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 19, 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 [6].

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 [3], 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 [6, 7, 10], 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 [6, 7, 10], 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,

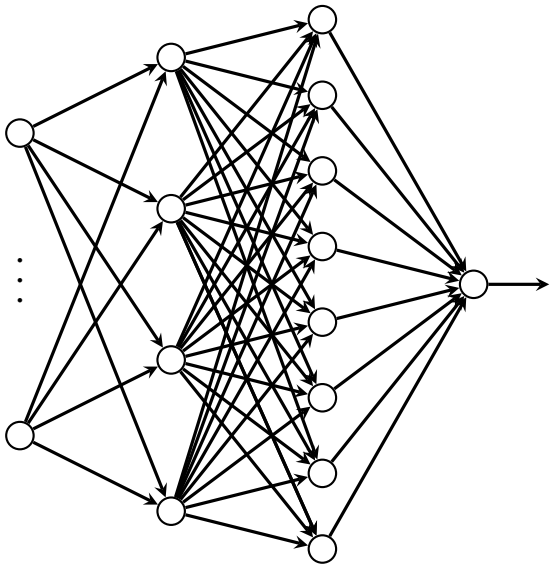


Figure 1.1: The ANN's structure for the classification task, the majority of the input nodes are omitted because they vary from dataset to dataset.

we have decided to follow the *normalized initialization*, as described in [5, 6], 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 section, 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*;

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 [14] and described in [6, 7, 10]. 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 formula 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 section 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 section 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 functions, 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 $(0,\ 1)$.

1.5 The Loss function

It section 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 two well known functions, varying on the type of task the ANN is fulfilling, the *Mean Squared Error*, MSE for short, when the task is a classification task, and the *Mean Euclidean Error*, MEE for short, when the task is a regression task. The MSE is obtained with the formula

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

while the MEE is obtained with the formula

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

where m represents the number of samples for which we computed the output $\hat{\mathbf{y}}$. Since this formulas are used for computing the performances of our models, our primary goal is to minimize their output as much as possible by computing the gradient during the backward propagation step of the back-propagation algorithm, as described in section 1.3. In order to do so, they have to be *differentiable* functions. The gradient for the MSE with respect to $\hat{\mathbf{y}}$ is

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

while the gradient for the MEE is

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

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 [6], supporting both *momentum* and *regularization*. [9]
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.

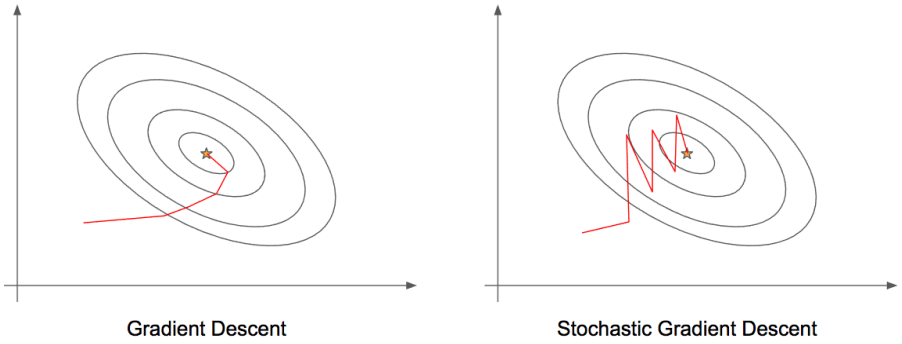


Figure 2.1: The different trajectories from GD and SGD.

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:

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

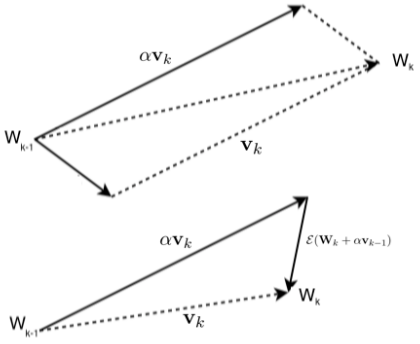
Futhermore, if we assume that the Hessian matrix of the Loss function is strictly positive definite at the optimum, which means that the Loss function is strogly convex, the convergence rate is $O(\frac{1}{k})$, with k as the number of epochs in the training. Otherwise, relaxing this assumption in presence of a convex problem, the convergence rate becomes $O(\frac{1}{\sqrt{k}})$[6, 11, 15].
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.

