UiO University of Oslo

Project 2

Data Analysis and Machine Learning FSY-STK4155

Maria Beatrice Cattini
Matteo D'Alessandro
Lisa Incollingo
Vanessa Ventura

November 16, 2022

Abstract

This project aims to implement, study and compare the performance of different methods for regression and classification. First we consider gradient methods, such as Gradient Descent and Stochastic Gradient Descent with different adaptive learning rate algorithms (RMSProp, Adagrad, ADAM), and considering the introduction of momentum. Subsequently, a code for a Feed-Forward Neural Network with flexible number of hidden layers and nodes is implemented and different activation functions are considered. The backpropagation algorithm is used for the training process. In our analysis, we tune the hyperparameters associated to the models above such as the learning rate, the number of hidden layers, neurons and epochs.
In the regression setting we consider generated data points from a second-degree polynomial function and compare gradient methods and the neural network based on their test mean squared error values. To validate the results, the methods were also compared to the performance of `python` libraries `scikit-learn` and `tensorflow`. We obtain very similar results, with the neural network obtaining the best overall error value of 0.0093 between the methods we implemented.
In the classification setting, we study the Wisconsin Breast Cancer data set and make predictions as to whether tumors are benign or malignant. In this case, we compare the predictive performance of the neural network with logistic regression, with the latter obtaining the best overall accuracy value of 98.2%.

Contents

1 Introduction

Conventional computer systems use an algorithmic approach to solve a problem, which limits problem solving capability to problem that we already understand and know how to solve. It would be great that computer could find a way to solve problems on data they haven't already been trained on, just like human brain. Is this possible?
Moved by this question, neurophysiologist W. McCulloch and mathematician Walter Pitts tried to make the first steps in the developement of the first models of Neural Networks in 1943 [1]. In the following decades the research on this topic continuously improved the capabilities of Neural Networks, which are now used in a plethora of different fields: forecasting and risk evaluation, image recognition, or even in medical diagnosis.

In this second project, our main aim is to implement a Feed-Forward Neural Network (FFNN) with a flexible number of layers and neurons and different activation functions and to compare its results with other methods both in regression and in classification.

Our work is structured in three main parts: in section 2 we discuss the methods that will later be applied and analysed in section 3. In the last section 4 we will present our final comparison results, including the performance comparison with neural network models from `python` libraries `scikit-learn` and `tensorflow`.

We start by solving the ordinary least squares and Ridge cases with gradient methods and studying the performance when tuning the main hyperparameters. In the regression part we also go on to consider the neural network results when applied on data generated from a second degree polynomial function with added noise:

$$f(x) = 1 + 3x + 2x^2 + \varepsilon$$

As for the classification setting, we consider both the neural network and logistic regression, working with data from the Wisconsin Breast Cancer Dataset (`https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data`) in order to classify the tumor state in malignant or benign [1] .

Our code can be consulted at the following GitHub link: `https://github.com/matteodales/FYS-STK4155_Applied_Data_Analysis_and_Machine_Learning/tree/main/project2`.

[1]See the paragraph 3.2.1

2 Methods

2.1 Gradient methods

Many machine learning problems deal with the need to find the minimum of a cost function.
Since this is not always possible to do analytically, we often have to resort to some numerical methods in order to obtain a proper approximation, while avoiding the risk of getting stuck in local minima.
The methods presented in this paragraph have been developed with this goal.

2.1.1 Gradient Descent

The *Gradient Descent* (GD) is an iterative method we can use in order to find a minimum of a function f, based on the idea that the direction of the fastest decrease of a function will be the direction of the negative gradient $-\nabla f(w)$.
The algorithm starts guessing a starting point w_0 for the minimum and iteratively updates the approximation with

$$w_{k+1} = w_k - \eta \nabla f(w_k)$$

where $\eta \in R$ is called the *learning rate*, or *step lenght*, $\nabla f(w)$ is the gradient of the function and $\mathbf{w}$ is the set of the coefficients.

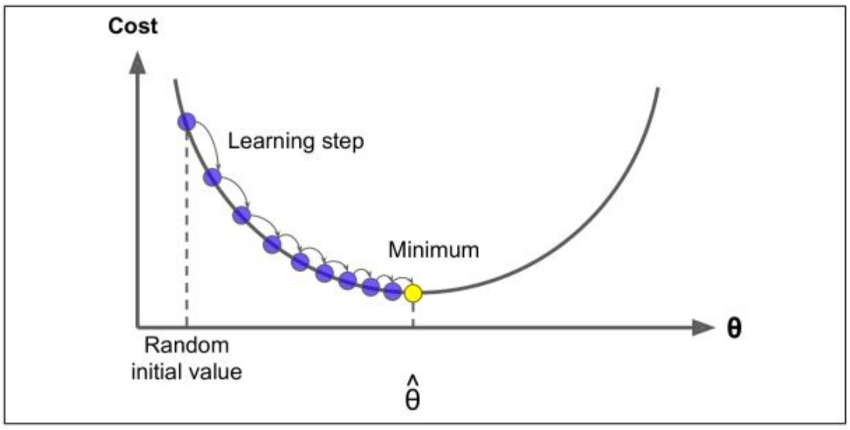

Figure 1: Gradient Descent on $f(\theta)$

The learning rate, so, is the step size at each iteration while moving toward a minimum of our function, as shown in Figure 1 [2].
Accordingly to this idea the sequence $\{w_k\}_{k=0,1,...}$ has to converge to either the global or a local minimum of the function.

