Computational Mathematics Project

Federico Rossetto, Silvia Severini

April 24, 2018

ABSTRACT

For this project we implemented and tested some predictive models on the Machine Learning *Cup Dataset*. The models we implemented are a Neural Network and a Standard Linear Regression. For the first one, we explored a *Standard Gradient Descent* approach with momentum and a *Deflected sub-gradient* method with the *Polyak* step size. The second model it's a standard Linear Regression where we experimented an SVD-based algorithm with regularization.

Contents

1 Introduction

Our goal for this project is to implement and test some predictive models on the Machine Learning *Cup Dataset*. This dataset is composed of a table of dimensions 1016×12. The target values are the last 2 columns, so the predictive function is of the form $f : \mathbb{R}^{10} \to \mathbb{R}^2$.
The models we implemented are a Neural Network and a Standard Linear Regression.
For the first one we created two different learning algorithms, two different types of regularization and three different Model Selection Algorithms that tune the parameters of each learning algorithm.
For the Standard Linear Regression model, we chose the SVD algorithm to solve the Least Square problem associated with regularization.

2 Neural Network

We now briefly describe the Neural Network we implemented for the "Machine Learning" class.
The following are the variables that we used:

- n: number of samples
- a: number of attributes
- o: number of outputs
- m_i: number of Hidden Units on i^{th} layer
- $m_0 = a$, $m_{h+1} = o$
- h: number of Hidden Layers

In this context, we will refer to (X, D) as our **Dataset** where X and D are two matrices, such that $X \in \mathbb{R}^{n \times a}, D \in \mathbb{R}^{n \times o}$.
Then we define our set of matrices that holds the weights used for our Neural Network. We define this as $W = \{W_i \mid W_i \in \mathbb{R}^{m_i \times m_{i+1}}\}$. Each i-th matrix holds the weights of the connections from the layer i to the layer $i + 1$.
We will also use $activate(x)$ as our **Activation Function**. This function is used to compute the output of each neuron.
After this basic definitions, we can describe our Neural Network function. We define $W(i) = \{W_j | W_j \in W, j \leq i\}$. Then we define the NN recursively:

$$NN(W, x) = R^h(W(h)), \qquad R^i(W(i)) = activate(W_i * R^{i-1}(W(i-1))), \qquad R^0 = x \tag{1}$$

In the end we only need to define our objective function that we will want to minimize. We will use the MSE with an L_1 regularization.

$$E(W) = \|d - NN(W)\|_2^2 + \lambda \sum_i \|W_i\|_1 \tag{2}$$

We will also compare this model with an L_2 regularization, so only the regularization term will change, using $\sum_i \|W_i\|_2^2$.

2.1 General Parameters choice

As general parameters for the neural network, we chose:

- the number of hidden layers $\in [1, 5]$;
- the number of units for each layer $\in [1, 50]$;
- the regularization parameter $\lambda \in (0, 0.2)$;

We restricted the possible number of layers and the number of units mainly to limit the complexity of our search of the optimal parameters. The aim of this is also avoid over-fitting the Dataset, so a wider range would only increment this risk.
For the regularization parameter we chose a small range. Choosing values grater than 0.2 would lead to the risk of arming the stability of the learning. To choose this sets of parameters and the ones specific of each learning algorithm, we apply different techniques.

2.2 Sub-Gradient of the Loss Function

To apply the two algorithms that we chose, we need to compute the Sub-Gradient of our *Objective Function*. We decide to split the Loss function in two components:

$$E(W) = \Delta(W) + \lambda \cdot \phi(W) \tag{3}$$

In the equation 3 we refer to $\phi(W)$ as the **Regularization term**, and with $\Delta(W)$ as the **Mean Square Error term** produced by our Neural Network on the labeled dataset. We now see in more details how we computed the Sub-Gradients of each component.

2.2.1 Sub-Gradient of the Mean Square Error

The sub-gradient of the Mean Square Error is computed applying the **Back-Propagation Algorithm** [DER86]. For each pattern $p = (x_p, t_p)$ considered, we compute the gradient using this two equations.

$$\frac{\partial \Delta_p}{\partial W_i} = -\delta_p^i * R_p^{i-1} \tag{4}$$

$$\delta_p^i = \begin{cases} (t_p - R_p^h) \circ f'(W_h * R_p^{h-1}) & i = h \\ W_{i+1}^T * \delta_p^{i+1} \circ f'(W_i * R_p^{i-1}) & i \neq h \end{cases} \tag{5}$$

We used the "$\circ$" sign to refer to the Hadamard Product, meaning an element-wise multiplication of the matrices.
The main idea is to compute the δ^i associated with each Layer using the equation 5, starting from the last one ($i = h$). Then compute the sub-gradient of each matrix by multiplying it with the output of the previous layer (equation 4).
Since we use a Stochastic Gradient estimation, we then make an average over the set of pattern considered.

2.2.2 Sub-Gradient of the Regularization

In our experiments, we will use the L1 and L2 Regularization techniques.

$$\phi_1(W) = \sum_i \sum_{w \in W_i} |w| \qquad \frac{\partial \phi_1(W)}{\partial w_{ji}} = \text{sign}(w_{ji}) \tag{6}$$

$$\phi_2(W) = \sum_i \sum_{w \in W_i} w^2 \qquad \frac{\partial \phi_2(W)}{\partial w_{ji}} = 2w_{ji} \tag{7}$$

The important difference between these two functions is that the equation 6 is not differentiable but it has the advantage of increasing the sparsity of the Weight Matrices.
To work around the non differentiable point in 0 of ϕ_1, we used $\partial\phi_1(0) = 0$.

2.3 Activation functions

As activation functions we used the Hyperbolic Tangent, the Linear Function and for some experiments the ReLU.
The architecture that we implemented uses either the ReLU or the hyperbolic tangent for the hidden layers and the linear function for the last layer, the output one. For each function we have computed the sub-gradient.

2.3.1 Hyperbolic Tangent

The function is both continue and differentiable.

$$f(x) = \frac{2}{1 + e^{-2x}} - 1 \qquad \partial f(x) = 1 - f(x)^2 \tag{8}$$

The main problem with this activation function is when we are dealing with many layers in a NN. In the Back-propagation Algorithm we are multiplying each layer for the $\partial f(x)$. Since $\partial f(x) \in [0, 1]$, this creates a *vanishing* of the gradient trough the layers making harder for a NN to learn.

