Computational Mathematics for Learning and Data Analysis

Federico Finocchio
f.finocchio@studenti.unipi.it

A.Y. 2020/2021

Abstract

Assigned project: ML project 6
(M1) is a neural network with topology of your choice, but mandatory piecewise-linear activation function (of your choice); any regularization is allowed.
(M2) is a standard L_2 linear regression (min least squares).
(A1) is a standard momentum descent approach applied to (M1).
(A2) is an algorithm of the class of deflected subgradient methods applied to (M1).
(A3) is a basic version of the direct linear least squares solver of your choice (normal equations, QR, or SVD) applied to (M2).

1 Introduction

This report is about the project assigned for the course of Computational Mathematics for Learning and Data Analysis. All the work in this project is the result of the knowledge gathered from the courses of **ML** and **CM**. The report contents that are not directly work of the authors is referenced and, as requested, we point to the references down to chapter and number of page (when necessary).

We start by giving a short description of the problem at hand and the methods used to solve it, including all the mathematical derivation needed to adapt the chosen methods to the problem. Next, we give a brief recap of the expected results for the experiments, properties of the problem that suits our methods and details about the solvability of our problem with the used methods. In the end, we show the achieved results, comparing them with the expected one describing which are the factors that determined a difference in the results.

The models to be implemented are:

- **M1**: a neural network, *ANN* in the following, with piecewise-linear activation function, with possible regularization;
- **M2**: standard L_2 linear regression

The methods to be applied to the models are:

- **A1**: standard momentum descent approach applied to **M1**;
- **A2**: deflected subgradient methods applied to **M1**;
- **A3**: basic version of one of the direct linear least squares solvers (i.e. normal equations, QR, SVD) applied to **M2**.

In the following we describe the main implementation choices and introduce some of the notation used in the rest of the report. The detailed description of the implemented methods is given in the related sections of this document.

2 Artificial Neural Network

The implemented *ANN* can be seen as a *fully connected multilayer Perceptron*. An *ANN* is composed by an interconnection of units, each one of them can be represented as the composition of two functions that determines the output given the fixed weight vector and the input from the previous layer. The two functions are referred as the *network function* and the *activation function*, where the former computes the scalar product of the input vector with the weight vector of the current unit, the latter is the function that directly determines the output of the current unit. In our particular case the activation function is required to be a *piecewise-linear activation function*.

The *ANN* will be structured with multiple layers, each layer will have all the units fully connected with the adjacent layers and, as convention, we refer to the first layer as *input layer* and to the last layer as *output layer*. The others are referred as *hidden layers*. Another important aspect when implementing an *ANN* is the choice of the number of units. Later in this report we show how the exact number of units in each layer is chosen, but we can already describe the structure of the input and the output layer. The *input layer* will contain a number of units that is the same as the number of features contained in the data that will be fed up to the *ANN*, instead the *output layer* will depend on the task to perform.

To simplify the development of the *ANN* we plan to fix the number of hidden layer and the number of units per layer, changing only the input/output layers depending on the task to be performed. The process involved in the determination of this characteristic of the *ANN* will be described in the testing section.

In the following sections we describe in more details which are the main aspects of the implemented *ANN* like the network structure, the functions used to compute the output of each unit and the algorithm used to let the network learn the task at hand.

2.1 Activation function

The choice of the activation function is a crucial step for the construction of the *ANN*. This function directly determines what is the output of each of the units in the network, depending on the result of the scalar product of the received input vector and the unit weights vectors.
The activation function, for this project, is required to be a *pieceweise-linear function*, so the choice can be restricted between the two most popular among them:

- **ReLU**:
  - defined as: $f(x) = max(0, x)$;
  - we can impose the derivative to be: $f'(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$
- **Leaky ReLU**:
  - defined as: $f(x) = \begin{cases} \alpha x & x \leq 0 \\ x & x > 0 \end{cases}$
  - also in this case we can impose the derivative to be: $f'(x) = \begin{cases} \alpha & x < 0 \\ 1 & x \geq 0 \end{cases}$

  Where α determines the slope of the negative part of the function, and usually is chosen as $\alpha = 0.01$.

The main characteristics of these functions are:

- Sparse activation;
- Avoid vanishing gradient;
- Efficient computation;

These functions are widely used to deal with the problem of *vanishing gradients* when using the backpropagation algorithm described in §2.2. Our main choice is using the **Leaky ReLU** activation function due to its simplicity and to the fact that it can help avoiding the dying ReLU problem related to the *ReLU* activation function. The only problem is that it is not entirely differentiable, in particular there is only one point (i.e. $x = 0$) where the derivative is not defined. As shown in [1, Chap. 6.3] the slight modification of the derivative with $f'(0) = 1$ leads to good results and does not impair the convergence of the learning algorithm from a practical point of view. As we point out in **§4.1.1**, this modification is not enough to guarantee the optimization algorithm to theoretically converge.

2.2 Backpropagation algorithm

The backpropagation algorithm [see 1, Chap. 6.5.4], in a multi-layer neural network, is used to compute the gradient of the cost function. The resulting gradient is used by the learning algorithm to minimize the squared difference between the network output values $\hat{\mathbf{y}}$ and the target values $\mathbf{y}$ associated to these outputs. The backpropagation algorithm can then be used to efficiently compute the derivative of the *ANN* seen as a composition of functions.
This algorithm is also described in [2], [3] and is composed by two main parts:

- **Forward phase**: data traverse the network from the input units to the output units, in such a way the network's output value is generated and used to compute the cost function. The procedure is shown in **Algorithm 1**.
- **Backward phase**: the error is computed by comparing the network's output with the expected one. The computed error is then propagated back to all the network's layers. At each backward step the *Chain Rule of Calculus* is used to compute the partial derivative of the unit's function related to the current layer's weights. The gradient at each layer represents how much the current units are responsible for the total error and the result of the backward phase is used by the optimization algorithm to update the weight vector of each layer. This phase is defined in **Algorithm 2**.

