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 16, 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 [3].

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 [1], 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 [3, 4, 6], 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.

1.2 Initializing the Network

As we know from [3, 4, 6], an ANN is composed by a set of weights $\mathbf{W}^{(i)}$, and a set of biases $\mathbf{b}^{(i)}$, $i \in \{1, \ldots, l\}$, 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, we have decided to follow the *normalized initialization*, as described in [2], 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.

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 respect to every network's unit;
2. optimize the information gathered during the first phase using a distinct technique, like the ones described in chapter 2;

For computing the gradient we have chosen to use the well known *backpropagation algorithm*, firstly introduced in [7] and described in [3, 4, 6]. 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 informtion 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. 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.

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 1.4. The function f applied on line 5 represents the layer's *activation function*, while $\lambda\Omega(\theta)$ represents the network's regularization term.

```
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:     $\mathbf{h}^{(0)} = \mathbf{x}$
3:     for $k = 1,\dots,l$ do
4:         $\mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{k-1}$
5:         $\mathbf{h}^{(k)} = f(\mathbf{a}^{(k)})$
6:     $\hat{\mathbf{y}} = \mathbf{h}^{(l)}$
7:     $J = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda\Omega(\theta)$
```

Algorithm 2 Backward computation for the (deep) neural network of algorithm 1. Here, the $\odot$ symbol represents the element-wise (Hadamard) product.

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

1.4 Loss function is differentiable?

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 **??** is the standard SGD version implemented, as described in [3], supporting both *momentum* and *regularization.* [5]
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 identify and compute a local minimum, moving along the direction of the steepest descent of the function, that is the negative gradient $-\nabla\mathcal{E}(\mathbf{W})$.
As we already said, the objective of the training procedure is to minimize the loss function. Of course, in order to arrive to a local minimum, it is necessary to update the synaptic weights $\mathbf{W}$ at every iteration as $\mathbf{W} = \mathbf{W} + \Delta\mathbf{W}$.
The correction applied to the weights is defined as follows, taking a step in the opposite direction of the cost gradient:

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}, \tag{2.1}$$

where η is the learning rate. The latter one is a fundamental hyperparameters which has to been choosen 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 be quite costly because of the potentially big number of training examples, getting a longer time to converges to the minimum when dealing with a very large dataset, 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 examples to be considered at each epoch of training (batch or mini batch).
The name stochastic derives from the fact that the samples are selected randomly, 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.
For what concernes the 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:

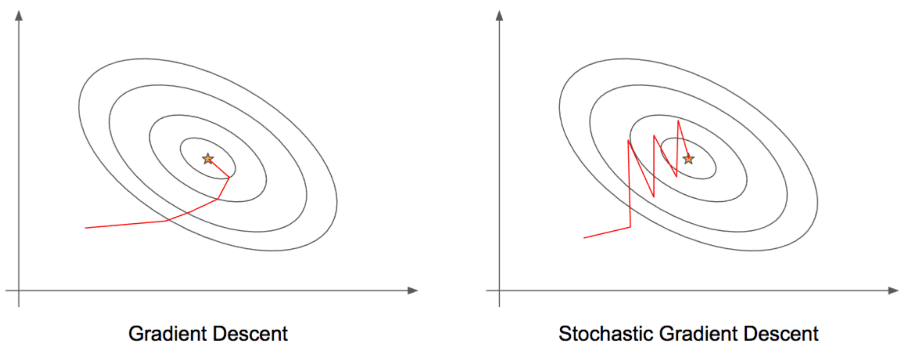

Figure 2.1: The different trajectories from GD and SGD.

$$\sum_{t=1}^{\infty} \eta_t = \infty \quad \text{and} \quad \sum_{t=1}^{\infty} \eta_t^2 < \infty \tag{2.2}$$

Futhermore, if we assume that the Hessian matrix of the loss function is strictly positive definite at the optimum, that is having 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 $O(\frac{1}{\sqrt{k}})$[3].
A property of SGD is that computation time per update does not grow with the number of training examples, allowing convergence even in presence of a large dataset.

2.1.1 Momentum

When computing the adjustment of the synaptic weights $\Delta\mathbf{W}$ as in Eq. 2.2, 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 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:

$$\mathbf{v}_k = \alpha\mathbf{v}_{k-1} + \eta\nabla\mathcal{E}(\mathbf{W}_k). \tag{2.3}$$

The new synaptic weights are then updated as:

$$\mathbf{W}_k = \mathbf{W}_{k-1} + \mathbf{v}_k. \tag{2.4}$$

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.2.3
The velocity vector $\mathbf{v}$ is computed as:

$$\mathbf{v}_k = \alpha\mathbf{v}_{k-1} + \eta\nabla\mathcal{E}(\mathbf{W}_k + \alpha\mathbf{v}_{k-1}). \tag{2.5}$$

The update of the weights $\mathbf{W}$ follows the one described in Eq. 2.4.