2.3.2 ReLU

This function is continue, defined Piece-Wise and not differentiable in 0. However we took $\partial f(0) = 0$, given that this value is in his sub-gradient set.

$$f(x) = \max(x, 0) \qquad \partial f(x) = \max(\text{sign}(x), 0) \tag{9}$$

We use this function in particular to make some test with *Deep Neural Network* because it solves the *Gradient Vanishing* issue that arises with the Hyperbolic Tangent.

2.3.3 Linear Function

For completeness we list also the linear activation function. This function is used to make the NN able to represent each possible output, since this is a Regression problem.

$$f(x) = x \qquad \partial f(x) = 1 \tag{10}$$

2.4 Non-Convexity of the Loss Function

To prove that our Neural Network Function is not *convex*, we show a plot obtained implementing a simple NN with one hidden neuron, performed on a noisy version of the $\sin(x)$ function. In the

figure 1 it can be clearly seen that the function is not *convex*.

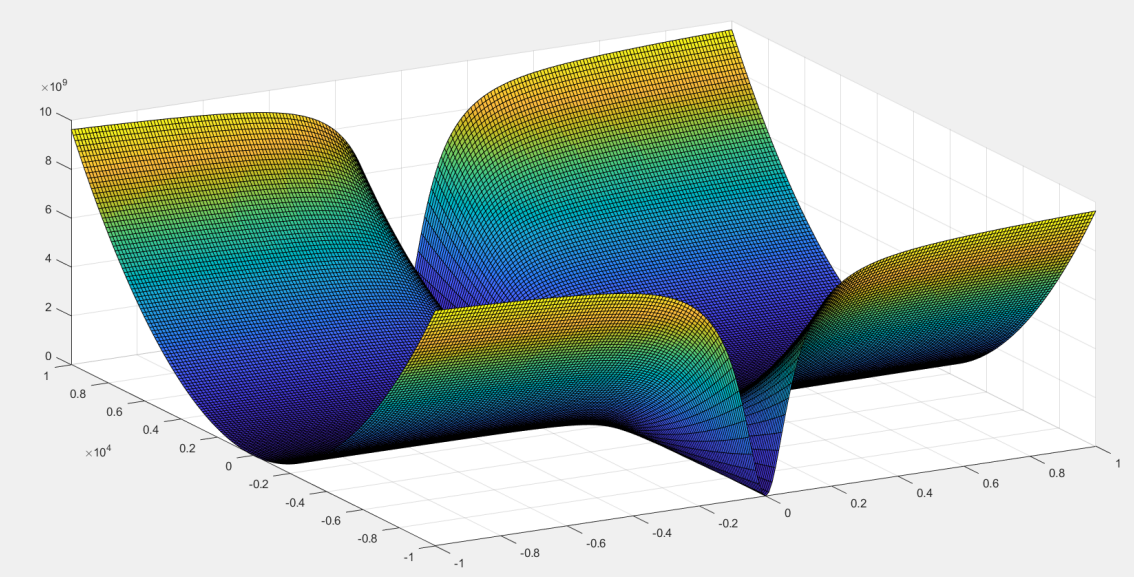

Figure 1: Plot of the MSE varying the weights of the Neural Network

2.5 Loss Function Lipschitz continuity

Here we want to prove that the Loss Function is locally Lipschitz Continuous. We will prove it using the Mean Value Theorem, so we need to prove that all the *partial derivatives* are bounded. To do so, we will split it in the two Components: MSE and Regularization.
We cannot make any assumption on the global Lipschitz Continuity, but we will prove that our Target Function is at least locally Lipschitz Continuous.

2.5.1 Locally Lipschitz continuity of Mean Square Error

We start proving the $\left\|\frac{\partial \Delta_p}{\partial W_i}\right\|$ is bounded for each i. This means that each *partial derivative* is bounded for each weight of each matrix.

$$\left\|\frac{\partial \Delta}{\partial W_i}\right\| \leq \left\|\delta^i\right\| * \left\|R^{i-1}\right\|$$

This is obtained by using the equation 4 and using the product property of the norm. Then we want to consider each term separately. We start first from the $\left\|R^{i-1}\right\|$. For this we have to consider two cases given by the two activation function that we are using. We know that, for the Hyperbolic tangent we have that $|f(x)| \leq 1$ whereas, in the case of the ReLU, we have $|f(x)| \leq |x|$.

From this, we will use the recursive expression of R^i. We start by using the ReLU architecture with the Linear function in the last layer (h-th layer). Then we will use the expression recursively to obtain the final result.

$$\left\|R^i\right\| = \left\|f_i(W_i * R^{i-1})\right\| \leq \|W_i\| * \left\|R^{i-1}\right\| \leq \left\|R^0\right\| * \prod_{i=0}^{h} \|W_i\|$$

In the case of the Hyperbolic tangent, we have a similar situation. We remind that with m_i we are referring to the dimension of the i-th layer. Here we use also the fact that the Norm of a unitary vector is the square root of the total number of elements.

$$\left\|R^i\right\| = \left\|f_i(W_i * R^{i-1})\right\| \leq \sqrt{m_{i+1}}$$

Now we need to consider the $\left\|\delta^i\right\|$ element. Here we want to apply the equation 5. We know that each $f'(x) \leq 1$ for each activation function. So we have:

$$\left\|\delta^h\right\| \leq \left\|t - R^h\right\| \qquad \left\|\delta^i\right\| \leq \|W_{i+1}\| * \left\|\delta^{i+1}\right\|$$

Applying it recursively we can obtain the following:

$$\left\|\delta^i\right\| \leq \left\|t - R^h\right\| * \prod_{j=i+1}^{h} \|W_j\| \leq (\|t\| + \left\|R^h\right\|) * \prod_{j=i+1}^{h} \|W_j\| \tag{11}$$

So in conclusion we have an upper-bound on each partial derivative, given that each component is bounded. The $\|t\| + \left\|R^h\right\|$ part is bounded because the $\|t\|$ term is the expected output (a given data), and the $\left\|R^h\right\|$ is bounded as we proved before. Then we are left only with a product of the norm of the matrices that are our input of the function. That proves that the $\left\|\delta^i\right\|$ are bounded for each i that we can choose, since we assumed to be in a compact domain.

In the end we have proven that both $\left\|\delta^i\right\|$ and $\left\|R^i\right\|$ are bounded, so also their product is bounded. This implies that each partial derivative is bounded in each compact set of the Weights Space.

2.5.2 Locally Lipschitz Continuity of Regularization

Here the regularization functions are defined as norms of the weights. Then, the gradient is bounded in both cases.

$$\left\|\frac{\partial \phi_1(W)}{\partial w_{ji}}\right\| \leq 1 \qquad \left\|\frac{\partial \phi_2(W)}{\partial w_{ji}}\right\| \leq 2 * |w_{ji}| \tag{12}$$

This gives us, by definition, that if we are in a compact domain, we are bounded. Then this proves that also the regularization part of our $\partial E(W)$ function is locally bounded, so $E(W)$ is locally Lipschitz Continuous.