Algorithm 1 Forward propagation

```
$\mathbf{h}_0 = \mathbf{x}$
for $k = 1, \dots, l$ do
    $\mathbf{net}_k = \mathbf{b}_k + \mathbf{W}_k \mathbf{h}_{k-1}$
    $\mathbf{h}_k = f(\mathbf{net}_k)$
end for
$\hat{\mathbf{y}} = \mathbf{h}_l$
$J = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \Omega(\theta)$
```

Algorithm 1 proceeds to compute the composition of functions that represents the network. The network's output value $\hat{\mathbf{y}}$ is produced by the *output layer*. Each of the $\mathbf{h}_i$ represents the output vector coming from the layer i. Once the predicted output is produced, the algorithm proceeds into computing the loss function $L(\hat{\mathbf{y}}, \mathbf{y})$ that estimates the error for the given output vector and, by adding this value to a regularizer $\Omega(\theta)$ we obtain the total cost J.

The gradient of the cost function is computed by the next algorithm and passed to the optimizer.

Algorithm 2 Backward computation

```
$\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y})$
for $k = l, l-1, \dots, 1$ do
    $\mathbf{g} \leftarrow \nabla_{\mathbf{net}_k} J = \mathbf{g} \odot f'(\mathbf{net}_k)$
    $\nabla_{\mathbf{b}_k} J = \mathbf{g} + \lambda \nabla_{\mathbf{b}_k} \Omega(\theta)$
    $\nabla_{\mathbf{W}_k} J = \mathbf{g} \mathbf{h}_{k-1}^T + \lambda \nabla_{\mathbf{W}_k} \Omega(\theta)$
    $\mathbf{g} \leftarrow \nabla_{\mathbf{h}_{k-1}} J = \mathbf{W}_k^T \mathbf{g}$
end for
```

The gradient produced as a result for the *backward phase* is used by the optimization algorithm to minimize the error function via automatic fine-tuning of the weight vector. Each layer proceeds to update its weight vector depending on the layer's contribute on the total error. The way the weight vectors are updated is determined by the type of optimization methods used. Detailed informations about the way the optimization algorithms work are given in the related section.

2.3 Loss function

The loss function is used to estimate the error at the output of the network. In a supervised learning approach the main aim is the minimization of the *Loss Function* via automatic tuning of the weight w_k at each layer k. In our case this is done via *subgradient method* and *standard momentum descent.*

We use the *MSE* as a measure of error, this is the averaged sum, over all available data, of the squared differences between the predicted value and the desired one.

This is obtained by:

$$MSE = \frac{1}{2N} \sum_{p=1}^{N} \sum_{k=1}^{K} (y_k - \widehat{y}_k)_p^2 \tag{1}$$

where N is the total number of examples our network is trained on and K is the total number of output units.

As described by [3, Chap. 4.3], the equation (1) changes based on the specific algorithm used to train the network. In particular for *batch learning* we have as *Loss Function* the one described above, instead for *online learning* the weights are updated on an *example-by-example* basis, so the function to be minimized is the instantaneous error computed for each pattern that flows into the network. In the following we show the derivation of the *Loss Function*, noting that for the *Stochastic learning* the minimization is performed only on the instantaneous error and for *Batch learning* the minimization is performed on the averaged error over the used patterns.

2.3.1 Derivation of the loss function

We start by defining:

$$E_{tot} = \frac{1}{N} \sum_{p=1}^{N} E_p$$

$$E_p = \frac{1}{2} \sum_{k=1}^{K} (y_k - \widehat{y}_k)^2$$

where E_{tot} is the average error over all the used samples and E_p is the instantaneous error for a given pattern p.

The ∇w to be used by the optimization algorithm is equal to:

$$\nabla w = -\frac{\partial E_{tot}}{\partial w} = -\frac{1}{N} \sum_{p=1}^{N} \frac{\partial E_p}{\partial w} = \frac{1}{N} \sum_{p=1}^{N} \nabla_p w$$

The $\nabla_p w$ for a generic unit t with inputs coming from unit i is equal to:

$$\nabla_p w_{t,i} = -\frac{\partial E_p}{\partial w_{t,i}} = -\frac{\partial E_p}{\partial o_t} * \frac{\partial o_t}{\partial net_t} * \frac{\partial net_t}{\partial w_{t,i}}$$

by defining:

$$\begin{cases} o_t = f_t(net_t), \\ net_t = \sum_{j \in C} w_{t,j} o_j \end{cases}$$

where, for a generic unit t, o_t is the output of the unit, f_t is the activation function, net_t is the network function and C is the set of all the units that are giving an input to the current one, we have that:

$$\frac{\partial o_t}{\partial net_t} = f_t{}'(net_t), \quad \text{and} \quad \frac{\partial net_t}{\partial w_{t,i}} = \frac{\partial \sum_{j \in C} w_{t,j} o_j}{\partial w_{t,i}} = o_i$$

By defining:

$$\delta_t = -\frac{\partial E_p}{\partial net_t} = -\frac{\partial E_p}{\partial o_t} * \frac{\partial o_t}{\partial net_t}$$

We have to study two different cases for $-\frac{\partial E_p}{\partial o_t}$, depending on wether o_t is the output coming from an output unit or an hidden unit.