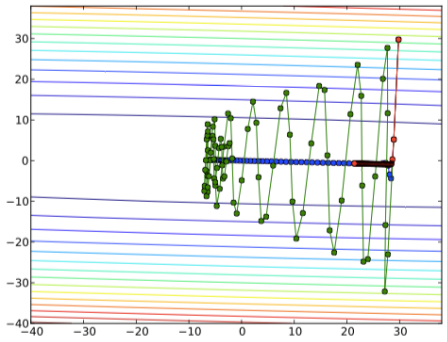
procedure STOCHASTIC GRADIENT DESCENT
  Initialize $\mathbf{W}$ and $\mathbf{v}$
  $k \leftarrow 0$
  while $k < max\_epochs$ **do**
    Sample a minibatch of m training examples $\{(x_0, y_0), (x_1, y_1), ..., (x_m, y_m)\}$
    if Nesterov Momentum **then**
      $\mathbf{W} \leftarrow \mathbf{W} + \alpha\mathbf{v}$
    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



(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).

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:

$$\mathbf{v}_k = \alpha \mathbf{v}_{k-1} + \eta \nabla L(\mathbf{W}_k). \tag{2.4}$$

The new synaptic weights are then updated as:

$$\mathbf{W}_k = \mathbf{W}_{k-1} + \mathbf{v}_k. \tag{2.5}$$

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:

$$\mathbf{v}_k = \alpha \mathbf{v}_{k-1} + \eta \nabla L(\mathbf{W}_k + \alpha \mathbf{v}_{k-1}). \tag{2.6}$$

The update of the weights $\mathbf{W}$ follows the one described in Eq. 2.5.
The variant respect Eq. 2.4, 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 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 $O(\frac{1}{k^2})$[13].

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:

$$\min_{\mathbf{W} \in \mathbb{R}^n} \quad L(\mathbf{W}) + \lambda\Omega(\mathbf{W}), \tag{2.7}$$

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 implemented:

- *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 $\Omega(\mathbf{W}) = \sum_{i=1}^{k} w_i^2 = \|\mathbf{W}\|_2^2$.

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:

$$\Delta\mathbf{W} = \alpha\mathbf{d}, \tag{2.8}$$

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:

$$\mathbf{W}_{k+1} = \mathbf{W}_k + \alpha_k\mathbf{d}_k, \quad k = 0, 1, ..., \tag{2.9}$$

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.2.8.

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

```
procedure NONLINEAR CONJUGATE GRADIENT
    Initialize W_0 and L_g
    k ← 0, g_0 ← 0
    while k < max_epochs do
        Evaluate the Loss function L_k and its gradient g_k
        if L_k < L_g  or  ||g_k|| > tolerance then
            return Error goal reached
        Compute the β with one of the methods HS, MHS, FR, PR
        Compute the direction: d_k ← −g_k + β d_k
        Compute the learning rate α_k with a Line Search
        Update the weights: W_{k+1} ← W_k + α_k d_k
        k ← k + 1
```

2.2.1 Search Direction

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

$$\mathbf{d}_k^T \mathbf{H} \mathbf{d}_{tk-1} = 0, \tag{2.10}$$

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:

$$\mathbf{d}_k = \begin{cases} -\mathbf{g}_0, & \text{if } k = 0; \\ -\mathbf{g}_k + \beta_k \mathbf{d}_{k-1}, & \text{otherwise,} \end{cases} \tag{2.11}$$

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.[8], 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:

$$\mathbf{d}_k^+ = \begin{cases} -\mathbf{g}_0, & \text{if } k = 0; \\ -(1 + \beta_k \frac{\mathbf{g}_k^T \mathbf{d}_k}{\|\mathbf{g}_k\|}) \mathbf{g}_k + \beta_k \mathbf{d}_{k-1}^+, & \text{otherwise.} \end{cases} \tag{2.12}$$

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 four: the Fletcher-Reeves (FR), 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. Since, in our network, we are dealing with a nonquadratic Loss function, the direction computed as in Eq. 2.11 could not be a descent direction. In order to avoid this issue, all the methods have been modified as follows, ensuring the global convergence [4]:

$$\beta^+ = max\{\beta, 0\}. \tag{2.13}$$

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. Furthermore, because of the nonquadratic nature of the error function, the algorithm will not necessarily converge in N steps, as it usually does when applied to quadratic functions. The use of β in Eq. 2.13 is similar to adopt the strategy of restarting the algorithm after N steps, initializing d_k to the current steepest descent direction [12].

$$\beta_k^{PR} = \frac{\mathbf{g}_k^T(\mathbf{g}_k - \mathbf{g}_{k-1})}{\|\mathbf{g}_{k-1}\|^2},\ \beta_k^{FR} = \frac{\|\mathbf{g}_k\|^2}{\|\mathbf{g}_{k-1}\|^2},\ \beta_k^{HS} = \frac{\mathbf{g}_k^T(\mathbf{g}_k - \mathbf{g}_{k-1})}{(\mathbf{g}_k - \mathbf{g}_{k-1}^T\mathbf{d}_{k-1})}. \tag{2.14}$$