2.5.3 Non globally Lipschitz Gradient

In this paragraph we want to show that our function has not a globally Lipschitz Gradient. To do this, we studied the case of a simple 1-layered neural network with one Hidden neuron, one input and one output. In this case we have that our function can be written has:

$$NN(x) = w_o * f(w_h * x)$$
$$E(w_o, w_h) = \|y - NN(x)\|_2^2$$

Here we omit the regularization term, since it has a globally Lipschitz gradient.
For this simple network we have computed the gradients and some second partial derivatives. If one partial derivative can't be bounded with a constant, it means that the Hessian matrix has not a bounded norm, and hence our gradient is not Lipschitz.
We now report the result given by the derivations:

$$\frac{\partial E(w_o, w_h)}{\partial w_o \partial w_h} = -x * f'(w_h * x) * [y - 2 * NN(x)]$$

From this we take the case where we are using a Relu activation function. We also assume, since in this cases they are considerate constants, that the x values are all positives. We can make this assumption because in the opposite case we have the same result, only symmetric for the w_h axis. Then the function become:

$$\frac{\partial E(w_o, w_h)}{\partial w_o \partial w_h} = -x^2 * max\{0, sign(w_h)\} * [\frac{y}{x} - 2 * w_o * max\{0, w_h\}]$$

It can be easily seen that if $w_h \leq 0$ then all the partial derivative is always 0, since the $max\{0, sign(w_h)\} = 0$. On the other side, if we take $w_h > 0$ we find that the resulting function is:

$$\frac{\partial E(w_o, w_h)}{\partial w_o \partial w_h} = -x * (y - 2 w_o w_h x) \tag{13}$$

So in this case we can see that the partial derivative cannot be bounded to any constant, since it depends linearly on the weights. This proves that the norm of the Hessian of our simplified Neural Network cannot be bounded, so the gradient is not globally Lipschitz in our specific case. This is a counter example on the globally Lipschitzianity of our general function.

2.6 Standard momentum descent approach

For our first algorithm we used a standard momentum descent approach [IS13]. The general update will be the step s_i. We will refer with W^k meaning the whole set of weights of our Neural Network at the k-th iteration.

$$W^{i+1} = W^i + s_i \tag{14}$$

To compute the step we will compute the sub-gradient in the W^i point and add the momentum (a portion of the last step).

$$s_i = \eta \cdot g_i + \alpha \cdot s_{i-1} \qquad \text{with } g_i \in -\partial E(W^i) \tag{15}$$

Note that here we are using the "+" sign as a sum between two set of matrices element-wise.

Algorithm 1 Standard Momentum Descent

procedure SGD(η, α, λ, I_{max})
  $W^0 \leftarrow Random\_Init()$
  $g_0 \leftarrow \partial E(W^0)$
  $i \leftarrow 0$
  while $i < I_{max}$ **do**
    $g_i \leftarrow \partial E(W^i)$
    $\Delta_i \leftarrow \eta * g_i + \alpha * \Delta_{i-1}$
    $W^{i+1} \leftarrow W^{i+1} + \Delta_i$
    $i \leftarrow i + 1$

2.6.1 Stopping criteria

For the SGD Algorithm used a fixed amount of epochs as stopping criteria. We tried to add also $\|\partial E(W)\| < \epsilon$, but we decided not to use it for two reasons. First, we are using a mini-batch estimation, so we cannot relay on a single gradient value to stop. Second, after some testing, we noticed that the norm was never going below 10^{-2}. This fact convinced us that was not possible to use an average of the past estimated sub-gradients.

2.6.2 Parameters choice

The parameters of the Standard Momentum Descent that we need to choose, in addition to those associated with the Neural Network, are:

- the learning rate $\eta \in (0, 1)$;
- the momentum rate $\alpha \in (0, 1)$;
- The I_{max} number of epochs that was fixed to 500 after some tries with other values.

For both the two parameters, we have chosen the largest possible domain, given that the algorithm may converges.

2.7 Deflected sub-gradient method (Polyak)

We implemented a deflected sub-gradient method using the Polyak step size[AF17].
Deflection:

$$d_i = \gamma_i * g_i + (1 - \gamma_i) * d_{i-1} \tag{16}$$

with $g_i \in -\partial E(W^i)$
We choose $\gamma_i = argmin \left\{ \|\gamma * g_i + (1 - \gamma) * d_{i-1}\|_2^2 : \gamma \in [0, 1] \right\}$
We can calculate this γ_i by solving the minimization problem $F(\gamma) = \|\gamma * g_i + (1 - \gamma) * d_{i-1}\|_2^2$. Since this is a quadratic problem and the norm is always positive, we can conclude that $F(\gamma)$ is convex.
Now we can compute $\nabla F(\gamma)$ and send it to 0, to find the optimal γ.
$\nabla F(\gamma) = 0 \Rightarrow 2 * (\gamma * g_i + (1 - \gamma) * d_{i-1})^T \cdot (g_i - d_{i-1}) = 0$
$\Rightarrow \gamma * (g_i - d_{i-1})^T * (g_i - d_{i-1}) = -(d_{i-1})^T \cdot (g_i - d_{i-1}) \Rightarrow$

$$\gamma_i = \frac{(d_{i-1})^T \cdot (d_{i-1} - g_i)}{\|g_i - d_{i-1}\|_2^2} \quad if \quad g_i \neq d_{i-1} \tag{17}$$

This gives us a solution for our problem, but we are not taking into account in any way the constraint $\gamma \in [0, 1]$. To achieve this, we need to "project" our solution. We will do it by taking the closest result to the actual solution.

$$d_i = \begin{cases} d_{i-1} & \gamma \leq 0 \\ g_i & \gamma \geq 1 \\ \gamma_i * g_i + (1 - \gamma_i) * d_{i-1} & \gamma \in (0, 1) \end{cases} \tag{18}$$

This gives us a way to compute the d_i direction. For the α_i we use the Step Rule by Polyak. We will also mention the Safe Rule, related to the choice of β_i.

$$\alpha_i = \frac{\beta_i * (E(W^i) - E^*)}{\|d_i\|^2} \quad \wedge \quad \beta_i \leq \gamma_i \leq 1 \tag{19}$$

Here the free parameters are E^*, the optimal value of the function $E(W)$, and β_i. We also need to fix $d_1 = g_1$, otherwise the algorithm will never move.