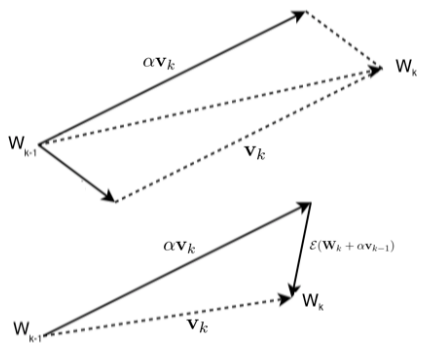

Figure 2.2: The classical momentum on top and the Nesterov Accelerated gradient on bottom.

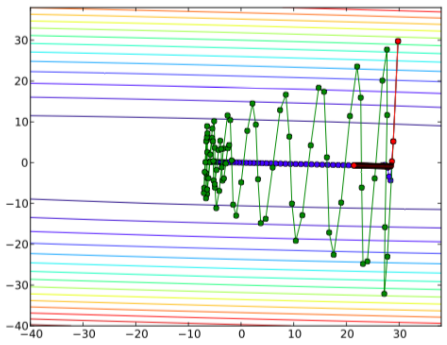

Figure 2.3: The trajectory from CM (in green) and the one from NAG (in blue).

The variant respect Eq. 2.3, shown in fig.2.2, is given by the fact that the gradient $\nabla\mathcal{E}$ 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 $O(\frac{1}{k^2})$.

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:

$$\min_{\mathbf{W} \in \mathbb{R}^n} \quad \mathcal{E}(\mathbf{W}) + \lambda \mathcal{E}_c(\mathbf{W}), \tag{2.6}$$

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

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.
We can, indeed, approximate the *loss function* in a quadratic form, around a given point $\mathbf{W}$, using the Taylor approximation:

$$\mathcal{E}(\mathbf{W} + \Delta\mathbf{W}) = \mathcal{E}(\mathbf{W}) + \mathbf{g}\mathcal{E}^T\Delta\mathbf{W} + \frac{1}{2}\Delta\mathbf{W}^T\mathbf{H}\Delta\mathbf{W}, \tag{2.7}$$

where $\mathbf{H}$ is the *Hessian* matrix and $\mathbf{g}$ the gradient vector, getting the benefit of choosing the search direction and the step size more carefully by using information from the second order approximation.
Given Eq. 2.7, the optimum adjustment to apply to the weights of the network should be

$$\Delta\mathbf{W}^* = \mathbf{H}^{-1}\mathbf{g}, \tag{2.8}$$

carrying the weight of the computation of the inverse of the Hessian.
In order to avoid this expensive computation, 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:

$$\Delta\mathbf{W} = \alpha\mathbf{d}, \tag{2.9}$$

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 following minimization problem:

$$\min_{\mathbf{W} \in \mathbb{R}^n} \quad \mathcal{E}(\mathbf{W}), \tag{2.10}$$

where $\mathcal{E}$ is the loss function and $\mathbf{W}$ are the synaptic weights of the network.
As showed in the pseudocode **??**, the iterative formula generates a sequence of weights $\{W_k\}$, for every epoch of training k, as:

$$\mathbf{W}_{k+1} = \mathbf{W}_k + \alpha\mathbf{d}_k, \quad k = 0, 1, ..., \tag{2.11}$$

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.2.9.

2.2.1 Search Direction

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

$$\mathbf{d}_k^T\mathbf{H}\mathbf{d}_{tk-1} = 0, \tag{2.12}$$

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:

$$\mathbf{d}_k = \begin{cases} -\mathbf{g}_0, & \text{if } k = 0; \\ -\mathbf{g}_k + \beta_k \mathbf{d}_{k-1}, & \text{otherwise}, \end{cases} \tag{2.13}$$

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.**??**, 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:

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

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. 2.13 could not be a descent direction. In order to avoid this issue, all the methods have been modified as follows, ensuring the global convergence:

$$\beta^+ = max\{\beta, 0\}. \tag{2.15}$$

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. 2.15 is similar to adopt the strategy of restarting the algorithm after N steps, initializing d_k to the current steepest descent direction.

