arithmetic, we set a threshold in both the `line_search` and the quadratic interpolation functions, which garantees that the algorithm stops if two values of α are too close or if the maximum number of iterations has been reached.

Chapter 3

Experiments

In this final chapter we present the results we obtained by applying our model to the datasets we have used to validate and test our ANN, namely, MONKS and CUP. Other than the results, we also present some details about the validation phase for each one of the datasets. In appendix A and B we added some graphs of the ANN's performances during the experimental phases, in order to enrich the presentation.

3.1 MONKS

Before delving into the details of the results we obtained by applying our model to the dataset, we provide some informations about the *preprocessing routines* and *validation schema* we decided to use. Here are the steps we followed in order to reach the final states of our analysis.

1. Since the MONKS datasets feature are categorical, that is, every features value represents a class, not a numerical value, we preprocessed the three datasets by writing a script for applying a *1-of-k encoding*, hence obtaining 17 binary input features.

2. As a supplementary preprocessing phase, we have applied a *symmetrization* to the matrix containing the datasets values, in order to ease the training during the validation phase by having a matrix of values closer to the symmetric behavior of the sigmoid function, which was introduced in section 1.4.

3. Since we have chosen to follow [1] for the hyperparameters' search during the validation phase, we first performed some *preliminary trials* in order to have a glimpse on the best intervals for searching our model's hyperparameters. During this trials we manually varied the model's hyperparameters, e.g. the learning rate, the momentum constant and so on for the SGD and the rho constant for the CGD, and observed the resulting *learning curves*. For this part of the analysis we have used the 20% of the training set as validation set, and the remaining part for training the network.

4. We then deepen the search using the most interesting intervals discovered during the preliminary trials in the validation phase by using our implementation of the (random) *grid search algorithm*, in which we also used our implementation of the *k-fold cross validation algorithm* (which follows the approach of using a value of 5 for the k parameter).

Our validation schema for the MONKS dataset essentially consists in using the random grid search algorithm to investigate some random sampled "points" in the hyperparameters' space, evaluating the performances for each one of this points and finally selecting

the best combinations of parameters based on the diffent metrics like *generalization error*, *accuracy*, *precision*, *recall* and *f1-score*. Both in Tab. 3.2 and Tab. 3.4 we can find the results for the application of our model to the MONKS dataset, using as optimizer, respectively, the Stochastic Gradient Descent and the Conjugate Gradient Methods, represented in a succinct fashion. Each row correspond to a specific dataset. The values for both the MSE and the Accuracy are represented by taking the mean over 10 executions on each dataset using the final configurations for the hyperparameters obtained during the validation phase. In Tab. 3.1 are reported the ranges for the hyperparameters involved in the validation phase. Tab 3.3 gives some additional statistics about the convergence for the SGD.

Table 3.1: Hyperparameters' ranges for the random grid search algorithm with SGD and CG.

Hyperparameters	Ranges
η	$[0.6, 0.8]$
α	$[0.5, 0.9]$
λ	$[0.001, 0.01]$

Hyperparameters	Ranges
σ_2	$[0.1, 0.4]$
ρ	$[0.0, 1.0]$

3.1.1 Results for Stochastic Gradient Descent

Task	Topology	Momentum	η	α	λ	MSE (TR - TS)	Accuracy (TR - TS) (%)
MONK 1	17 -> 4 -> 8 -> 1	standard	0.65	0.75	0.0	0.010 - 0.016	100 % - 99 %
MONK 2	17 -> 4 -> 8 -> 1	standard	0.71	0.89	0.0	0.0072 - 0.010	100 % - 100 %
MONK 3	17 -> 4 -> 8 -> 1	standard	0.67	0.85	0.0027	0.012 - 0.014	98 % - 98 %
MONK 1	17 -> 4 -> 8 -> 1	nesterov	0.63	0.73	0.0	0.016 - 0.024	98 % - 98 %
MONK 2	17 -> 4 -> 8 -> 1	nesterov	0.64	0.85	0.0	0.010 - 0.014	99 % - 99 %
MONK 3	17 -> 4 -> 8 -> 1	nesterov	0.61	0.76	0.0036	0.014 - 0.016	98 % - 98 %

Table 3.2: Results for the Stochastic Gradient Descent.