2.7.1 Target Level Estimation

To estimate the E^* we implemented a Target Level strategy. This mean that we introduced some more parameters but it allowed us to use a better estimation rather then fixing a lower bound as we have done at the beginning.

The strategy that we used is a *Non-vanishing Threshold* [DF09]. It had the downside that we introduced a new δ parameter, but it improved the precision of the results overall. Also we didn't want to introduce too many parameters in our algorithm and using a Vanishing Threshold would have meant introducing at least two more parameters to handle it.
In general the estimation at each step became:

$$E_i^* = E_{best}(i) - \delta \tag{20}$$

where $E_{best}(i)$ is the best value of the function found until the i-th iteration.

2.7.2 Stopping criteria

Since we introduced a target level technique, we could not rely on the $|f_i - E^*| < \epsilon$ criterion. The other two options that were left were checking the norm of the sub-gradient and setting up a cap on the maximum number of iterations performed.
We decided not to use the norm of the sub-gradient as criterion for two reasons. First, we experimentally observed that the norm usually was never going under a value of 10^{-2} (as explained in sec 2.6.1). Second, we thought that since we were using a mini-batch estimation of the Loss in each iteration, even if we encountered a Zero Gradient, we could not fully trust the value that we had found.
Given this considerations, we used a maximum number of iterations as a stopping condition criteria. This could have been another free parameter of our algorithm, but we decide to fix it at a value of 500 after testing some different values. This had the effect to be large enough to give the algorithm the time to reach a good enough solution, but at the same time it is not too big to arm the speed of the training.
The speed of the algorithm was a key point of our experiment, since we where using a Random and Genetic search, that needed to perform 1000 different cross-validation training to find the best parameters for the algorithm.

2.7.3 Parameters choice

In the case of the *Polyak* Algorithm, in addition to the general parameters of the Neural Network, we need to choose :

- The parameter $\beta \in (0, 1)$, that is a sort of Learning Rate;
- The fixed Threshold $\delta \in (0, 2)$;
- The I_{max} number of epochs that was fixed to 500;

For the parameter β, we used a domain similar to the *Learning Rate* for the previous method, because it has a similar role. The fixed threshold δ is used to implement the target level estimation.

Algorithm 2 Poliak

procedure POLIAK(β, δ, I_{max})
  $W^0 \leftarrow Random\_Init()$,
  $f_0 \leftarrow E(W^0)$, $g_0 \leftarrow \partial E(W^0)$, $d_{-1} \leftarrow g_0$
  $f_{best} \leftarrow f_0$
  $i \leftarrow 0$
  while $i < I_{max}$ **do**
    if $d_{i-1} = g_i$ **then** ▷ compute Deflection and Step
      $d_i \leftarrow g_i$
    else
      $\gamma_i \leftarrow (d_{i-1})^T \cdot (d_{i-1} - g_i)/\|g_i - d_{i-1}\|_2^2$
      if $\gamma_i < 0$ **then**
        $d_i \leftarrow d_{i-1}$
      else if $\gamma_i > 1$ **then**
        $d_i \leftarrow g_i$
      else
        $d_i \leftarrow \gamma_i * g_i + (1 - \gamma_i) * d_{i-1}$
    $f_{best} \leftarrow \min(f_{best}, f_i)$
    $\alpha_i \leftarrow \beta_i * (f_i - f_{best} + \delta)/\|d_i\|^2$
    $W^{i+1} \leftarrow W^i + \alpha_i d_i$ ▷ Update of the Weights
    $i \leftarrow i + 1$
    $f_i \leftarrow E(W^i)$
    $g_i \leftarrow \partial E(W^i)$
  return f_i

2.8 Mini-Batch Estimations

As we said earlier, in each iteration of our algorithm we used a *mini-batch* approach to estimate the $\partial E(W^i)$ and the $E(W^i)$ values. To find the best size of our mini-batch, we performed some experiments that have been described in the 4.3 section.

2.8.1 Decaying Learning Rate

The use of a mini batch approach has also some consequences on the stability of the learning curve of the algorithm, in particular when we are near a minimum point. To solve this issue we used a *Decaying Learning Rate* [GBC16]. Then, we used a Learning Rate η_i that decays through the iterations of our algorithm. We first define $\alpha_i = \frac{i}{\tau}$, then the general step in the i-th iteration become:

$$\eta_i = \begin{cases} \eta_{max} * \alpha_i + \eta_{min} * (1 - \alpha_i) & i \leq \tau \\ \eta_{min} & i > \tau \end{cases} \tag{21}$$

This is a linear decay of the learning rate. To use this approach we needed to introduce even more free-parameters so we decide to fix some of them, trusting the literature on the subject [GBC16].
This mean that we fixed the $\eta_{min} = 0.01 * \eta_{max}$.
We decided also to fix $\tau = 250$, that is half of our total iterations.
In this way there was only left to fix the η_{max} that is the initial learning rate so we did not increment the number of free-parameters to tune.

2.8.2 Incremental sub-gradient

Searching for the theory of convergence behind the use of a mini-batch approach, we have found some proves of his convergence in the case of the sub-gradient. Some of this theory, however, was using a small addiction to the algorithm. Every some epochs, they where doing a full batch iteration. We decided to test also this approach, to find out if it could improve our results. The results are shown in the section 4.4.

3 Theoretical convergence results

The theoretical convergence results according to the scientific community are summarized below.

We want to recall that the goal of our algorithm is to find an ϵ-optimal point. Since we know that our function is bounded below, we can say that even if an optimal point does not exists, meaning that our function decreases infinitely, we can find a finite point that is ϵ-optimal.

From now on, we refer to x_* as one arbitrary ϵ-optimal point. We also assume that $f(x_j) - f(x_*) \leq \epsilon$ is a stopping condition of our algorithm even if we do not use it since we do not really know the value $f(x_*)$.

Now we split this theoretical convergence summary in two sections. In the first one we assume the convexity of the function, whereas in the other we report some results of non convex optimization, where the function is assumed to be globally Lipschitz, and L-Smooth.

In our function, we do not have neither of this properties, as we discussed before. In this situation we were not able to find proof that ensures the convergence of our algorithm. In the scientific community there are convergence proofs that do not use any of this mentioned assumptions, but for other types of optimization algorithms like the Quasi-Secant algorithm that we will see ([BJK+13]).