The HS and the PR methods in Eq. 2.14 have very similar performances and they are two of the most efficient conjugate gradient methods, but they are not globally convergent for nonlineat function. That's why the modification of Eq. 2.13 has been adopted. Moreover, the HS method is considered superior to other methods when applied to nonquadratic functions.
For what concernes the FR method (also described in Eq.2.14), it requires a constrain on the parameters of the inexact line search procedure of section 2.2.3, used to identify the right step length α. In particular, it requires that $\sigma_1 < \sigma_2 < 0.5$ in order to garantee that the Armijo Wolfe conditions are satisfied, and it seems to be less efficient and robust than the other methods [2], chapter 5. Anyway, by imposing this condition, the FR method is globally convergent even when dealing with nonlinear functions.
The last method tested is the MHS^+, a modified version of the Hestenes-Stiefel one [9]. 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:

$$\beta_k^{MHS} = \frac{\mathbf{g}_k^T\widetilde{y}_{k-1}^*}{\mathbf{d}_{k-1}^T\widetilde{y}_{k-1}^*}. \tag{2.15}$$

In order to better understand the formula 2.15, 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:

$$\mathbf{B}_k(\mathbf{W}_k - \mathbf{W}_{k-1}) = \mathbf{y}_{k-1}, \tag{2.16}$$

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

$$\mathbf{B}_{k-1}(\mathbf{W}_k - \mathbf{W}_{k-1}) = \widetilde{y}_{k-1}, \tag{2.17}$$

$$\widetilde{y}_{k-1} = y_{k-1} + \frac{\theta_{k-1}}{(\mathbf{W}_k - \mathbf{W}_{k-1})^T\mathbf{u}}\mathbf{u}, \tag{2.18}$$

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

$$\theta_{k-1} = 2(L_{k-1} - L_k) + (\mathbf{g}_k + \mathbf{g}_{k-1})^T(\mathbf{W}_k - \mathbf{W}_{k-1}). \tag{2.19}$$

Since for $\|(\mathbf{W}_k - \mathbf{W}_{k-1})\| > 1$ the standard secant Eq.2.16 better approximates $\nabla^2 L_{k-1}(\mathbf{W}_k - \mathbf{W}_{k-1})$ than the modified version in Eq.2.18, Livieris and Pintelas[9] proposed a modification of the equation in this way:

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

$$\widetilde{y}^*_{k-1} = y_{k-1} + \rho_{k-1}\frac{max\{\theta_{k-1}, 0\}}{(\mathbf{W}_k - \mathbf{W}_{k-1})^T\mathbf{u}}\mathbf{u}, \tag{2.21}$$

where $\rho_{k-1} \in \{0, 1\}$ is a parameter that switch between the standard secant equation Eq.2.16 and the modified one Eq.2.20, setting ρ_{k-1} as:

$$\rho_{k-1} = \begin{cases} 1, & \text{if } \|(\mathbf{W}_k - \mathbf{W}_{k-1})\| \leq 1; \\ 0, & \text{otherwise.} \end{cases} \tag{2.22}$$

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

Assumption 1 - *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 2 - *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*

$$\|\boldsymbol{g}(\boldsymbol{W}) - \boldsymbol{g}(\widetilde{\boldsymbol{W}})\|, \forall\ \ \boldsymbol{W}, \widetilde{\boldsymbol{W}} \in \mathcal{N}, \tag{2.23}$$

if the conjugate gradient algorithm is performed using MSH^+ and computing the search direction $\mathbf{d}^+$ as defined by Eq.2.12, we have:

$$\lim_{k\to\infty} \|\mathbf{g}_k\| = 0, \tag{2.24}$$

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:

$$\min_{\alpha} \quad L(\mathbf{W} + \alpha\mathbf{d}). \tag{2.25}$$

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:

  $$L(\mathbf{W}_k + \alpha_k\mathbf{d}_k) \leq L(\mathbf{W}_k) + \sigma_1\alpha\nabla L_k^T\mathbf{d}_k \tag{2.26}$$

  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:

$$|\nabla L(\mathbf{W}_k + \alpha_k \mathbf{d}_k)^T \mathbf{d}_k| \leq L(\mathbf{W}_k) + \sigma_2 |\nabla L_k^T \mathbf{d}_k| \tag{2.27}$$

  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 $\alpha_{i+1} \in (\alpha_i, \alpha_{max})$;
    $i \leftarrow i + 1$;

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 [2], 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.

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_*;
    $\alpha_{lo} \leftarrow \alpha_j$;

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. We enrich the presentation by adding some graphs of the ANN's performances during the experimental phase.

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 standard approach of using a value of 3 for the k parameter).

That is, our validation schema for the MONKS dataset essentially consist 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*.

Bibliography

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

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

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

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

[5] 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.

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

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

[8] 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.

[9] 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.

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

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

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

[13] 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.

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

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