Case t output unit:

$$\begin{aligned}
-\frac{\partial E_p}{\partial o_t} &= -\frac{1}{2} * \frac{\sum_{k=1}^{K} \partial((y_k - \widehat{y}_k)^2)}{\partial o_t} = (y_t - \hat{y}_t), \text{ and} \\
\delta_t &= -\frac{\partial E_p}{\partial net_t} = (y_t - \hat{y}_t) * {f_t}'(net_t)
\end{aligned}$$

We finally have that:

$$\nabla_p w_{t,i} = \delta_t * o_i = (y_t - \hat{y}_t) * {f_t}'(net_t) * o_i$$

Case t hidden unit: Since a generic hidden unit t contributes to the output generated by all the units k in the layer to the immediate right of t, to estimate its contribution on the network error we use the propagated error δ_k:

$$\begin{aligned}
-\frac{\partial E_p}{\partial o_t} &= \sum_{k=1}^{K} -\frac{\partial E_p}{\partial net_k}\frac{\partial net_k}{o_t} = \sum_{k=1}^{K} \delta_k w_{k,t}, \text{ and} \\
\delta_t &= \sum_{k=1}^{K} \delta_k w_{k,t} * {f_t}'(net_t)
\end{aligned}$$

where each δ_k is the exact result obtained in the previous step of backward computation for the units connected to t. This represents a backward step and is the core of the backpropagation algorithm.

We finally have that:

$$\nabla_p w_{t,i} = \sum_{k=1}^{K} \delta_k w_{k,t} * {f_t}'(net_t) * o_i$$

2.3.2 Properties of loss function

In this section we describe general properties of the chosen loss function, in particular discussing continuity, differentiability and convexity. We have that:

- **Continuity**: the loss function used is represented by the sum of square functions composed with the *ANN* function. Given that the *ANN* can be represented as a composition of Lipschitz continuous functions, in particular *ReLU* activation function and the linear function $W_k \hat{y}_{k-1} + b_k$ at each layer k, using [4, Claim 12.7] we can say that the network function is a Lipschitz continuous function. Considering the fact that a square function is Lipschitz continuous only if the input set is bounded, noting that the *ReLU* function output is not bounded, we can conclude that our loss function is not Lipschitz continuous unless we provide a bound on the output values of the different *ReLU* activation functions.
- **Convexity**: we first study convexity of the *ANN* by representing it as a composition of functions. The functions that builds up the network are composition of the *ReLU* activation functions, which are convex, with the linear function $W_k \hat{y}_{k-1} + b_k$ for each layer k. As seen in [5, Chap. 3.2.4], given that the composition $f = h \circ g : \mathbb{R}^n \rightarrow \mathbb{R}$ of two functions $h : \mathbb{R} \rightarrow \mathbb{R}$ and $g : \mathbb{R}^n \rightarrow \mathbb{R}$ is convex if:
  - h is convex;
  - h is increasing;
  - g is convex.

  In our case, the *ANN* is convex. However, the MSE is the composition of convex functions, but noting that the square function is increasing only for positive values, it means that in our case the loss function is not convex.

- **Differentiability**: using *piecewise-linear functions* as activation functions for the *ANN* leads the loss function to be non-differentiable, however using the assumption made in §2.1, i.e. fixing the value of the activation function's derivative in the points where this is not differentiable, from a practical point of view, does not impair convergence of the algorithm and allows to use the backpropagation algorithm to compute the gradient of the network. The requirement on the differentiability of the loss functions is not needed for the subgradient method as it will be described in a later section of this document.

Further discussions about the properties of the loss function are needed for the implementation of the subgradient methods, in fact we know that it requires the optimized function to be convex and Lipschitz continuous.

3 Least Square

The *Least Square problem* is described in [6, Lecture 11] and [7, Chap. 3] as the problem of finding a solution of an overdetermined system of equations $Ax = b$ by finding a vector x that minimizes the 2-norm of the residual vector defined as $r = b - Ax$.

The *Least Square problem*, given $A \in \mathbb{R}^{m \times n},\ m \geq n,\ b \in \mathbb{R}^m$, has the following form:

$$\text{find } x \in \mathbb{R}^n \text{ that solves the minimization problem } \min_x \|b - Ax\|_2 \ . \tag{2}$$

We describe in section §4.3, related to the implemented direct solver, that this kind of problems have a unique solution if the matrix A has *full column rank*.

4 Methods

This section gives detailed information about the required methods that will be implemented and applied to the models described in §2 and §3. The main methods to be implemented are:

- Standard momentum descent approach applied to the *ANN*;
- Deflected subgradient method applied to the *ANN*;
- Direct linear least square solver applied to the *Least Square problem*.

4.1 Momentum method

The momentum approach is a technique that accelerates the gradient descent accumulating a velocity vector in directions of persistent reduction [8].

We can define *Classical Momentum* (CM) as:

$$v_{t+1} = \mu v_t - \epsilon \nabla f(\Theta_t) \tag{3}$$

$$\Theta_{t+1} = \Theta_t + v_{t+1} \tag{4}$$

where $\epsilon > 0$ is the learning rate, $\mu \in [0, 1)$ is the momentum coefficient and $\nabla f(\Theta_t)$ is the gradient at Θ_t. The hyperparameter of momentum determines how quickly the contributions of previous gradients exponentially decay. Another important aspect is that the larger μ is, relative to ϵ, the more previous gradients affect the current direction. At the moment, no prior choices of the momentum coefficient and learning rate can be done, these will be studied more in the detail in a later phase of the project where we implement and test these models, but as suggested by [1, Chap. 8], common values for μ are $0.5, 0.9, 0.99$.