In order to compare the results in a similar configuration between the SGD and the CG, we have decided to adopt only the *batch* mode for training the ANN. Futhermore, for investigate more about the SGD's performances, we have decided to use both the *Nesterov momentum* and the *Standard momentum*, as described in [9, 23], both in the validation and testing phases. The regularization constant λ is used only in the third dataset, since Monk 3 is the only one among the three that has noisy samples.

Task	Momentum	Convergence on Epoch	Max Accuracy on Epoch
MONK 1	standard	1000	837
MONK 2	standard	1000	607
MONK 3	standard	1000	427
MONK 1	nesterov	1000	641
MONK 2	nesterov	1000	450
MONK 3	nesterov	1000	317

Table 3.3: Additional statistics for the Stochastic Gradient Descent.

As we can see from Tab. 3.3, the Nesterov's momentum is, in general, faster than the standard one in reaching the epoch in which the accuracy score reaches its maximal value. The value of 1000 for the convergence's epochs has to be interpreted as a non convergence for the task, since 1000 is the maximal number of epochs that we have used for the training and the testing phases.

3.1.2 Results for Conjugate Gradient Methods

Task	Topology	Activation	β	σ_1	σ_2	ρ	MSE (TR - TS)	Accuracy (TR - TS) (%)
MONK 1	17 -> 4 -> 8 -> 1	sigmoid	MHS^+	0.0001	0.27	0.67	$9.84e^{-5}$ - 0.0021	100 % - 100 %
MONK 2	17 -> 4 -> 8 -> 1	sigmoid	MHS^+	0.0001	0.12	0.73	$9.62e^{-5}$ - 0.0026	100 % - 99 %
MONK 3	17 -> 4 -> 8 -> 1	sigmoid	MHS^+	0.0001	0.24	0.27	0.0038 - 0.010	99 % - 98 %
MONK 1	17 -> 4 -> 8 -> 1	sigmoid	HS^+	0.0001	0.28	0.0	$9.94e^{-5}$ - 0.00059	100 % - 100 %
MONK 2	17 -> 4 -> 8 -> 1	sigmoid	HS^+	0.0001	0.35	0.0	$9.85e^{-5}$ - 0.0019	100 % - 100 %
MONK 3	17 -> 4 -> 8 -> 1	sigmoid	HS^+	0.0001	0.14	0.0	0.0033 - 0.011	99 % - 98 %
MONK 1	17 -> 4 -> 8 -> 1	sigmoid	PR^+	0.0001	0.11	0.0	0.014 - 0.030	97 % - 94 %
MONK 2	17 -> 4 -> 8 -> 1	sigmoid	PR^+	0.0001	0.21	0.0	0.050 - 0.059	97 % - 88 %
MONK 3	17 -> 4 -> 8 -> 1	sigmoid	PR^+	0.0001	0.12	0.0	0.040 - 0.048	97 % - 90 %

Table 3.4: Results for the Conjugate Gradient Methods.

For what concernes the results for the Conjugate Gradient Methods, for all the configurations tested, the direction has been set to the modified one, since it's the one who gave better results in the preliminary tests. It's immediate to see the higher performances achieved by the modified MHS^+ method and by the HS^+ one, reaching an accuracy of 100% in both the training and test sets. On the contrary, the PR^+ method performes worse than the Stochastic Gradient Descent, behaving as in presence of overfitting. Like in the results obtained from SGD, the third dataset tested (Monk 3) gets a little worse performances because of the presence of noise in it. Anyway, it's interesting to underline that these models are able to reach higher accuracies with the ability to "auto-tune" themeselves, that is without the necessity to modify at hand hyperparameters as the learning rate or the the momentum term. Another nice behaviour with respect to the one showed in SGD, is evident from the learning curves in Appendix A: the Conjugate Gradient Methodss reach the highest accuracy in the very first epochs, while the Stochastic Gradient Descent needs much more epochs of training to converge towards an optimum value.