3.1 Convex case

If we assume that our function is convex, we can prove that our algorithm lives in a bounded set under the condition that for each step we have $\alpha_i \leq \frac{2*(f(x_i)-f_*)}{G}$. This condition ensures that the distance from x_* decreases on each step, so if we take $R \geq \|x_0 - x_*\|$, then each point x_j will be inside of the closed ball $X = B(x_0, R)$, since $\|x_{j+1} - x_*\| \leq \|x_j - x_*\|$. This set would be bounded and compact and thus it would exist a constant $G \geq g_i$ where $g_i \in \partial f(x_i)$, $x_i \in X$.

3.1.1 Assumptions

This arguments give us the assumptions that are used in the proves below:

- The norm of the subgradient is bounded by a constant that is called $G_X \geq \|g_i\|$;
- There exist a constant R such that $R \geq \|x_0 - x_*\|$

3.1.2 Sub-gradient method

A sub-gradient method use the following iteration:

$$x^{k+1} = x^k - \alpha_k g^k$$

where g^k is any sub-gradient of f at x^k and $\alpha_k > 0$ is the step size at the k^{th} iteration.

It has been proven [SB08] that:

$$f^k_{best} - f* \leq \frac{R^2 + G^2 * \sum_{i=j}^{k} \alpha_i^2}{2 * \sum_{i=j}^{k} \alpha_i} \tag{22}$$

Where $\left\|g^k\right\|_2 \leq G$ for all k and $R \geq \left\|x^j - x*\right\|_2$.
In our algorithm we are using a α_i that decays through the iterations. Since it decays only for the first few iterations, we approximate it as fixed. This is useful to see the convergence of our algorithm. With a constant step size, we have:

$$f^k_{best} - f* \leq \frac{R^2 + G^2 * \alpha^2 * k}{2 * \alpha * k} \qquad \lim_{k \to \inf} f^k_{best} - f^* \leq \frac{G^2 \alpha}{2}$$

So, this algorithm converges in within a $G^2\alpha/2$ range to the optimal value.

3.1.3 Polyak step size

It has been proven that, in the case of a Polyak step size, the step $\alpha_k = \frac{(f^k - f^*)}{\|g^k\|^2}$ [SB08]. Using this we have:

$$\sum_{i=j}^{k} (f(x^i) - f^*)^2 \leq R^2 G^2 \tag{23}$$

From this, we can say that $(f(x^i) - f^*)^2 \to 0$. This is true because it is a necessary condition for our series to converge. Since $R^2G^2 < \infty$, this means that the series has to converge. So we can say that to be ϵ-suboptimal, the algorithm needs $k = (RG/\epsilon)^2$ steps. This has also been proven to be optimal.

3.1.4 Incremental subgradient

To ensure the convergence of the Incremental Sub-gradient approach, we need that our function to minimize is composed of a sum of sub-functions.

$$f(x) = \sum_i f_i(x) \qquad \text{with } x \in X$$

Also we need that each function f_i is convex and that X is a non-empty, closed and convex subset of $\mathbb{R}^n$. For our case, since we are using a MSE estimation, we notice that we are in the same case:

$$E(W) = \frac{1}{N} \sum_p E_p(W) \tag{24}$$

Here N is the total amount of pattern that we are considering, and we can take each $f_i = \frac{1}{N} E_p(W)$. Unfortunately we are not in the case of f_i convex. Moreover, we cannot make any assumption on our domain. We cannot say that it is a closed set, but we know that it is not-empty. If this conditions hold, we could ensure the convergence of this algorithm as demonstrated in the paper [NB99].

3.1.5 Momentum Convergence

It has been proven [IS13] that the momentum can considerably accelerate the convergence of the SGD algorithm to a local minimum requiring R times fewer iterations to reach the same level of accuracy where R is the condition number of the curvature at the minimum and the momentum is set to $\frac{\sqrt{R}-1}{\sqrt{R}+1}$. In general the momentum is useful mostly to pass through canyon-shapes regions, where the function keep jumping between two or more set of close points.

3.1.6 Convergence in the differentiable case

If we are using the L2 regularization and the Hyperbolic Tangent as activation function, our $E(w) \in C^2$, meaning that our target function is two times differentiable. So we know that the convergence in this case is linear with a Rate $R = (\frac{\lambda^1-\lambda^n}{\lambda^1+\lambda^n})^2$. This has been proven in the book [NW06].

3.2 Non-convex Case

In the case of a non-convex function, like ours, we need to assume the Lipschitzianity of the function and the gradient. Under this conditions, it as been proven the asymptotic convergence of the Stochastic Momentum Sub-gradient algorithms in the paper [TY16].
In this paper the possible algorithms described are unified as one, called "Unified Momentum":

$$\begin{cases} y_{k+1} = x_k + \alpha G(x_k) \\ y^s_{k+1} = x_k + s\alpha G(x_k) \\ x_{k+1} = y_{k+1} + \beta(y^s_{k+1} - y^s_k) \end{cases} \tag{25}$$

where $G(x) = G(x;\xi)$ is a stochastic subgradient of f(x) at x depending on a random variable ξ such that $E[G(x,\xi)] = \nabla f(x)$.

With specific values of the parameter s, we obtain different algorithms. With $s = 1$ we have the Nesterov Accelerated Gradient, with $s = 0$ we have the usual Heavy Ball and with $s = \frac{1}{1-\beta}$ we have the Stochastic Sub-gradient Descent Algorithm. In this paper then, they prove,. both for the convex and non-convex cases, the convergence of the Unified Algorithm.

In this section we want to focus on the non-convex case, where supposing the global Lipschitzianity of both the function and the gradient, they prove that setting $\alpha = \min(\frac{1-\beta}{2L}, \frac{C}{\sqrt{t+1}})$, it exists an upper bound that depends only on the selection of the hyper-parameters and on the number of iterations. It can be summarized with the following formula where $K_{\alpha,\beta,s}$ is a constant that depends on some fixed hyper-parameters and t is the number of iterations of the algorithm.:

$$\min_j \|\nabla f(x_j)\|^2 \leq \frac{K_{\alpha,\beta,s}}{\sqrt{t+1}} \tag{26}$$

This bound ensures that the algorithm converges asymptotically under the above conditions, given that we make a proper choice of our hyper-parameters.

In the case of our *Deflected Sub-gradient*, this proof of convergence does not hold. This is caused by the fact that we are assuming that both β and s are constants, whereas in the *deflection* we are dynamically combining them with γ_k.

We have the same problem with the *Polyak step size* rule, since in our algorithm we are taking it as depending on the current value of the function as we can see from equation 19. In the proof above instead, we are taking α_k that vanishes through the iterations.