The next algorithm is the pseudocode relative to the *Gradient descent with momentum*, given the learning rate and the momentum coefficient, performs a descent approach until termination conditions are met.

Algorithm 3 Gradient descent with momentum. Termination conditions, learning rate and momentum coefficients have to be determined by testing as it will be shown in a later phase of the project. For the moment we assume that they are given.

```
 1: Initialize $\Theta$ and $\mathbf{v}$
 2: while termination conditions not met do
 3:     Sample $m$ examples $(x_1, y_1), (x_2, y_2) \dots (x_m, y_m)$
 4:     $\tilde{\Theta} \leftarrow \Theta$
 5:     if Nesterov then
 6:         $\tilde{\Theta} \leftarrow \tilde{\Theta} + \mu\mathbf{v}$
 7:     end if
 8:     Compute gradient estimate: $\mathbf{g} \leftarrow \frac{1}{m}\nabla_{\tilde{\Theta}} \sum_i (L(f(x_i, \tilde{\Theta}), y_i))$
 9:     Compute velocity update: $\mathbf{v} \leftarrow \mu\mathbf{v} - \epsilon g$
10:     Apply update: $\Theta \leftarrow \Theta + \mathbf{v}$
11: end while
```

In Algorithm 3 the amount of samples taken at *line 3* determines the type of *Gradient descent algorithm*, such as:

- $\mathbf{m} = \mathbf{1}$: *stochastic gradient descent (SGD)*;
- $\mathbf{m} < \mathbf{n}$, where n is the total number of examples: *batch stochastic gradient descent*;
- $\mathbf{m} = \mathbf{n}$: *standard gradient descent (GD).*

A further improvement is given by a modification of the *CM* approach, called *Nesterov's Accelerated Gradient* (NAG) that, seen as a momentum approach, can be defined as:

$$v_{t+1} = \mu v_t - \epsilon \nabla f(\Theta_t + \mu v_t) \tag{5}$$

$$\Theta_{t+1} = \Theta_t + v_{t+1} \tag{6}$$

The only difference from *CM* is relative to the point where the gradient is computed, in fact NAG performs a partial update to Θ_t and uses this update to compute the gradient at step t. After computing the gradient, the update rule is the same, but in this way NAG allow changing v in a more responsive way. This modification is implemented in Algorithm 3 at *line 6* where an update to the point of evaluation of the gradient is performed. Differences between *CM* and *NAG* can be seen in **Figure 1**.

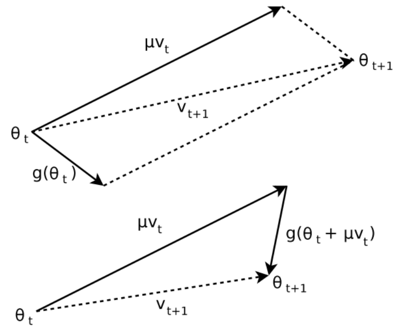

Figure 1: *CM (on top) shows how the update of the vector v_t using the gradient at Θ_t differs from the NAG (on bottom). When μv_t is a poor update, we can see how NAG points v_{t+1} back towards Θ_t more strongly than CM.*

4.1.1 Convergence

As shown in [8], *NAG* can help avoiding oscillations in the path taken by *CM*. In **Figure 2** we can see the comparison between the two momentum methods with same momentum and learning rate coefficients. This shows how the correction rule for a poor update, as shown in Equation 5, over multiple iterations, can help *NAG* to be more effective than *CM* at decelerating over time. This also helps *NAG* to be more tolerant to larger values of μ compared to *CM*.

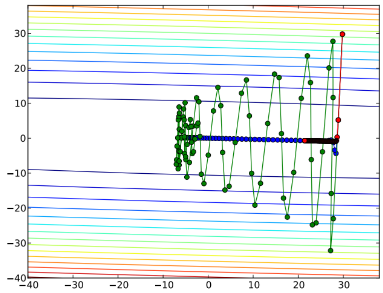

Figure 2: *Given the global minimizer of the objective function in (0,0), the red, blue and green curves show, respectively, the trajectories of gradient descent, NAG and CM. It's clearly visible the difference in oscillations between CM and NAG approach.*

To study convergence of the *SGD* involving *CM* and *NAG*, applied to a non-convex objective function, we refer to the results obtained in [9] that describes a unified framework for both momentum methods. The unified framework uses the constant $s = 0$ and $s = 1$ to refer, respectively, to *CM* and *NAG*. By defining $C > 0$ a positive constant and $G_k = G(x_k; \xi_k)$ a stochastic gradient of $f(x)$ at x_k, depending on a random variable ξ_k, such that $\mathbb{E}[G(x_k; \xi_k)] = \nabla f(x_k)$, the following theorem shows the convergence of *SGD* with momentum for a non-convex objective function $f(x)$.

Theorem 4.1. *Suppose $f(x)$ is a non-convex and L-smooth function, $\mathbb{E}[\| G(x;\xi) - \nabla f(x)\|_2^2] \leq \delta^2$ and $\|\nabla f(x)\|_2 \leq B$ for any x. Let the stochastic gradient descent method run for t iterations. By setting $\epsilon = \min\{\frac{1-\mu}{2L[1+((1-\mu)s-1)^2]}, \frac{C}{\sqrt{t+1}}\}$ we have:*

$$\min_{k=0,\dots,t} \mathbb{E}[\|\nabla f(x_k)\|_2^2] \leq \frac{2(f(x_0) - f^*)(1-\mu)}{t+1} \max\left\{\frac{2L[1+((1-\mu)s-1)^2]}{1-\mu}, \frac{\sqrt{t+1}}{C}\right\} + \frac{C}{\sqrt{t+1}} \frac{L\mu^2(B^2+\delta^2) + L\delta^2(1-\mu)^2}{(1-\mu)^3}$$

This theorem shows that the gradient norm converges in expectation at $\mathcal{O}(\frac{1}{\sqrt{t}})$ [9]. Additionally, **Theorem 4.1** suggests that for *NAG* ($s = 1$) we can set a larger initial learning rate than that of *CM* ($s = 0$), as also shown in [9, Section 4], which leads to a faster convergence in training error.