[2]A. Géron, 1943 [2][p.120]

If the function is convex, the Gradient Descent will converge to the global solution. However, in the general case, the function can have plateau or irregular holes that make the convergence to the minimum really difficult, as shown in
Figure 2 [3].

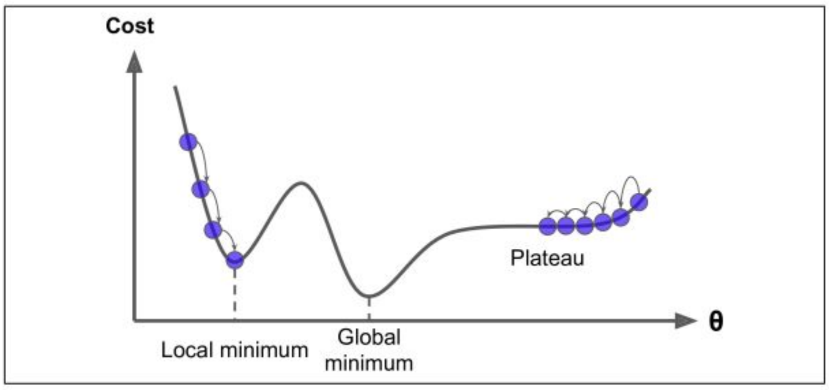

Figure 2: Pitfalls of Gradient Descent

Hence this method could have several problems:

– the choice of the starting point and the size of the learning rate are really sensitive in order to not get stuck in a local minimum;
– it's computationally expensive to calculate for complex functions.

Introducing some randomness could prevent this shortcomings.

2.1.2 Stochastic Gradient Descent

The *Stochastic Gradient Descent* is a stochastic approximation of the Gradient Descent.
The latter uses the whole training set to compute the gradients at every step, which makes the computation slower when we deal with large amounts of data.
In the Stochastic Gradient Descent, instead, we calculate the gradient only for a randomly selected datapoint.
This variation makes the algorithm faster, since the amount of data is smaller but, on the other hand it's not decaying as regularly as the Gradient Descent.
As shown in the figure 3 [4], indeed, the algorithm will end up really close to the minimum, but once it gets there it will continue to bounce around.
This means that the approximations found are not necessarily optimal.

[3]A. Géron, 1943 [2][p.122]
[4]A. Géron, 1943 [2][p.126]

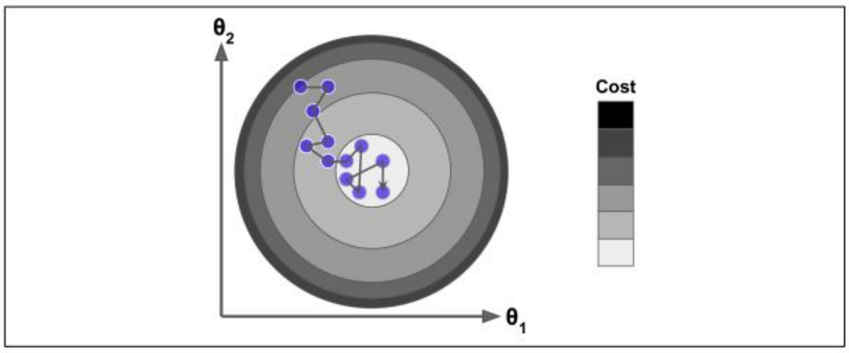

Figure 3: Stochastic Gradient Descent convergence to the minimum

2.1.3 Mini-batch Stochastic Gradient Descent

The *Mini-batch Gradient Descent* (SGD) is an improvement of the standard Gradient Descent and of the Stochastic Gradient Descent.
Instead of computing the gradients on the whole data we have or just on one instance, this algorithm calculate the gradients on small random sets of instances called **minibatches**.
If there are **n** data points and we decide that the size of each minibatch is **M**, there will be $\frac{\mathbf{n}}{\mathbf{M}}$ minibatches.
Denoting these minibatches with B_k we will have $B_k = 1, 2, .., \frac{n}{M}$.
This method progress better in the parameter space then the plain Stochastic Gradient Descent.

As we introduced the concept of batches, another notion must be introduced: the **epochs**. The number of epochs is a hyperparameter that defines the number of times the learning algorithm will run through the entire training dataset.
An epoch means that every sample in the training dataset has had the opportunity to update internal model parameters.

2.1.4 Introducing momentum

Adding a *momentum* term to the Stochastic Gradient Descent gives a memory of the direction we are moving in and allows us to overcome the oscillations of noisy gradients and overpass flat spots of the search space.
It's implemented as follows:

$$v_{k+1} = \gamma v_k + \eta_{k+1} \nabla_k f(w_k)$$

Where γ is the *momentum parameter*, with $0 \leq \gamma \leq 1$.
So the update of the coefficients has the form

$$w_{k+1} = w_k - v_k$$

with v_k that is a running average of recently calculated gradients.

2.2 Tuning methods: Adagrad, RMSprop, Adam

The learning rate η defines how fast and how precisely the algorithm learns to reach the minimum of the cost function.
Generally a large learning rate allows the model to learn faster, at the cost of arriving on a sub-optimal final result.
A smaller learning rate may allow the algorithm to reach a better result but may take significantly longer to train.