3.2 CUP

A main difference with respect to the models used for Monk, is the choice of the topology of the Network. First of all, in this case, being the final task a regression one in which there are two different values to be predicted, the output layer of the ANN is composed by two neurons of output, each one associated to an identity function.
Then, after some preliminary trials, we have decided to set the number of the hidden layers to [16, 32].
Of course, as for the Monk dataset, we have performed a validation phase on the CUP dataset, in order to discover the best parameters for the network. As described in Sec. 3.1, it has been implemented a *3-fold cross validation algorithm* with a (random) *grid search*. Table 3.5 shows the ranges in which the searching of the hyperparameters has been carried out. Once again, the momentum type chosen is Nesterov, while the direction used in the Conjugate Gradient Methods is the modified one.

Table 3.5: Hyperparameters' ranges for the random grid search algorithm with SGD and CG.

Hyperparameters	Ranges
η	$[0.004, 0.2]$
α	$[0.5, 0.9]$
λ	$[0.0003, 0.003]$

Hyperparameters	Ranges
σ_2	$[0.1, 0.4]$
ρ	$[0.0, 1.0]$

In tables 3.6 and 3.7 are annotated the average results obtained from 10 executions of

each one of the best models identified thanks to the validation step.

Model	Topology	Batch size	Activation	η	α	λ	MSE (TR - TS)
SGD	10 -> 16 -> 32 -> 2	batch	identity	0.084	0.79	0.0009	1.00 - 1.35

Table 3.6: Results for the Stochastic Gradient Descent.

β	Topology	Batch size	Activation	σ_1	σ_2	ρ	MSE (TR - TS)
MHS^+	10 -> -> 16 -> 32 -> 2	batch	identity	0.0001	0.27	0.29	0.97 - 1.50
HS^+	10 ->-> 16 -> 32 -> 2	batch	identity	0.0001	0.39	0.0	0.97 - 1.50
PR^+	10 ->-> 16 -> 32 -> 2	batch	identity	0.0001	0.86	0.00	1.15 - 1.41

Table 3.7: Results for the Conjugate Gradient Methods.

As for the experiments with the Monk datasets, we attach the learning curves in Appendix B.