3.2.1 Quasi-secant Algorithm

For the Deflected sub-gradient, we have found a proof of convergence in non-convex and non differentiable case using the Quasi-Secant Method. This method is very similar to the one used by us, with the only difference that instead of using sub-gradients, it uses the quasi-secants, that are a generalization. We will report here the definition of quasi-secant.

Definition (Quasi-secant): A vector $v \in \mathbb{R}^n$ is called a Quasi-secant of a locally Lipschitz function $f : \mathbb{R}^n \to \mathbb{R}$ in a point x along the direction $d \in \mathbb{R}^n$, $\|d\| = 1$ with length $h > 0$ if and only if:

$$f(x + hd) - f(x) \leq h\langle v, d\rangle$$

And v is a sub-gradient of the function f in a neighborhood of x.

In the paper [BJK+13], it is proven the convergence of this algorithm. The major difference between this and ours is the usage of two different stopping condition, rather than fixing an amount of steps. This conditions puts the focus on the norm of the direction, that is a condition that can ensure the ϵ-optimality.

Under this stopping conditions, the algorithm convergences in a finite number of steps m, given by the inequality:

$$m \leq \frac{2 * log_2(\epsilon/C)}{log_2(C_1)} + 1 \tag{27}$$

Where $C = \max\{\|v\| : v \text{ is a quasi-secant of } x\}$, and C_1 is a constant derived by C. In the paper is also reported a variant that actually uses the sub-gradients, given that the function is locally Lipschitz and directionally differentiable. Since we are in the non-differentiable case, it is not exactly what we have, but it is close enough. Moreover, in this proof they used a fixed step size whereas we applied a Polyak step-size Rule, but we can use an upper-bound to have a similar situation. This shows that in general the Deflected sub-gradient approach can be convergent also for the non-convex case.

4 Neural Network experiments

In this section we show some experiments that we did starting from the choice of the model with three search methods. Then we explain the choices for the size of the mini-batch and the batch rate. After this, we try the two different activation functions that we showed above. Finally, we made a comparison between the explained algorithms, SGD and the Deflected subgradient, and the *Neural Network Toolbox* of the *Matlab* environment.

4.1 Model Selection choice

We have implemented three different algorithms to tune the free-parameters of our algorithm. Fro this experiments we used a full batch approach. The algorithms are:

- **Grid Search:** we used a bisecting approach, where we fix the limits and keep splitting trying to find the best result at each step and moving around it. In this case we are performing 160 Cross-Validation Training with 3 folds.
- **Random Search:** we used a random approach, where we take at random each parameter and search for the parameters that give us the best result on the Validation set. In this case we are performing 1000 Cross-Validation Training with 3 folds.
- **Genetic Search:** we implemented a Genetic algorithm to search for the best parameters. Here each individual is composed of the set of parameters that we are considering and as objective function we take the final result of our training. In this case we are performing 1000 Cross-Validation Training with 3 folds.

	Parameters	**Tr. Error**	**Val. error**	**Time**
Grid Search	{11, 0.25005, 0.003125, 0.3125}	1.204	1.256	88 s
Random Search	{[17;27], 0.041629, 0.0020932, 0.86181}	1.077	1.229	1069 s
Genetic Search	{[30;40], 0.022742, 0.0014358, 0.92035}	1.017	1.163	1208 s

Table 1: SGD model selection

	Parameters	**Tr. Error**	**Val. error**	**Time**
Grid Search	{50, 0.5, 0.005625, 1.375}	1.188	1.318	346 s
Random Search	{[41;33], 0.14077, 0.0006998, 1.978}	0.904	1.2547	2188 s
Genetic Search	{[8;47], 0.109, 5.4037e-05, 1.4731}	1.208	1.238	2821 s

Table 2: Deflected subgradient model selection

As we can see, the best performing technique seems to be the Genetic Algorithm. Our choice is driven by the Validation error that each algorithm try to minimize. In both cases we can see that the Genetic Algorithm is outperforming the other two and we plan to use it for the next experiments.

One counter argument that can be posed, is that the Grid Search is performing much less training than the other two algorithms. This is caused by the fact that this algorithm is converging to a solution much faster, trough a bisecting search on the variables.

4.2 Regularization

Here we show the comparison between the usage of L1 and L2 regularization. We will keep using a full batch approach and we search for the hyper-parameters using the Genetic Algorithm.

	Tr. Error	**Val. error**	**Time**
Deflected L1	1.208	1.238	2821 s
SGD L1	1.017	1.163	1208 s
Deflected L2	0.970	1.166	3019 s
SGD L2	1.127	1.161	1467 s

Table 3: Regularization choice

Here we can see that the L2 regularization is outperforming slightly in both algorithms. However since we are mainly implementing the L1 regularization, we will use it for the Deflected Algorithm, and we use the L2 only for the SGD.

4.3 Mini-Batch size

For the choice of the Mini-Batch size we applied an experimental approach: we tried different dimensions and we chose the one that performed the best. According to the table 4.3, we can see that the best size, based on execution time and errors, is the one that use 10% of the data set for the SGD and 50% for the Deflected Subgradient. We will use this choices for the next experiments that we will perform.

	SGD L2			**Deflected Subgradient L1**		
Mini-Batch Size	**Train Error**	**Val. Error**	**Time**	**Train Error**	**Val. Error**	**Time**
100%	1.017	1.163	1208 s	1.208	1.238	2821 s
50%	1.171	1.213	584 s	1.201	1.279	1517 s
20%	1.054	1.179	548 s	1.243	1.333	1429 s
10%	1.101	1.172	474 s	1.2928	1.3113	1121 s
5%	1.054	1.212	257 s	1.2958	1.3546	689 s

Table 4: Mini-batch size choice

4.4 Batch Rate

Here we test the incremental method approach. We will try to compare different full batch iteration rates and search for the best performing one. In this case we are only applying this method to the deflected sub-gradient.

Batch Rate	**Train Error**	**Val. Error**	**Time**
1/2	1.319	1.261	1962 s
1/5	1.2069	1.260	1415 s
1/10	1.1993	1.2795	1174 s
0	1.201	1.279	1517 s

Table 5: Batch Rate choice for the Deflected Sub-gradient

In table 5 we can see the results of the experiments. Batch Rate equals to 0 means that we are not performing a full batch but only mini batch iterations.
We chose that the best value for the Batch Rate is 1/5. We based our choice on a trade off between the error and the execution time.

4.5 Activation Functions

Here we aim to compare the performances of the two activation functions that we have described before: the Hyperbolic Tangent and the ReLU. We also tried the ReLU function on more than two layers to exploit the fact that this function is usually chosen for this purpose. We did not tried to search for models with more than two layer for the Hyperbolic Tangent because of the Gradient Vanish issue that we already discussed in the 2.3.1 section.