However, noting that in our case the assumptions made for **Theorem 4.1** do not hold, in particular our objective function is not continuously differentiable and not Lipschitz continuous, we can't use this result to prove convergence for our setting. At the moment we are not able to state anything about the converge, in practice, of our algorithm with our specific setting (i.e. non-convex, non-differentiable objective function), so we postpone this discussion in the testing phase.

4.2 Subgradient Method

The *subgradient method* (*SM*) is a minimization method used to minimize non-differentiable convex objective functions. It is not a descent method, the value of the function is not decreasing at every step, in fact the direction negative to a subgradient is not necessarily a direction of descent of the function $f(\cdot)$ [10].

Given $f : \mathbb{R}^n \to \mathbb{R}$ a convex function not necessarily smooth, to minimize f the *SM* constructs a sequence of iterates $\{x_k\}$ by the iterative formula:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k g_k$$

where $g_k \in \partial f(x_k)$ is any subgradient of $f(\cdot)$ at x_k, $\alpha_k > 0$ is the kth stepsize and x_k is the iterate at the step k. It is worthwile to recall that a subgradient of f at x is any vector g that satisfies the inequality $f(y) \geq f(x) + g^T(y - x)$ for all y.

We show a pseudocode for the *SM*:

Algorithm 4 Basic subgradient method. Assuming starting point x_1 and subgradient at each point are given.

```
Initialize $x_1$
Starting upper bound: $UB_1 \leftarrow f(x_1)$
Starting optimal point: $x^* \leftarrow x_1$
$k \leftarrow 1$
while termination conditions not met do
    Find a subgradient of $f$ in $x_k$: $g_k \in \partial f(x_k)$
    if $g_k = 0$ then
        Terminate with $x^* = x_k$
    end if
    Select a direction: $d_k \leftarrow -g_k / \|g_k\|_2$
    Select a step size: $\alpha_k > 0$
    $x_{k+1} \leftarrow x_k + \alpha_k d_k$
    if $f(x_{k+1}) < UB_k$ then
        $UB_{k+1} \leftarrow f(x_{k+1})$
        $x^* \leftarrow x_{k+1}$
    else
        $UB_{k+1} \leftarrow UB_k$
    end if
    $k \leftarrow k + 1$
end while
```

As shown in [11, Chap. 8.9], however, the stopping criterion $g_k = 0$ may never be realized because the algorithm selects the subgradient g_k arbitrarily. Usually, a stopping criterion is imposing a limit on the number of iterations performed by the algorithm. If we know the optimal value, which in general is unknown, we can impose the algorithm to stop when we reach a desired accuracy $UB_k < f^* + \epsilon$.

4.2.1 Convergence

We start with the assumption that *SMs* are feasible only for those problems that do not require a high accuracy, as shown in [12]. In fact, *SM* requires $\Theta(1/\epsilon^2)$ iterations to attain an absolute error up to ϵ. We can also note that the complexity does not depend on the size of the problem.

To study converge we first look for a bound on the distance to the optimal set, assuming that there exist an optimal solution, by [10, Theorem 7.4] we have:

Theorem 4.2. *Let the subgradient method use non-negative step sizes $\{\alpha_k\}$ such that*

$$\sum_{k=1}^{\infty} \alpha_k = \infty \ \textit{and,} \ \sum_{k=1}^{\infty} \alpha_k^2 < \infty. \tag{7}$$

Then the sequence $\{x_k\}$ generated by the subgradient method is convergent to a solution of the problem.

Proof. By assuming x^* is an optimal solution and considering that $f(x_k) - f(x^*) \geq 0$, then:

$$\begin{aligned} \|x_{k+1} - x^*\|_2^2 &= \|x_k - \alpha_k g_k - x^*\|_2^2 \\ &= \|x_k - x^*\|_2^2 - 2\alpha_k \langle g_k \,, x_k - x^* \rangle + \alpha_k^2 \|g_k\|_2^2 \\ &\leq \|x_k - x^*\|_2^2 - 2\alpha_k (f(x_k) - f(x^*)) + \alpha_k^2 \|g_k\|_2^2 \\ &\leq \|x_k - x^*\|_2^2 + \alpha_k^2 \|g_k\|_2^2 \,, \end{aligned} \tag{8}$$

where the third equation of (8) comes from $g_k^T(x_k - x^*) \geq f(x_k) - f(x^*)$, that follows from the definition of subgradient. By induction on k we have:

$$\|x_k - x^*\|_2^2 \leq \|x_0 - x^*\|_2^2 - 2\sum_{l=0}^{k-1} \alpha_l (f(x_l) - f^*) + \sum_{l=0}^{k-1} \alpha_l^2 \|g_l\|_2^2 \tag{9}$$