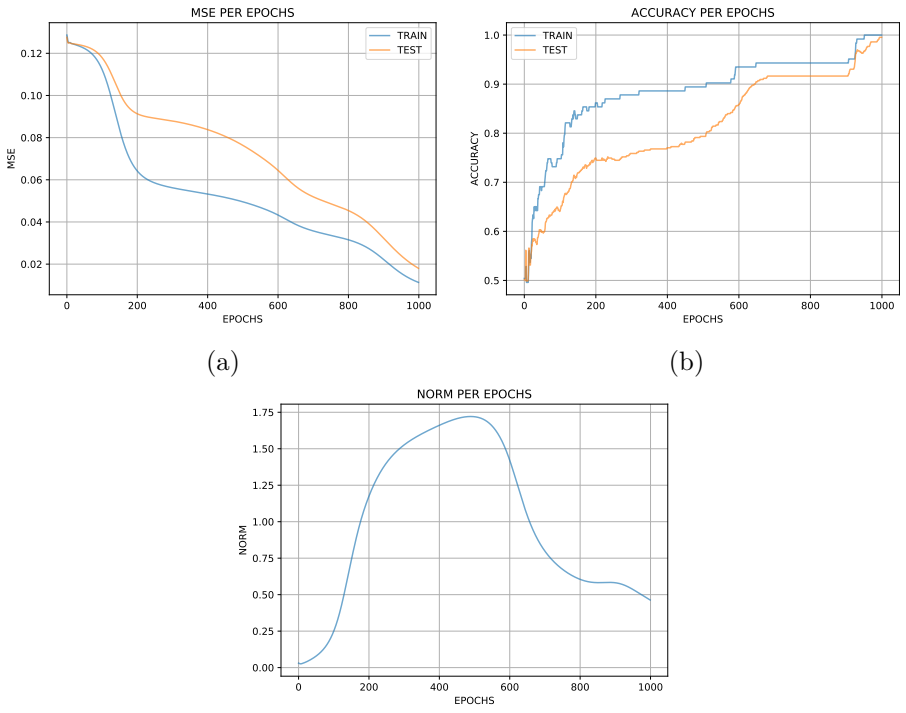
Appendix A

MONKS' learning curves

Here we present the *learning curves* for the three MONKS datasets. We have sampled these curves during the final tests on each dataset by plotting one of the 10 trails we used for building the statistics we presented in section 3.1, hence each plot presents a possible learning curve for the best hyperparameters' selection for each dataset. Alongside of the learning curves we also provide the *accuracy plots* in order to show how the accuracy score progresses during the epochs, and the *gradient's norm plots* in order to show the convergence rate.

A.1 Stochastic Gradient Descent

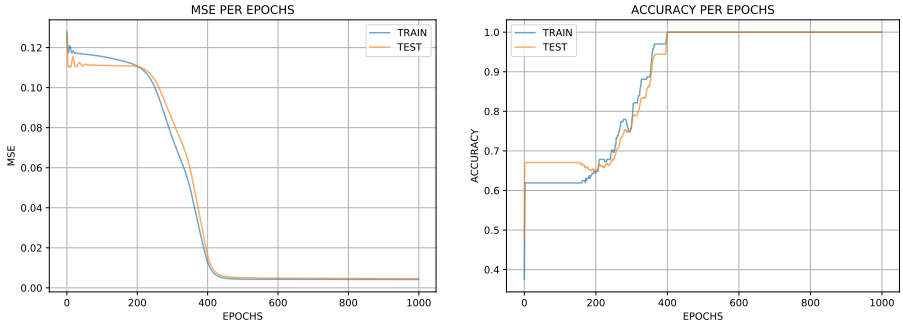
A.1.1 Standard momentum



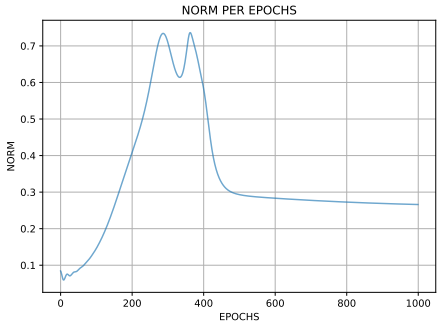
(a) (b)

(c)

Figure A.1: Example of a final learning, accuracy score curve and convergence rate curve on MONKS 1 with standard momentum.

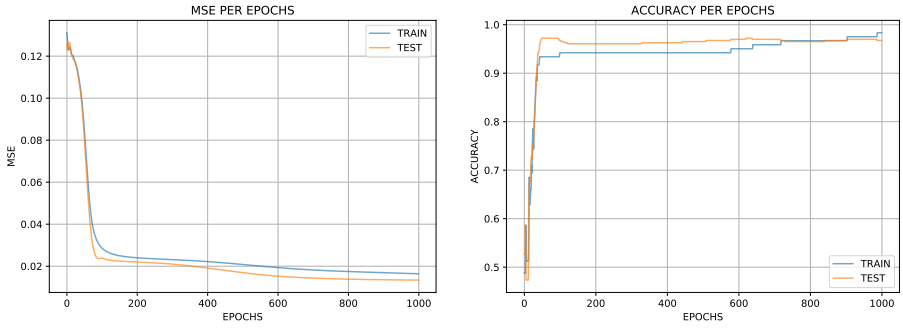


(a) (b)

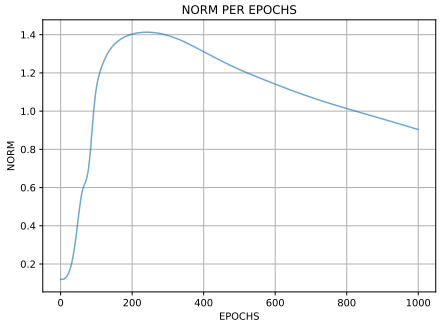


(c)

Figure A.2: Example of a final learning, accuracy score curve and convergence rate curve on MONKS 2 with standard momentum.