$$\beta_k^{PR} = \frac{\mathbf{g}_k^T(\mathbf{g}_k - \mathbf{g}_{k-1})}{\|\mathbf{g}_{k-1}\|^2}, \ \beta_k^{FR} = \frac{\|\mathbf{g}_k\|^2}{\|\mathbf{g}_{k-1}\|^2}, \ \beta_k^{HS} = \frac{\mathbf{g}_k^T(\mathbf{g}_k - \mathbf{g}_{k-1})}{(\mathbf{g}_k - \mathbf{g}_{k-1}^T \mathbf{d}_{k-1})}. \tag{2.16}$$

The HS and the PR methods in eq. 2.16 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. 2.15 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.2.16), 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.Anyway, by imposing this condition, the FR method is globally convergent even when dealing with nonlinear functions.
The last method tested is the MHS^+ (eq. 2.17, a modified version of the Hestenes-Stiefel one. 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:

$$\beta_k^{MHS} = \frac{\mathbf{g}_k^T \widetilde{y}_{k-1}^*}{\mathbf{d}_{k-1}^T \widetilde{y}_{k-1}^*}. \tag{2.17}$$

In order to better understand the formula 2.17, 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 $\nabla^2 \mathcal{E}_{k-1}$ is update such that $\mathbf{B}_k$ satisfies the secant condition:

$$\mathbf{B}_k(\mathbf{W}_k - \mathbf{W}_{k-1}) = \mathbf{y}_{k-1}, \tag{2.18}$$

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:

$$\mathbf{B}_{k-1}(\mathbf{W}_k - \mathbf{W}_{k-1}) = \widetilde{y}_{k-1}, \tag{2.19}$$

$$\widetilde{y}_{k-1} = y_{k-1} + \frac{\theta_{k-1}}{(\mathbf{W}_k - \mathbf{W}_{k-1})^T \mathbf{u}} \mathbf{u}, \tag{2.20}$$

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

$$\theta_{k-1} = 2(\mathcal{E}_{k-1} - \mathcal{E}_k) + (\mathbf{g}_k + \mathbf{g}_{k-1})^T (\mathbf{W}_k - \mathbf{W}_{k-1}). \tag{2.21}$$

Since for $\|(\mathbf{W}_k - \mathbf{W}_{k-1})\| > 1$ the standard secant Eq.2.18 better approximates $\nabla^2 \mathcal{E}_{k-1}(\mathbf{W}_k - \mathbf{W}_{k-1})$ than the modified version in Eq.2.20, Livieris et al. proposed a modification of the equation in this way:

$$\mathbf{B}_{k-1}(\mathbf{W}_k - \mathbf{W}_{k-1}) = \widetilde{y}_{k-1}^*, \tag{2.22}$$

$$\widetilde{y}_{k-1}^* = y_{k-1} + \rho_{k-1} \frac{max\{\theta_{k-1}, 0\}}{(\mathbf{W}_k - \mathbf{W}_{k-1})^T \mathbf{u}} \mathbf{u}, \tag{2.23}$$

where $\rho_{k-1} \in \{0, 1\}$ is a parameter that switch between the standard secant Eq.2.18 and the modified one 2.22, setting ρ_{k-1} as:

$$\rho_{k-1} = \begin{cases} 1, & \|(\mathbf{W}_k - \mathbf{W}_{k-1})\| \leq 1; \\ 0, & \text{otherwise.} \end{cases} \tag{2.24}$$

It's suggested to use MHS^+ with the search direction $\mathbf{d}^+$ defined by Eq.2.14.

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:

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

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:
$$\mathcal{E}(W_k+\alpha_k d_k) \leq \mathcal{E}(W_k) + \sigma_1 \alpha \nabla\mathcal{E}_k^T d_k \tag{2.26}$$
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 $\nabla\mathcal{E}_k^T d_k$.

  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:
$$|\nabla\mathcal{E}(W_k+\alpha_k d_k)^T d_k| \leq \mathcal{E}(W_k) + \sigma_2 |\nabla\mathcal{E}_k^T d_k| \tag{2.27}$$
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.

The algorithm satisfing the Strong Wolfe conditions is implemented through three functions, as described in the pseudocodes 3, **??**, 5: `line_search`, `zoom` and `interpolate_alpha`. Since two consecutive values may be similar in finite-precision arithmetic, we set a threshold in both the `line_search` and `interpolate_alpha` 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.