$$\leq \|x_0 - x^*\|_2^2 + \sum_{l=0}^{k-1} \alpha_l^2 \|g_l\|_2^2 \tag{10}$$

$$\leq \|x_0 - x^*\|_2^2 + \sum_{l=0}^{\infty} \alpha_l^2 \|g_l\|_2^2 \tag{11}$$

By **(7)**, the sum in **(11)** is bounded, thus the sequence $\{x_k\}$ is bounded. Using the result from [10, Theorem 7.2] with *learning rate* $\bar{\tau} = 0$, there exists an infinite set of iterations $\mathcal{K}$ such that for $k \in \mathcal{K}$, as $k \to \infty$, we have $f(x_k) \to f(x^*)$. We can choose an infinite set $\mathcal{K}_1 \subset \mathcal{K}$ such that the subsequence $\{x_k\}$, with $k \in \mathcal{K}_1$, is convergent to $\hat{x}$ which must be an optimal solution and can be substituted to x^*. Choosing $l \in \mathcal{K}_1$, adding inequalities (8) from $k = l$ to m we obtain:

$$\|x_{m+1} - \hat{x}\|_2^2 \leq \|x_l - \hat{x}\|_2^2 + \sum_{k=l}^{\infty} \alpha_k^2 \|g_k\|_2^2 \quad m = l+1, l+2, \dots .$$

For each $\epsilon > 0$ we can choose $l \in K_1$ such that $\|x_l - \hat{x}\|_2^2 \leq \epsilon$, and $\sum_{k=l}^{\infty} \alpha_k^2 \|g_k\|_2^2 \leq \epsilon$. Then $\|x_{m+1} - \hat{x}\|_2^2 \leq 2\epsilon$ for all $m \geq l$, which proves that the entire sequence $\{x_k\}$ is convergent to $\hat{x}$. □

To study convergence rate we refer to [13, Theorem 3.1]:

Theorem 4.3. *When $f : \mathbb{R}^n \to \mathbb{R}$ is convex and its subgradients are bounded by L, for any $g \in \partial f(x)$ at any x, subgradient descent starting at x_0 s.t. $\|x_0 - x^*\|_2 \leq R$ with step size $\alpha_l \leftarrow \frac{R}{\sqrt{k}\|g_l\|_2}$ satisfy $f_k^* - f^* \leq \frac{LR}{\sqrt{k}}$ after k iterations.*

Proof. Rearranging (9) we have:

$$\begin{aligned} 2\sum_{l=0}^{k-1} \alpha_l (f(x_l) - f^*) &\leq \|x_0 - x^*\|_2^2 - \|x_k - x^*\|_2^2 + \sum_{l=0}^{k-1} \alpha_l^2 \|g_l\|_2^2 \\ &\leq \|x_0 - x^*\|_2^2 + \sum_{l=1}^{k-1} \alpha_l^2 \|g_l\|_2^2 \end{aligned} \tag{12}$$

Let $\beta_l = \alpha_l \|g_l\|_2$, $f_k^* = \min_{l<k} f(x_l)$. By the result obtained in (12) at the k-th iteration, we have:

$$2(f_k^* - f^*)\sum_{l=0}^{k-1} \frac{\beta_l}{\|g_l\|_2} \leq \|x_0 - x^*\|_2^2 + \sum_{l=0}^{k-1} \beta_l^2 \leq R^2 + \sum_{l=0}^{k-1} \beta_l^2.$$

Since the subgradients are bounded by L:

$$\frac{2}{L}(f_k^* - f^*)\sum_{l=0}^{k-1} \beta_l \leq R^2 + \sum_{l=0}^{k-1} \beta_l^2$$

By rearranging, we obtain

$$f_k^* - f^* \leq \frac{R^2 + \sum_{l=1}^{k} \beta_l^2}{\frac{2}{L}\sum_{l=1}^{k} \beta_l}$$

Since the bound is symmetric in $\{\beta_l\}$, the bound is minimized when all the β_ls are equal. For a given k, the bound is minimized at $\frac{R}{\sqrt{k}}$. The optimized bound is $f_k^* - f^* \leq \frac{LR}{\sqrt{k}}$. □

Thus, *subgradient method* attains $\mathcal{O}(\frac{1}{\sqrt{k}})$-suboptimality after k iterations, that means it obtains an ϵ-suboptimal point after at most $\mathcal{O}(\frac{1}{\epsilon^2})$ iterations.

However, as seen in §2.3.2, our loss function is not convex, so we can't state anything at this point about convergence of our setting, given that it doesn't respect the assumptions made for **Theorem 4.2** and **Theorem 4.3**. We postpone the discussion about practical convergence of the algorithm in the testing section.

4.3 Direct solver for Linear Least Square

We have chosen to implement the direct solver via *QR factorization*. This section gives a description of all the properties needed for a *least square problem* to be solved via this method and all the expected results for this kind of implementation. In a successive section we plan to insert the comparison between the theoretical results shown in this section and the actual result obtained in testing the implemented algorithm.

4.3.1 QR factorization

As described in [7, Chap. 5], *QR decomposition* (or factorization) is a factorization of a matrix A in a product of an orthogonal matrix and a triangular matrix obtained via successive orthogonal transformations.

Theorem 4.4. *Any matrix $A \in \mathbb{R}^{m\times n}$, $m \geq n$, can be transformed to upper triangular form by an orthogonal matrix. The transformation is equivalent to a decomposition*

$$A = Q\begin{bmatrix} R \\ 0 \end{bmatrix},$$

where $Q \in \mathbb{R}^{m\times m}$ is orthogonal and $R \in \mathbb{R}^{n\times n}$ is upper triangular. If the columns of A are linearly independent, then R is nonsingular.

If we partition $Q = (Q_1\ Q_2)$ where $Q_1 \in \mathbb{R}^{m\times n}$, noting that in the multiplication Q_2 is multiplied by zero, we can write:

$$A = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}\begin{bmatrix} R \\ 0 \end{bmatrix} = Q_1 R, \tag{13}$$

where equation (13) refers to the **thin QR factorization**. This form of the *QR factorization* is the one used from now on to solve the *linear least square problem.*

To implement this factorization we make use of the *Householder transformations* described in [7, Chap. 4.2.2].