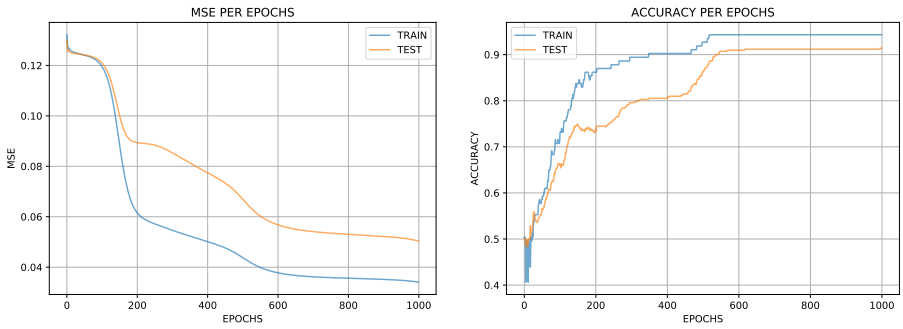
(a) (b)



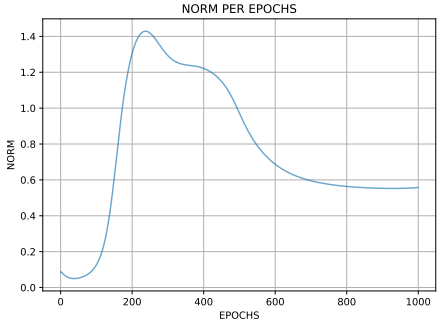
(c)

Figure A.3: Example of a final learning, accuracy score curve and convergence rate curve on MONKS 3 with standard momentum.

A.1.2 Nesterov's momentum

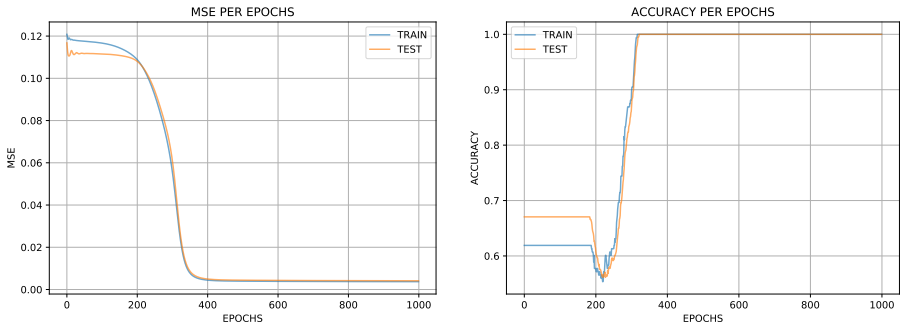


(a) (b)

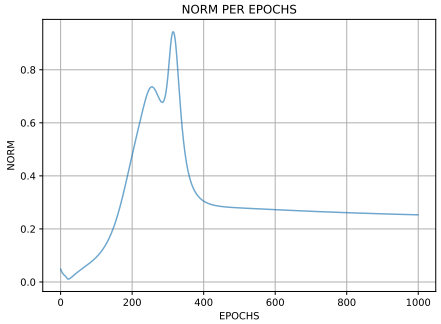


(c)

Figure A.4: Example of a final learning, accuracy score curve and convergence rate curve on MONKS 1 with Nesterov's momentum.