	SGD L2			**Deflected Subgradient L1**		
Activation Function	**Train Error**	**Val. Error**	**Time**	**Train Error**	**Val. Error**	**Time**
Hyperbolic Tangent	1.101	1.172	474 s	1.201	1.279	1517 s
ReLU	1.146	1.2748	699 s	1.3674	1.439	798 s
ReLU ($\gg$ 2 layers)	1.2295	1.2814	372 s	1.369	1.4348	1246 s

Table 6: Activation Function choice

As we can see in table 6, the ReLU function is not performing well in this task. This can be caused, in the case of more than two layers, on the continually increasing number of parameters given by the large amount of hidden layers. In the end, we decided to keep using the Hyperbolic Tangent that give us better results.

4.6 Final comparison

In the following table we show the results obtained with two algorithms: the SGD and Deflected subgradient. We also compare them to the Neural Network toolbox offered by the *Matlab* environment. This toolbox gives you three different learning algorithms without the possibility to use cross-validation. It tunes his parameters automatically, but it does not allow you to put more than one hidden layer. To chose the best algorithm, we tried each one and we found that the "Levenberg-Marquardt" is the best performing. We also tried different numbers of hidden units. Our final choice was 25 hidden units. Moreover, we want to mention that probably the algorithm that this tool uses do not need any parameters tuning, since it is performing only one single training.
In figure 2 and in figure 3 we can see the plots of the Learning curves and the Test curves respectively. In the first plot we can notice that the NN toolbox is converging faster and at a lower error with respect to the others but the test acts a bit worse. Comparing our two algorithms, we can see that the SGD has a steeper curvature on the first and also it has the lowest test error with respect to the off-the-shelfs algorithm.

	Tr. Error	**Val. error**	**Test error**
SGD	1.064	1.169	1.224
Deflected	1.203	1.260	1.295
NN toolbox	1.031	1.154	1.243

Table 7: Comparison Neural Network algorithms

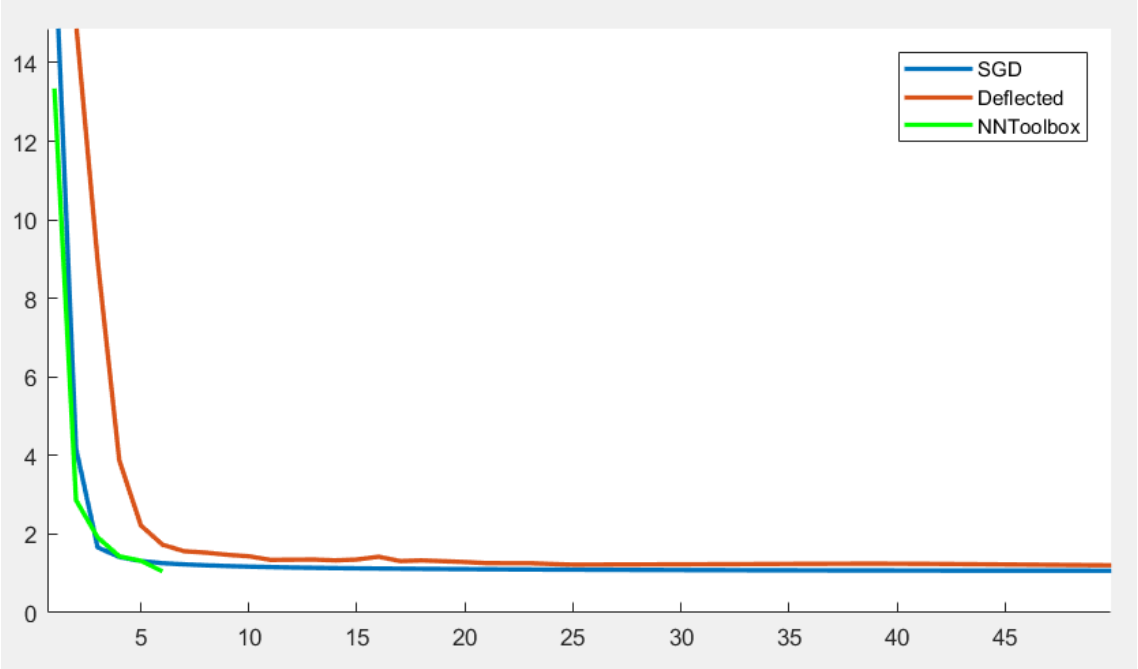

Figure 2: Comparison Graph between the Learning Curves of the Algorithms

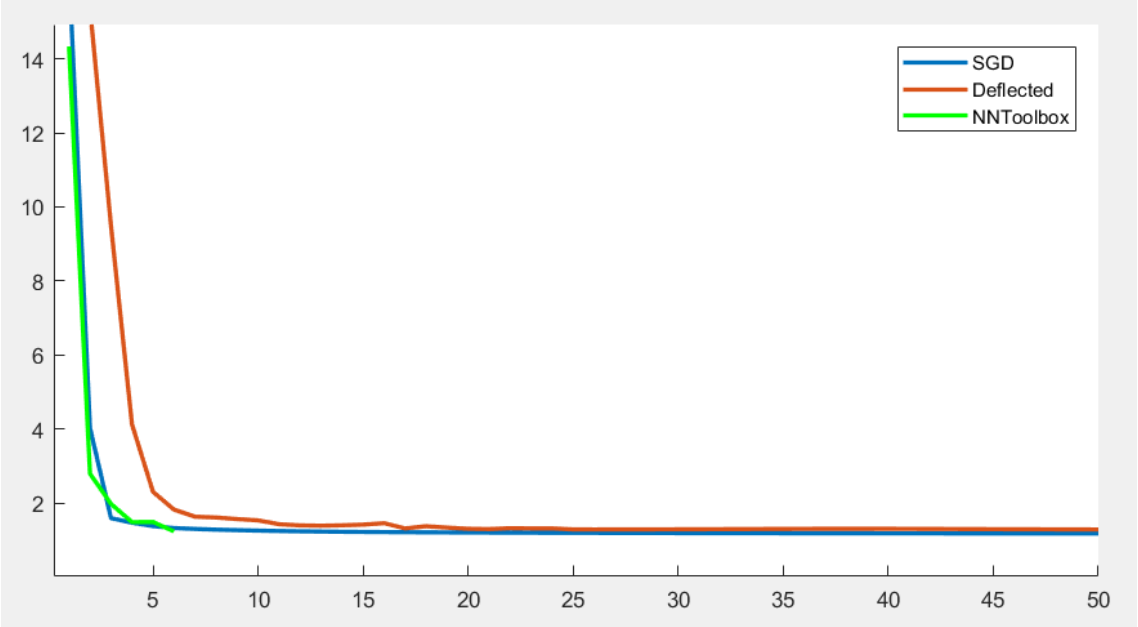

Figure 3: Comparison Graph between the Test Curves of the Algorithms

5 Standard Linear Regression

Our goal was to find an Hyper-Plane that has the minimum distance to all the data points in our dataset. This mean find a matrix $x^* \in \mathbb{R}^{10\times 2}$ such that:

$$x^* = \arg\min_x \|Ax - b\| \tag{28}$$

where $A \in \mathbb{R}^{1016\times 10}$ is our Input matrix and $b \in \mathbb{R}^{1016\times 2}$ is our Output matrix.