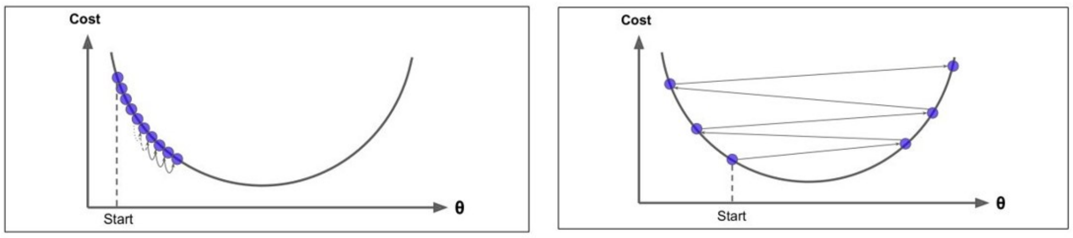

Figure 4: Comparison of convergece with different values of η

Several methods were implemented to adapt this hyperparameter in order to achieve a better outcome.

The **Adagrad method**, in addition to the momentum method, keeps track not only of the first moment of the gradient $g = \nabla f(w)$, but even of the second one denoted by $s_k = \mathbf{E}[g_t^2]$.
The algorithm rules are:

$$s_k = s_k + g_k^2$$

$$w_{k+1} = w_k - \eta_k \frac{g_t}{\sqrt{s_k + e}}$$

The second equation is almost identical to the Gradient Descent, but with a big difference: a gradient scaling is introduced. Hence, this algorithm decays the learning rate, but it does so faster for steep dimensions than for gentler slopes of our cost function. That's why this methods are called adaptive learning rates. This method performs really well with a convex function, but not that well for a non-convex function.

In this case, **RMSprop** is used. This method is really similar to Adagrad, but we introduce a new decay rate β. The update rules are the following:

$$s_k = \beta s_k + (1 - \beta) g_k^2$$

$$g = \nabla f(w)$$

So the update of the parameters is:

$$w_{k+1} = w_k - \eta_k \frac{g_k}{\sqrt{s_k + e}}$$

Where β is typically equal to 0.9 and is the decaying moving average squared partial derivative for the previous iteration.

Adam method is mixture of *momentum method* and *RMSprop*. As the first one it keeps track of the decaying average of past gradients $m_k = \mathbf{E}[g_k]$ and as the second one it keeps track of the decaying average of past squared gradients.
The rules are the following:

$$m_k = \beta_1 m_{k-1} + (1 - \beta_1) g_k$$

$$s_k = \beta_2 s_{k-1} + (1 - \beta_2) g_k^2$$

$$m_k = \frac{m_k}{1 - \beta_1^t}$$

$$s_k = \frac{m_k}{1 - \beta_2^t}$$

And the update of the parameters is:

$$w_{k+1} = w_k - \eta_k \frac{m_k}{\sqrt{s_k} + e}$$

Where β_1 and β_2 regularize the memory lifetime of the first and the second moment. Their values are fixed to be 0.9 and 0.99 respectively.

2.3 Neural Networks

Neural Networks (NNs) are computational learning systems based on a collection of connected units or nodes called neurons, which reproduce the activity of neurons in the brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons in the form of real numbers, and the output of each neuron is computed by some non-linear function of the sum of its inputs. Typically, neurons are aggregated into an input layer, an output layer and eventual in-between hidden layers.

In this project we will focus on a specific NN model called *Multilayer Perceptron* (MLP): all nodes of a layer will be connected to every node in the subsequent layer, making it a fully connected network. In addition, the information flow will only go forward through the layers.

2.3.1 Feed Forward Neural Network

The Feed Forward algorithm connects the input and output layers by advancing to following layers without generating feedback loops. Each connection in the network is associated to a weight and each node has an intrinsic bias parameter and an activation function which returns the output of the node. Hence, the output node y is produced by the activation function f:

$$y = f(\sum_{i=1}^{n} w_i x_i + b_i) = f(z)$$

where w_i are the weight, b_i are the biases and x_i are the inputs which correspond to the outputs of the nodes in the previous layer.
Figure 5 [5] allows us to visualize the process:

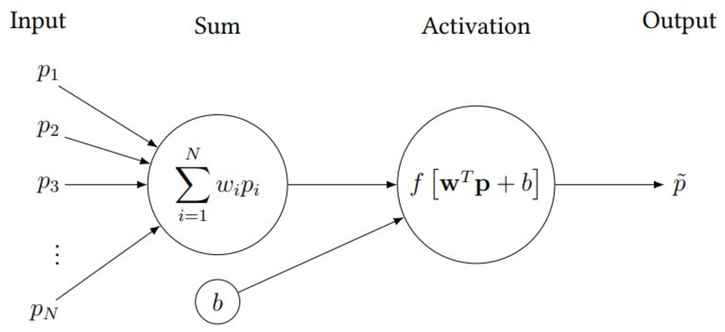

Figure 5: FFNN algorithm: the input values $p_i, i = 1, ..., N$ from the first layer multiplied by corresponding weights $w_i, i = 1, ..., N$ and summed. Then the activation function f is applied to $\mathbf{w^T p} + b$, in which b is a vector $N \times 1$. The output goes on to become input for neurons in the next layer.

[5] M. Ledum, 2017 [3]

If we consider an MLP with l hidden layers, the mathematical expression of FFNN is:

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^{l+1} f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^l\left(\dots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\dots\right) + b_k^l\right) + b_1^{l+1}\right]$$

where N_l is the number of nodes in layer l, f^l is the corresponding activation function with weights and biases. The inputs of every layer are the outputs of the previous one. Thus, the high number of coefficients associated to a FFNN, even with a small number of neurons, make this kind of model very adaptable but also prone to over fitting when dealing with smaller amounts of data.