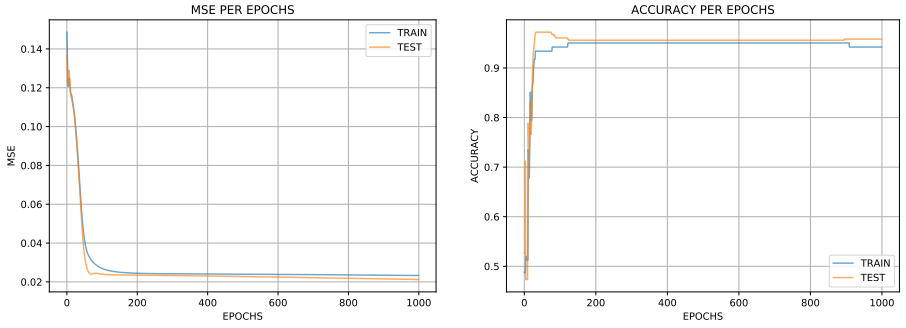


(a) (b)

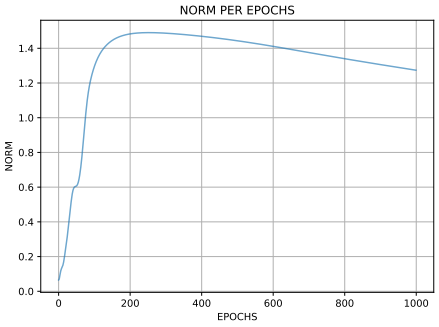


(c)

Figure A.5: Example of a final learning, accuracy score curve and convergence rate curve on MONKS 2 with Nesterov's momentum.



(a) (b)



(c)

Figure A.6: Example of a final learning, accuracy score curve and convergence rate curve on MONKS 3 with Nesterov's momentum.

Appendix B

CUP's learning curves

As in the previous section, the following curves are the result of the final tests on the test dataset. It has been plotted one of the 10 trails we used for building the statistics we presented in Section 3.2, hence each plot presents a possible learning curve for the best hyperparameters selection for each dataset.

B.1 Stochastic Gradient Descent

Bibliography

[1] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, 2 2012.

[2] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning, 2016. quantization overview.

[3] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[4] L. C. W. DIXON. Conjugate gradient algorithms: Quadratic termination without linear searches. *IMA Journal of Applied Mathematics*, 15, 1975.

[5] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[6] J. Gilbert and J. Nocedal. Global convergence properties of conjugate gradient methods for optimization. *SIAM Journal on Optimization*, 2(1):21–42, 1992.

[7] Jorge Gilbert, Jean Charles; Nocedal. Global convergence properties of conjugate gradient methods for optimization. *SIAM Journal on Optimization*, 2, 02 1992.

[8] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*, 2010.

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

[10] David F. Griffiths. *[Lecture Notes in Mathematics] Numerical Analysis Volume 1066 // Nonconvex minimization calculations and the conjugate gradient method*, volume 10.1007/BFb0099514. 1984.

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

[12] Li Zhang; Weijun Zhou; Donghui Li. Global convergence of a modified fletcher-reeves conjugate gradient method with armijo-type line search. *Numerische Mathematik*, 104, 10 2006.

[13] Ioannis E. Livieris and Panagiotis Pintelas. A new conjugate gradient algorithm for training neural networks based on a modified secant equation. *Applied Mathematics and Computation*, 221:491 – 502, 2013.

[14] T.M. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997.

[15] Grgoire Montavon, Genevive Orr, and Klaus-Robert Mller. *Neural Networks: Tricks of the Trade*. Springer Publishing Company, Incorporated, 2nd edition, 2012.

[16] Martin Fodslette Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6(4):525 – 533, 1993.

[17] Y. E. NESTEROV. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. *Dokl. Akad. Nauk SSSR*, 269:543–547, 1983.

[18] M J. D. Powell. Convergence properties of algorithms for nonlinear optimization. *SIAM Review*, 28, 12 1986.

[19] Zengxin Wei; Guoyin Li; Liqun Qi. New quasi-newton methods for unconstrained optimization problems. *Applied Mathematics and Computation*, 175, 2006.

[20] D. E. RUMERLHAR. Learning representation by back-propagating errors. *Nature*, 323:533–536, 1986.

[21] David Saad, editor. *On-line Learning in Neural Networks*. Cambridge University Press, New York, NY, USA, 1998.

[22] Shai Shalev-Shwartz and Ambuj Tewari. Stochastic methods for l1-regularized loss minimization. *Journal of Machine Learning Research*, 12:1865–1892, 2011.

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