Lemma 4.5. *For every* $\mathbf{v} \in \mathbb{R}^m$*, the matrix* $\mathbf{H} = I - \frac{2}{v^T v} vv^T = I - \frac{2}{\|v\|_2^2} vv^T = I - 2uu^T$*, (where* $u = \frac{1}{\|v\|_2} v$ *has norm 1) is orthogonal. We call* $\mathbf{H}$ *an Householder transformation.*

Lemma 4.6. *Let* x, y *be two vectors such that* $\|x\|_2 = \|y\|_2$*. If one chooses* $v = x - y$*, then* $H = I - \frac{2}{v^T v} vv^T$ *is such that* $Hx = y$*.*

By choosing $y = \|x\|_2 \, e_1 = \begin{bmatrix} \|x\|_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ we can build a procedure to find the householder vector $\mathbf{u}$ of a generic vector $\mathbf{x}$. The pseudocode for this procedure is shown in Algorithm 5.

Algorithm 5 Householder vector

```
1: $\mathbf{s} \leftarrow norm(x)$
2: $\mathbf{v} \leftarrow x$
3: $\mathbf{v}[1] \leftarrow \mathbf{v}[1] - s$
4: $\mathbf{u} \leftarrow \mathbf{v}/norm(\mathbf{v})$
```

Now we illustrate the method used to compute the QR factorization through the *Householder transformation.* By a sequence of orthogonal transformation we can transform any matrix $A \in \mathbb{R}^{m \times n}, m \geq n$,

$$A \to Q^T A = \begin{bmatrix} R \\ 0 \end{bmatrix}, R \in \mathbb{R}^{n \times n}$$

where R is upper triangular and $Q \in \mathbb{R}^{m \times m}$ is orthogonal. As shown in [7, Chap. 5.1] we can illustrate the procedure using a smaller matrix $A \in \mathbb{R}^{5 \times 4}$. Basically the algorithm proceeds by zeroing the elements under the main diagonal, where at each step i the elements below the element $a_{i,i}$ are zeroed by left-multiplying the current matrix A_i to a matrix H_{i+1}.

In the first step we zero the elements below the main diagonal in the first column:

$$H_1 A = H_1 \begin{pmatrix} \mathrm{x} & \mathrm{x} & \mathrm{x} & \mathrm{x} \\ \mathrm{x} & \mathrm{x} & \mathrm{x} & \mathrm{x} \\ \mathrm{x} & \mathrm{x} & \mathrm{x} & \mathrm{x} \\ \mathrm{x} & \mathrm{x} & \mathrm{x} & \mathrm{x} \\ \mathrm{x} & \mathrm{x} & \mathrm{x} & \mathrm{x} \end{pmatrix} = \begin{pmatrix} + & + & + & + \\ 0 & + & + & + \\ 0 & + & + & + \\ 0 & + & + & + \\ 0 & + & + & + \end{pmatrix} = A_1,$$

where $+$ denotes an element that has changed in the transformation. The orthogonal matrix H_1 can be taken equal to a *Householder transformation.* In the second step we use an embedded *Householder transformation* to zero the elements below the diagonal of the second column of matrix A_1:

$$H_2 A_1 = H_2 \begin{pmatrix} \mathrm{x} & \mathrm{x} & \mathrm{x} & \mathrm{x} \\ 0 & \mathrm{x} & \mathrm{x} & \mathrm{x} \\ 0 & \mathrm{x} & \mathrm{x} & \mathrm{x} \\ 0 & \mathrm{x} & \mathrm{x} & \mathrm{x} \\ 0 & \mathrm{x} & \mathrm{x} & \mathrm{x} \end{pmatrix} = \begin{pmatrix} \mathrm{x} & \mathrm{x} & \mathrm{x} & \mathrm{x} \\ 0 & + & + & + \\ 0 & 0 & + & + \\ 0 & 0 & + & + \\ 0 & 0 & + & + \end{pmatrix} = A_2$$

And so on, after the fourth step we have computed the upper triangular matrix R. The sequence of transformations is summarized as:

$$Q^TA = \begin{bmatrix} R \\ 0 \end{bmatrix}, \; Q^T = H_4H_3H_2H_1.$$

Assuming $A \in \mathbb{R}^{m \times n}$ the matrices H_i have the following structure:

$$\begin{aligned} H_1 &= I - 2u_1u_1^T, \; u_1 \in \mathbb{R}^m \\ H_2 &= \begin{pmatrix} 1 & 0 \\ 0 & P_2 \end{pmatrix}, \; P_2 = I - 2u_2u_2^T, \; u_2 \in \mathbb{R}^{m-1} \\ H_3 &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & P_3 \end{pmatrix}, \; P_3 = I - 2u_3u_3^T, \; u_3 \in \mathbb{R}^{m-2} \end{aligned}$$

Thus vectors $\mathbf{u_i}$, obtained with the procedure defined in Algorithm 5, become shorter at each step and we embed *Householder transformations* of increasingly smaller dimensions in identity matrices.

4.3.2 Solving Least Square via QR factorization

In this section we show how *QR factorization*, shown in §4.3.1, can be used to solve the *linear least square problem* defined in equation (2).
In the following we use the fact that the Euclidean vector norm is invariant under orthogonal transformations, i.e. $\|Qy\|_2 = \|y\|_2$.