2.3.2 Activation function

The choice of the activation function is very important to a given NN and, for the hidden layers,it must be a non-constant, bounded, monotonically-increasing and continuous function. If these restrictions are respected, the following theorem holds.

Theorem (Universal Approximation Theorem). *A feed-forward neural network with just a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of real functions arbitrarily closely.*

Some frequently used activation functions include:

1. Sigmoid:
$$f(z) = \frac{1}{1+e^{-z}}$$
Since $f(z) \in (0,1)$, it can be used as an output function in classification problems when we want to predict the probability of belonging to a class as output of the network.

2. Hyperbolic tangent: $f(z) = \tanh(z)$, $f(z) \in (-1,1)$

3. Rectified Linear Unit function (ReLU): $f(z)=\begin{cases} 0 \text{ if } z < 0 \\ z \text{ if } z > 0 \end{cases}$

4. Leaky ReLU: $f(z) = \begin{cases} \alpha z \text{ if } z < 0 \\ z \text{ if } z > 0 \end{cases}$, $\alpha = 0.01$

The use of activation functions in the ReLU family is linked to the vanishing gradients problem: the other activation functions we considered have derivatives between 0 and 1, and since the backpropagation algorithm used for learning requires the multiplication of this derivatives over the layers, the gradients used to update the weights can progressively get smaller and smaller, slowing down the learning process. This problem can be addressed by using the ReLU functions since their derivative is equal to exactly 1 when the input is positive, which mitigates the shrinking of the gradients.

2.3.3 Back propagation algorithm

After the FFNN process, the output layer contains the predictions of the model, but usually these predictions are not satisfactory because bias and weights are initialized randomly. To improve the performance of NN we will compare its predictions with the real targets of our training data set by applying a specific cost function and use these results to update the weights and biases.

The method used for updating the parameters is called back propagation algorithm.

Firstly we define the outputs and activated outputs respectively as:

$$z_j^l = \sum_{i=1}^{M_{l-1}} w_{ij}^l a_i^{l-1} + b_j^l,$$

$$a_j^l = f(z_j^l).$$

The computations occur from the output layer to the input layer in a backward process. Our aim is to minimize the cost function by calculating its gradients. The way we update the weights and biases each epoch is given by:

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial C}{\partial W^{(l)}}$$

$$b^{(l)} \leftarrow b^{(l)} - \eta \frac{\partial C}{\partial b^{(l)}}.$$

Where η is the learning rate, that lets us know how much weight and bias should be adjusted at each update. By using the chain rule, we can express the cost function gradients as:

$$\frac{\partial C}{\partial (w_{ij}^L)} = \frac{\partial C}{\partial (a_i^L)} \frac{\partial a_i^L}{\partial (z_j^L)} \frac{\partial z_j^L}{\partial (w_{ij}^L)}$$

Where, by the definition of the activated outputs:

$$\frac{\partial z_j^l}{\partial w_{ij}^l} = a_i^{l-1},$$

Finally, we define the output error for each node as:

$$\delta_j^L = f'(z_j^L)\frac{\partial C}{\partial(a_j^L)}$$

where f' is the derivative of the activation function utilized.
We can now update the weights and the biases using gradient descent for each layer in the following way:

$$w_{jk}^L \leftarrow w_{jk}^L - \eta \delta_j^L a_k^{L-1}$$
$$b_j^L \leftarrow b_j^L - \eta \frac{\partial C}{\partial b_j^L} = b_j^L - \eta \delta_j^L$$

To procede in the algorithm, we can then compute the back propagated error for each $l = L-1, L-2, \dots, 2$ with L the number of layers, as:

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l)$$

By repeating the Feed Forward and back-propagation algorithms over a certain number of epochs, we hope to move in the coefficient space towards a minimum for the cost function.

In our case, to reduce the computational cost of the process we will use minibatch SGD.

2.4 Logistic regression

Logistic regression is used for classification problems to estimate the probability that a given datapoint $x = (x_1 \dots x_N)$ belongs to discrete classes $y_i = 0 \dots k$. In this case, the outputs will be probabilities that the data belongs to a class, so they have to sum to one and belong to $[0, 1]$.
For our analysis we will consider two classes: $k = 1$. By using the sigmoid function, we can write the probabilities of each of the two outcomes as:

$$p(y_i = 1|x_i, \beta) = \frac{e^{\beta^T x_i}}{1 + e^{\beta^T x_i}}$$
$$p(y_i = 0|x_i, \beta) = 1 - p(y_i = 1|x_i, \beta)$$

where β are the parameters of the model which we want to estimate. To simplify the notation we will call $p(y_i = 1|x_i, \beta) = p_1(x_i, \beta)$ and $p(y_i = 0|x_i, \beta) = p_0(x_i, \beta)$.