The `line_search` function try to find and return a good α; if it fails, it returns an interval in which continue the searching, invoking the `zoom` function, which decreases the size of the interval, until it finds and returns a good step length.
`Zoom` invokes another function, `interpolate_alpha`, which is nothing more than the implementation of a bisection interpolation in order to find a trial α inside the given interval.

Algorithm 3 Line Search

```
procedure LINE_SEARCH

    $\alpha_0 \leftarrow 0$;
    $i \leftarrow 1$;
    while $i \leq max\_iter$ do
        Evaluate$\mathcal{E}(\alpha_i)$;
        if [$\mathcal{E}(\alpha_i) > \mathcal{E}(0) + \sigma_1 \alpha_i \nabla\mathcal{E}_0^T d_0$] or [$\mathcal{E}(\alpha_i) \leq \mathcal{E}(\alpha_{i-1})$ and $i > 1$] then
            $\alpha_* \leftarrow$ zoom($\alpha_{i-1}, \alpha_i$); return $\alpha_*$;
        Evaluate $\nabla\mathcal{E}_i$
        if $|\nabla\mathcal{E}_i| \leq -\sigma_2 \nabla\mathcal{E}_0^T d_0$ then
            $\alpha_* \leftarrow \alpha_i$; return $\alpha_*$;
        if $\nabla\mathcal{E}_i \geq 0$ then
            $\alpha_* \leftarrow$ zoom($\alpha_i, \alpha_{i-1}$); return $\alpha_*$;
        if ($|\mathcal{E}_i - \mathcal{E}_{i-1}| \leq threshold$ then
            $\alpha_* \leftarrow \alpha_i$ return $\alpha_*$;
        Choose $\alpha_{i+1} \in (\alpha_i, \alpha_{max})$;
        $i \leftarrow i + 1$;
```

Algorithm 4 Zoom

```
1: procedure ZOOM
2:     while True do
3:         $\alpha_j \leftarrow \textit{interpolate\_alpha}(\alpha_{lo}, \alpha_{hi})$;
4:         Evaluate $\mathcal{E}(\alpha_j)$;
5:         if $\left[\mathcal{E}(\alpha_j) > \mathcal{E}(0) + \sigma_1 \alpha_j \nabla\mathcal{E}_0^T d_0\right]$ or $[\mathcal{E}(\alpha_j) \leq \mathcal{E}(\alpha_{lo})]$ then
6:             $\alpha_* \leftarrow \alpha_j$;
7:             return $\alpha_*$;
8:         else
9:             Evaluate $\nabla\mathcal{E}_j^T d_j$;
10:            if $\left|\nabla\mathcal{E}_j^T d_j\right| \leq -\sigma_2 \nabla\mathcal{E}_0^T d_0$ then
11:                $\alpha_* \leftarrow \alpha_j$;
12:                return $\alpha_*$;
13:            if $\nabla\mathcal{E}_j^T d_j(\alpha_{hi} - \alpha_{lo}) \geq 0$ then
14:                $\alpha_{hi} \leftarrow \alpha_{lo}$;
15:            if $(|\mathcal{E}_j - \mathcal{E}_0| \leq \textit{threshold}$ then
16:                $\alpha_* \leftarrow \alpha_j$
17:                return $\alpha_*$;
18:        $\alpha_{lo} \leftarrow \alpha_j$;
```

Algorithm 5 Interpolate

```
1: procedure INTERPOLATE_ALPHA
2:     $i \leftarrow \mathit{1}$;
3:     while $i \leq \textit{max\_iter}$ do
4:         $\alpha_{mid} \leftarrow (\alpha_{hi} - \alpha_{lo})/2$
5:         Evaluate $\mathcal{E}(\alpha_{mid})$;
6:         if $[\mathcal{E}(\alpha_{mid}) == 0]$ or $[(\alpha_{hi} - \alpha_{lo})/2 < \textit{threshold}]$ then return $\alpha_{mid}$;
7:         Evaluate $\mathcal{E}(\_midalpha_{lo})$;
8:         if $sign(\mathcal{E}(\alpha_{mid})) == sign(\mathcal{E}(\alpha_{lo}))$ then
9:             $\alpha_{lo} \leftarrow \alpha_{mid}$;
10:        else
11:            $\alpha_{hi} \leftarrow \alpha_{mid}$;
12:        $i \leftarrow i + 1$;
```

Chapter 3

Test

3.1 Monk

3.2 Cup

Bibliography

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

[2] 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 (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

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

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

[5] 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.

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

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