Theorem 4.7. *Let the matrix $A \in \mathbb{R}^{m \times n}$ have full column rank and thin QR decomposition $A = Q_1R$. Then the least squares problem* $\min_x \|b - Ax\|_2$ *has the unique solution*

$$x = R^{-1}Q_1^Tb.$$

Proof. Introducing the QR decomposition of A in the residual vector, we get

$$\|r\|_2^2 = \|b - Ax\|_2^2 = \left\| b - Q \begin{bmatrix} R \\ 0 \end{bmatrix} x \right\|_2^2 = \left\| Q^Tb - Q^TQ \begin{bmatrix} R \\ 0 \end{bmatrix} x) \right\|_2^2 = \left\| Q^Tb - \begin{bmatrix} R \\ 0 \end{bmatrix} x \right\|_2^2$$

Then we partition $Q = (Q_1 \; Q_2)$, where $Q_1 \in \mathbb{R}^{m \times n}$, so we can write

$$\|r\|_2^2 = \left\| \begin{bmatrix} Q_1^Tb \\ Q_2^Tb \end{bmatrix} - \begin{bmatrix} Rx \\ 0 \end{bmatrix} \right\|_2^2 = \left\| \begin{bmatrix} Q_1^Tb - Rx \\ Q_2^Tb \end{bmatrix} \right\|_2^2 \tag{14}$$

Under the assumption that the columns of A are linearly independent and since the bottom block of equation (14) is independent from vector x, we can solve $Rx = Q_1^Tb$ and minimize $\|r\|_2^2$ by making the upper block in equation (14) equal to zero. □

As shown in [6, Lecture 10] and [7, Chap. 5.3], in the solution of (14) there is no need in computing Q because we only need the product Q_1^Tb to solve the reduced problem $Rx = Q_1^Tb$ as described by **Theorem 4.7**. We can exploit this fact and apply the following two algorithms to obtain the upper triangular matrix R (Algorithm 6) and the product Q_1^Tb (Algorithm 7).

Algorithm 6 Householder QR factorization

```
for $k = 1, 2, \dots, min(m, n)$ do
    $[\mathbf{u}, \mathbf{s}] \leftarrow householder\_vector(\mathbf{A_{k:end,k}})$
    $\mathbf{A_{j,j}} \leftarrow \mathbf{s}$
    $\mathbf{A_{j+1:end,j}} \leftarrow 0$
    $\mathbf{A_{j:end,j+1:end}} \leftarrow \mathbf{A_{j:end,j+1:end}} - 2u(u^T(\mathbf{A_{j:end,j+1:end}}))$
end for
```

Algorithm 7 Implicit computation of $Q_1^T b$

```
for $k = 1, 2, \ldots, n$ do
    $\mathbf{b_{k:m}} \leftarrow \mathbf{b_{k:m}} - 2\mathbf{v_k}(\mathbf{v_k^T}\mathbf{b_{k:m}})$
end for
```

As a final step we give the total amount of operations needed (for a *thin QR factorization*) to solve the *Least Square problem* via QR factorization using Householder transformations. As shown in [7, Chap. 5.4], we can estimate the number of flops for computing R approximately with:

$$4\sum_{k=0}^{n-1}(m-k)(n-k) \approx 2mn^2 - \frac{2n^3}{3} + \mathcal{O}(mn),$$

and we can state the behaviour in two common regimes:

- Square matrices ($\mathbf{m} = \mathbf{n}$): $\frac{4}{3}n^3$
- $\mathbf{m} \gg \mathbf{n}$: scales like $2mn^2$

5 Experiments

We plan to use the *MONK* datasets to test and compare our methods. The dataset is:

Inserire informazioni sul dataset e su come si intende trattarlo

The necessary transformations to the original dataset are:

- 1-hot encoding o qualcosa di simile
- normalizzazione
- rimozione di valori nulli

For what concerns the tuning of the network we plan to use:

- cross-validation to study the best hyperparameters of the network
- grid search to search into the parameter space after choosing the "optimal" range of values for each hyperparameter
- final training of the network with convergence speed

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[2] Thomas M. Mitchell. *Machine Learning*. 1st ed. McGraw-Hill, Inc., 1997.

[3] Simon S. Haykin. *Neural networks and learning machines*. 3rd ed. Prentice Hall, 2009.

[4] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. USA: Cambridge University Press, 2014.

[5] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[6] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997.

[7] Lars Elden. *Matrix methods in data mining and pattern recognition*. Vol. 4. SIAM, 2007.

[8] Ilya Sutskever et al. “On the importance of initialization and momentum in deep learning”. In: *Proceedings of the 30th International Conference on Machine Learning*. Vol. 28. PMLR, 2013, pp. 1139–1147. URL: http://proceedings.mlr.press/v28/sutskever13.html.

[9] Tianbao Yang, Qihang Lin, and Zhe Li. *Unified Convergence Analysis of Stochastic Momentum Methods for Convex and Non-convex Optimization*. 2016. arXiv: 1604.03257 [math.OC].

[10] Andrzej Ruszczynski. *Nonlinear Optimization*. USA: Princeton University Press, 2006. ISBN: 0691119155.

[11] M. S. Bazaraa, Hanif D. Sherali, and C. M. Shetty. *Nonlinear programming: theory and algorithms*. 3rd ed. Wiley-Interscience, 2006.

[12] Antonio Frangioni, Bernard Gendron, and Enrico Gorgone. “On the computational efficiency of subgradient methods: a case study with Lagrangian bounds”. In: *Mathematical Programming Computation* 9 (2017).

[13] Yuekai Sun. *Notes on first-order methods for minimizing non-smooth functions*. 2015. URL: http://web.stanford.edu/class/msande318/notes/notes-first-order-nonsmooth.pdf.