In order to do obtain the optimal β parameters, we will use the principle of maximum likelihood and, since maximizing the logarithm of a function is equivalent to maximizing the function itself, we can work with the log-likelihood function:

$$l(\beta) = \sum_{i=1}^{n} \left[y_i \log(p_1(x_i, \beta)) + (1 - y_i) \log(1 - (p_1(x_i, \beta)) \right]$$

The task of maximizing the log-likelihood is equivalent to minimizing the negative log-likelihood, which we will consider as our cost function for logistic regression:

$$C(\beta) = -\sum_{i=1}^{n} \left[y_i \beta^T x_i - \log(1 + e^{\beta^T x_i}) \right] \tag{1}$$

which is also called cross-entropy.
We set its derivative equal to 0:

$$\frac{\partial C(\beta)}{\partial \beta} = -\sum_{i=1}^{N} x_i (y_i - p_1(x_i, \beta)) = 0$$

This expression can be written in a more compact way as:

$$\frac{\partial C(\beta)}{\partial \beta} = -X^T (y - p)$$

where X is the design matrix, y is the vector containing our target values y_i and p is the vector containing the fitted probabilities.
If we now define a diagonal matrix W with elements $p(y_i|x_i, \beta)(1 - p(y_i|x_i, \beta)$ we can write also a compact expression of the Hessian matrix which contains the second derivatives:

$$\frac{\partial^2 C(\beta)}{\partial \beta \partial \beta^T} = X^T W X$$

Now, the values β can be found iteratively by:

$$\beta^{n+1} = \beta^n - \left(\frac{\partial^2 C(\beta)}{\partial \beta \partial \beta^T} \right)^{-1} \left(\frac{\partial C(\beta)}{\partial \beta} \right) = \beta^n - \left(X^T W X \right)^{-1} \left(-X^T (y - p) \right)$$

where we can replace the inverse Hessian with a parameter η which is the learning rate. This leads to the computation of the gradient descent in the following form:

$$\beta^{n+1} = \beta^n - \eta \left(X^T (y - p) \right)$$

Our results could be improved by adding an l_2 regularization term λ. So, instead of (1), we minimize:

$$C(\beta) = -\sum_{i=1}^{n} \left[y_i \beta^T x_i - \log(1 + e^{\beta^T x_i}) \right] + \lambda ||\beta||_2^2$$

3 Discussion and results

3.1 Regression

Firstly we consider a regression setting. We are looking to apply the optimization methods described before to fit a second order polynomial:

$$f(x) = 1 + 3x + 2x^2 + \varepsilon$$

Where $\varepsilon \sim N(0, \sigma)$, follows a normally distributed noise with $\sigma = 0.1$.
For this analysis we select 100 random points with $x \in [0, 1]$.
We have split the data into training and test set with a proportion of 0.2 for the test set.
In order to estimate the performance of regression models we make use of the Mean Square Error (MSE), defined as follows

$$MSE(y_i, \tilde{y}) = \frac{\sum_{i=1}^{n}(y_i - \tilde{y_i})^2}{n}$$

where $\tilde{y}$ stands for the predicted response value of the model, while y_i indicates the observed value of the response variable and n the total number of the observations.

Firstly we implement different variations of gradient methods and compare the results we obtain.
The first crucial tuning parameter to consider for these methods is the learning rate: considering a too small value could result in needing many iterations to reach small MSE values, while a too big learning rate risks the complete divergence of the model.

As shown in Figure 6, the final value that we have chosen for most of the following analysis is $\eta = 0.01$.
Our analysis showed that for values of η over 0.015, the model doesn't converge. Moreover, the heatmap shows that for a low value of η and an insufficient number of epochs, the resulting MSE is much higher.

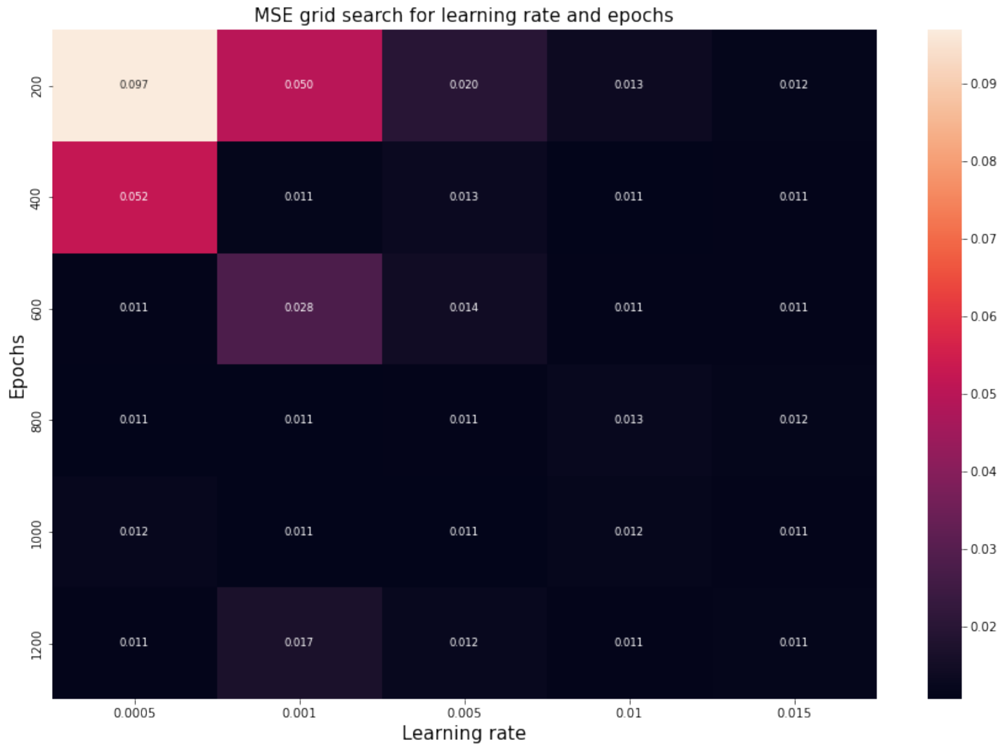

Figure 6: Comparing GD MSE with different learning rates and number of epochs. The optimal MSE is obtained for $\eta = 0.01$ and 400 epochs.