To solve this problem we evaluated the algorithms to solve a Least Square problem.

Here we are in the situation where x^* and b are not vectors but matrices. This doesn't change the solution of the problem, since it is equivalent to solve two separate problems each one with a column of the two matrices.

5.1 Algorithm Choice

To choose the best Algorithm for our problem, we first evaluated this following possible choices:

- Normal Equations
- QR Factorization
- SVD Factorization

5.1.1 Normal Equations

The biggest downside of this algorithm is the fact that it is not backward stable. In this case the error as a relative magnitude of $\kappa(A)^2$, where the QR and SVD have a relative error of $\kappa(A)$. Since we want to use a stabler model, we preferred to discard this algorithm.

5.1.2 SVD-QR Factorization

Since we are in the case where $m \gg n$, the performances of the SVD and QR algorithms are approximately the same.
We chose the SVD for a main reason: it allowed us to introduce a form of regularization. This is a key benefit in our opinion, since we are dealing with a noisy problem.
To see this need of regularization more in details, we have compute the singular values of our matrix:

```
Singular Values:
84.2913, 40.5825, 17.8716, 17.3410, 13.1352,
11.3026, 10.5055, 10.1639, 9.7722, 9.1663
```

The $\frac{\sigma_{max}}{\sigma_{min}} \approx 9$ is not considered a problem for the noisiness of the problem. We decided to experiment with some regularization anyway, because we are dealing with a Machine Learning task and we know that the problem can benefit from this technique.

5.2 Implementation

In this section we summarize the main ideas used to create this model. We are using a SVD factorization of our matrix $A = USV^T$ and we compute the result x with the following Ridge Regression formula:

$$x^* = \sum_{i=1}^{n} \frac{\sigma_i}{\sigma_i^2 + \alpha^2} * v_i u_i^T b \tag{29}$$

where v_i and u_i refer to the columns of V and U.

Algorithm 3 Ridge Regression

```
1: procedure RIDGE(A, b, α)
2:     [U, S, V] ← SVD(A, 0)                    ▷ Using Thin-SVD
3:     x ← 0*, i ← 0
4:     while i < n do
5:         k ← σ_i / (σ_i^2 + α^2)
6:         x ← x + k * v_i u_i^T b
7:         i ← i + 1
8:     return x
```

5.3 Parameters Choice

In this model we only needed to set the parameter α. We decided to take as minimum value of the domain of search, $\alpha = 0$ that means "no-regularization" and as maximum value we chose $max_i \sigma_i$, the maximum singular value to be regularized. So the domain become $\alpha \in [0, 84]$.

6 LMS experiments

In this section we report the tests performed and we compare the results of our algorithm against the off-the-shelf one available on the MATLAB environment. First of all, we chose the regularization parameter for our algorithm with some experiments. We performed 500 tries with 50 different

values of regularization between 1 and 20. The best one is 6.9.

In the following table 8 we can see that our algorithm is slower but the error is lower.
With our algorithm there is a small improvement of 0.1% of the total error. However, in the end, the error is too high with respect of the Neural Network one.

	Val. Error	**time**
SVD	18.3176	0.001799
Off-the-shelf	18.3373	0.000084

Table 8: Comparison LMS algorithms

7 Conclusion

In this project we implemented two algorithms for the learning process of a Neural Network: a Standard Gradient Descent approach and a Deflected subgradient one. In the second part, we also tried to implement a simple linear regression with the SVD method. During the development we explored the theory behind our methods to try to improve them. In the end, we performed some experiments to compare the results and performances that we could achieve.

To conclude we can clearly see that, for this task, the Neural Network model performs much better than the Linear Regression model. Between the Neural Network learning algorithms, we can notice that the best performing is the SGD. On the other hand, for the Linear Model we were able to enhance slightly the performances of the off-shelfs method using a regularization technique.

References

[AB12] N. Karmitsa A. Al Nuaimat N. Sultanova A.M. Bagirov, L. Jin. Subgradient method for nonconvex nonsmooth optimization. 2012.

[AF17] Enrico Gorgone Antonio Frangioni, Bernard Gendron. On the computational efficiency of subgradient methods: a case study with lagrangian bounds. 2017.

[BJK+13] A. M. Bagirov, L. Jin, N. Karmitsa, A. AlNuaimat, and N. Sultanova. Subgradient method for nonconvex nonsmooth optimization. *Journal of Optimization Theory and Applications*, 157(2):416–435, May 2013.

[DER86] J. L. McClelland D. E. Rumelhart. *Parallel Distributed Processing*, volume 1, chapter 8 Learning internal representations by error propagation, pages 322–328. MIT Press, 1986.

[DF09] Giacomo D'Antonio and Antonio Frangioni. Convergence analysis of deflected conditional subgradient methods. 2009.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[IS13] George Dahl Geoffrey Hinton Ilya Sutskever, James Martens. On the importance of initialization and momentum in deep learning. 2013.

[LT97] David Bau Lloyd Trefethen. *Numerical Linear Algebra*. SIAM, 1997.

[NB99] Angelia Nedic and Dimitri Bertsekas. Incremental subgradient methods for nondifferentiable optimization. 12, 01 1999.

[NW06] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 2006.

[SB04] Lieven Vandenberghe Stephen Boyd. *Convex Optimization*. Cambridge University Press, 2004.

[SB08] Almir Mutapcic Stephen Boyd. *Subgradient Methods, Notes for EE364b*. Stanford University, April 13, 2008. `https://see.stanford.edu/materials/lsocoee364b/02-subgrad_method_notes.pdf`.

[TY16] Zhe Li Tianbao Yang, Qihang Lin. Unified convergence analysis of stochastic momentum methods for convex and non-convex optimization. 2016.