We now consider the Gradient Descent (GD) with and without momentum.
In Figure 7 we can see that the addition of momentum negatively impacts the performance in the first iterations, but the overall convergence for an higher number of epochs is much smoother and doesn't oscillate around the minimum error values like as it happens for the plain GD.
Hence, we conclude that the memory component of the gradient added by the momentum can improve the regression performance.

Moreover, in Figure 8 we look for the optimal value of the momentum parameter and observe a clear reduction of the MSE when compared with GD without momentum.

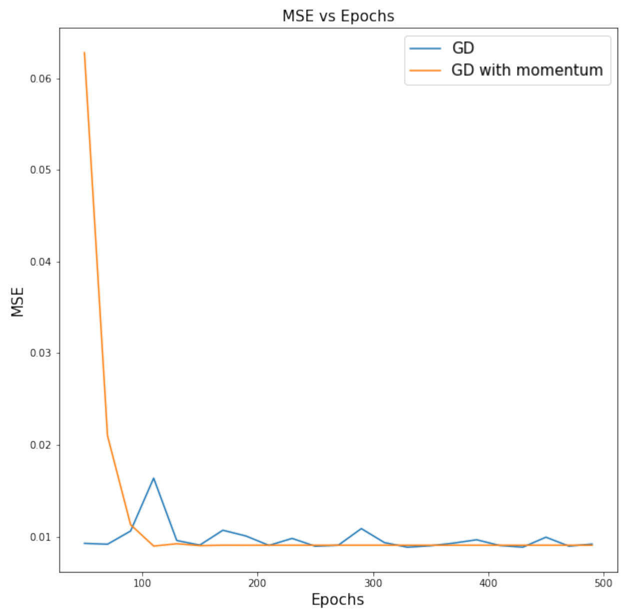

Figure 7: Comparing GD convergence with and without momentum over 500 epochs with $\eta = 0.01$ and $\gamma = 0.9$.

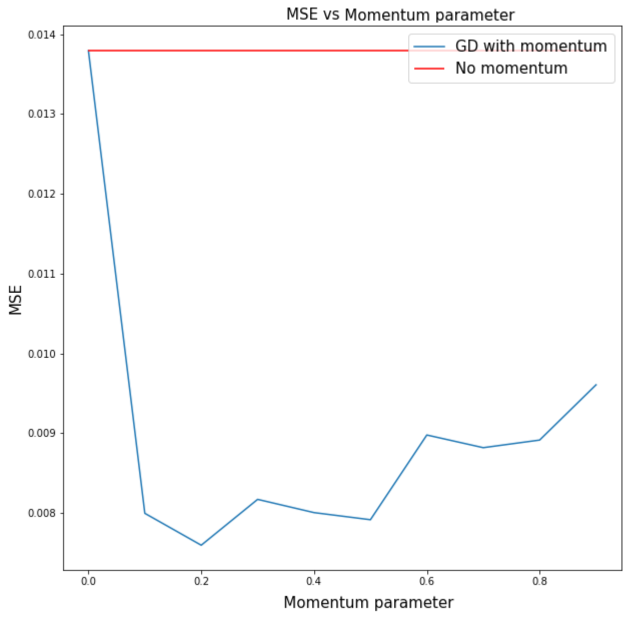

Figure 8: Comparing GD with and without momentum with fixed $\eta = 0.01$ and 400 epochs. In this case, the optimal MSE value is obtained for $\gamma = 0.2$.

We now compare GD with Stochastic Gradient Descent (SGD).
Figure 9 shows the MSE results when changing the learning rate and the size of the minibatches for the method.
When considering a minibatch of size 80, which is the size of the training set, we are effectively performing GD without stochasticity.
The results highlight how the test error in our case is not particularly influenced by the size of the batch and that SGD and GD have similar performance in our analysis.

In figure 10 we can see how the time performance of SGD is worse than GD in our case, since the method has to cycle through all minibatches for each epoch. Given the little difference in MSE results, plain GD would probably be the best choice for analysing our data.

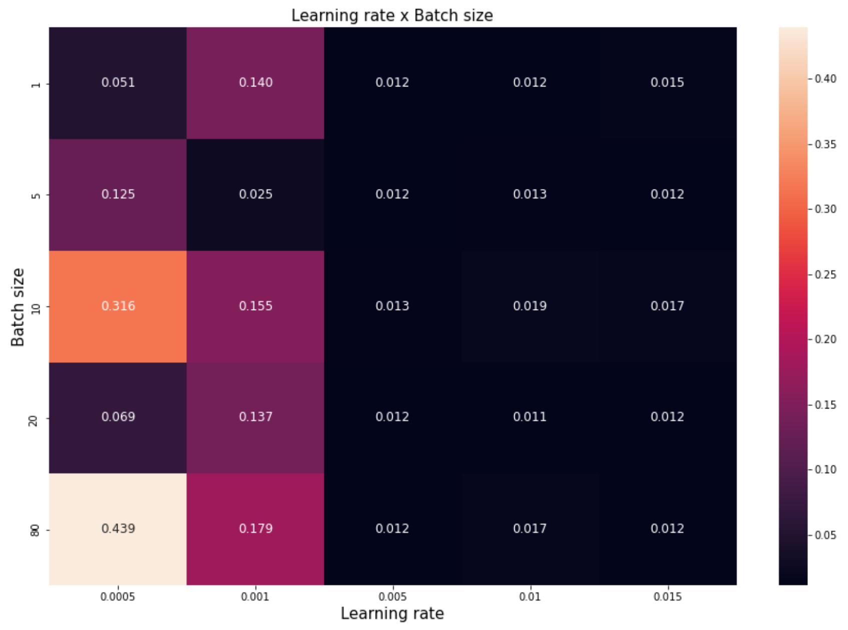

Figure 9: Grid search for changing η and batch size, with number of epochs fixed to 400. The best MSE is obtained for $\eta = 0.01$ and batch size 20.

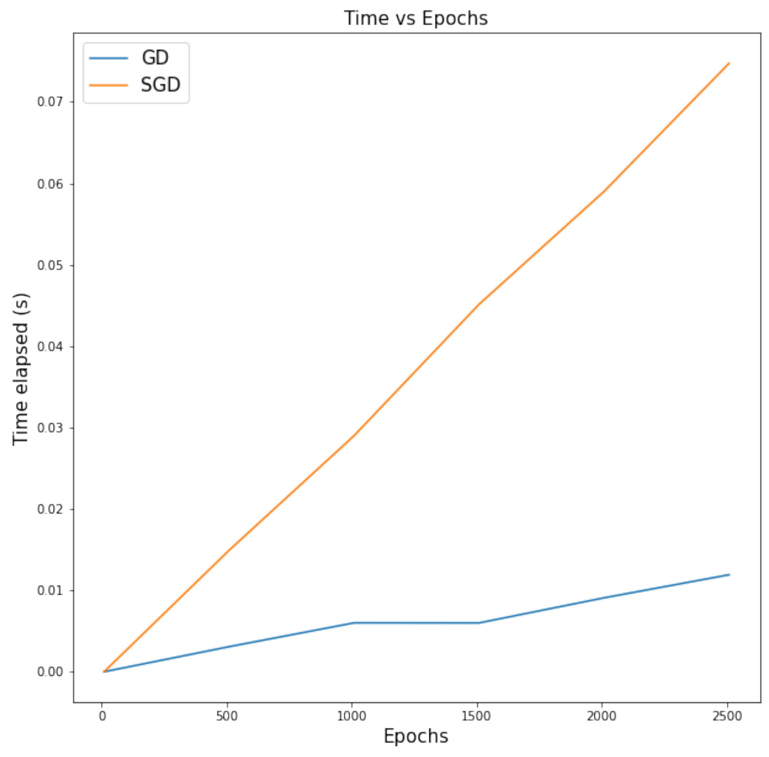

Figure 10: Comparing GD and SGD with batch size 20 on time complexity for changing number of epochs.

3.1.1 Learning rate updating methods

We now introduce adaptive methods for the learning rate in SGD.
Figure 11 presents a general comparison of the convergence of these methods for a growing number of epochs.

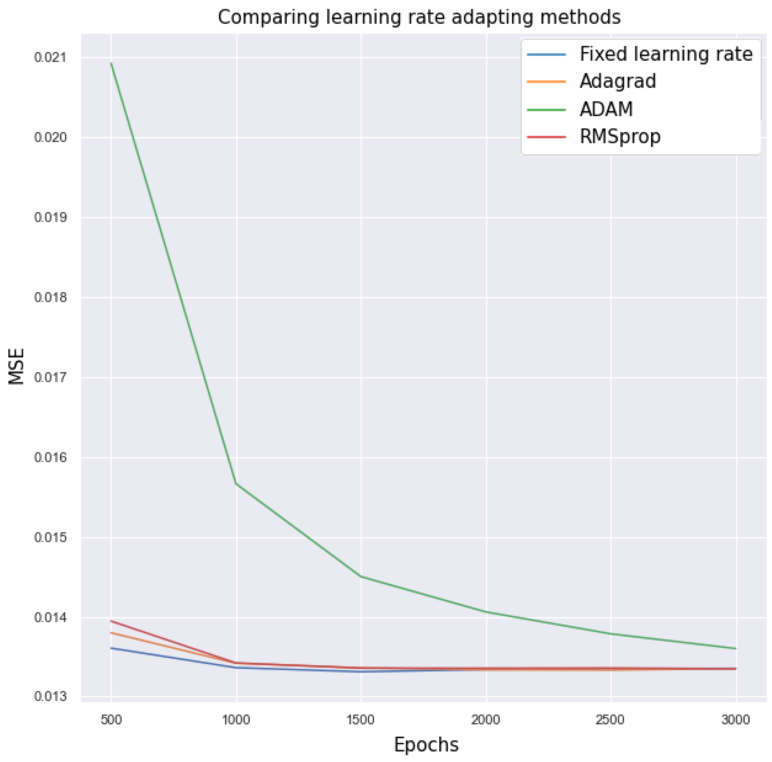

Figure 11: Comparing test MSE for SGD with different learning rate adapting methods for growing number of epochs.

For this analysis we used standard values for the parameters in the methods: $\beta_1 = 0.9, \beta_2 = 0.99$.
As we can see, all methods besides ADAM present a very similar performance. This difference in the MSE values for the ADAM method could be due to mistakes in our implementation and should be explored further.

3.1.2 Ridge regression

We also implement a gradient method for applying Ridge regression to our data: Figure 12 analyses the MSE results for changing values of η and the Ridge parameter λ.

While values of MSE for Ridge and standard OLS ($\lambda = 0$) are very similar, the best results are obtained for $\lambda = 0.1$ and $\eta = 0.01$ showing that the introduction of a regularization parameter can improve our model.

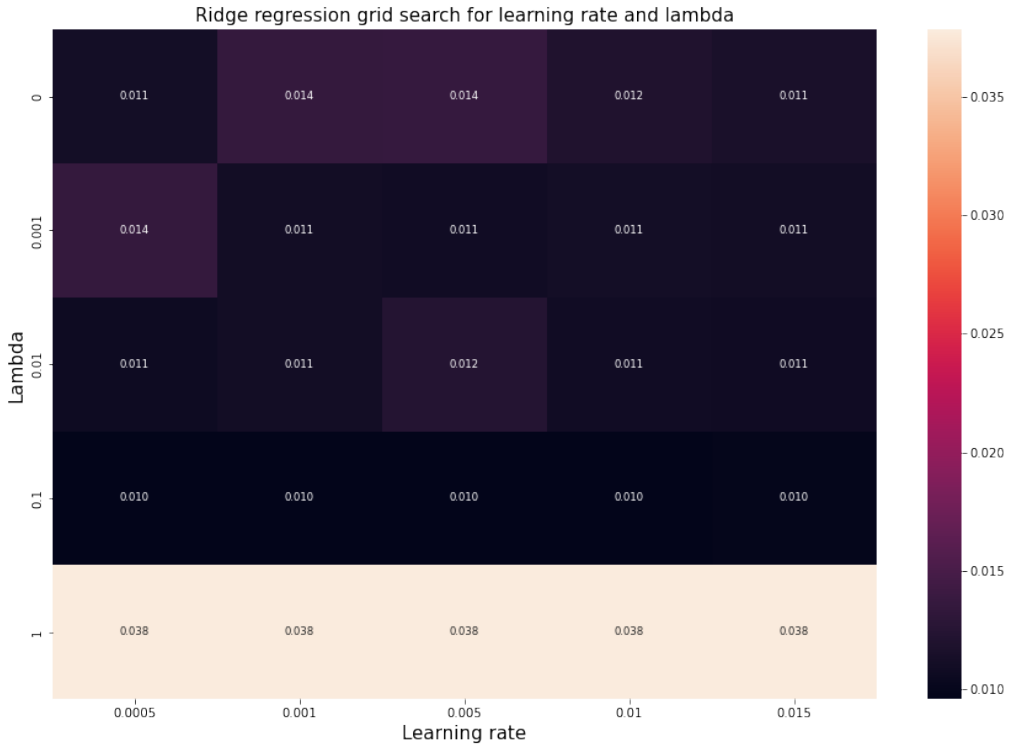

Figure 12: Grid search for the optimal MSE as function of the learning rate η and Ridge regularization parameter λ.

3.1.3 Neural Networks

Finally we try fitting our data by implementing a Feed Forward Neural Network with a flexible number of layers, nodes per layer and different activation functions. Since we are working with regression, the output layer will always contain a single neuron and have a linear activation function.

The cost function considered is the MSE.

Since neural networks generally work with an high number of parameters, in order to obtain proper results in our regression analysis, we consider an higher number of data points to fit: the following analysis is performed on 1000 points generated by the previously presented polynomial function and with an added noise of variance $\sigma = 0.1$.

Firstly we consider a single layer network with sigmoid activation function and tune the number of neurons and the learning rate.
In Figure 13, we can see that even with a low number of neurons we can already

obtain a very low test MSE.
We get the optimal value for 30 neurons and $\eta = 0.001$.

Subsequently we introduce an l_2 norm parameter and tune it together with the learning rate.

This is done by changing the way the weights and biases are updated at each iteration by introducing a term dependent on the current value of the coefficients:

$$W^{(l)} \leftarrow W^{(l)} - \eta\left(\frac{\partial C}{\partial W^{(l)}} + \lambda W^{(l)}\right)$$
$$b^{(l)} \leftarrow b^{(l)} - \eta\left(\frac{\partial C}{\partial b^{(l)}} + \lambda b^{(l)}\right).$$

As seen in Figure 14, the model is optimal without the introduction of the regularization ($\lambda = 0$) and with $\eta = 0.001$.
The variability in our data is probably not enough to justify the need for a regularization of the network coefficients. The role of λ could possibly be further investigated: for example, introducing an higher number of neurons could improve the predictive ability of the network and the introduction of a regularization parameter could help the model deal with overfitting.

We go on to consider a different number of hidden layers in our network, all with a sigmoid activation function and 30 neurons each.
Figure 15 shows how introducing more than one layer in the network has a negative impact on the convergence. We can interpret this as a form of overfitting: the models with multiple layers contain too many weights and biases for the data at hand and tend to fit the training data too closely while worsening their performance on the test set.

We now introduce different activation functions for the neurons in our single layer. The learning rate and regularization parameter have been tuned for each activation function separately (the results can be seen in Appendix **??**).
As shown in Figure 16, the convergence is very quick for all activation functions, and the performance is generally similar. However, while the ReLU and leaky ReLU activation functions present a better initial convergence, the sigmoid shows a smoother descent to the optimal error value.

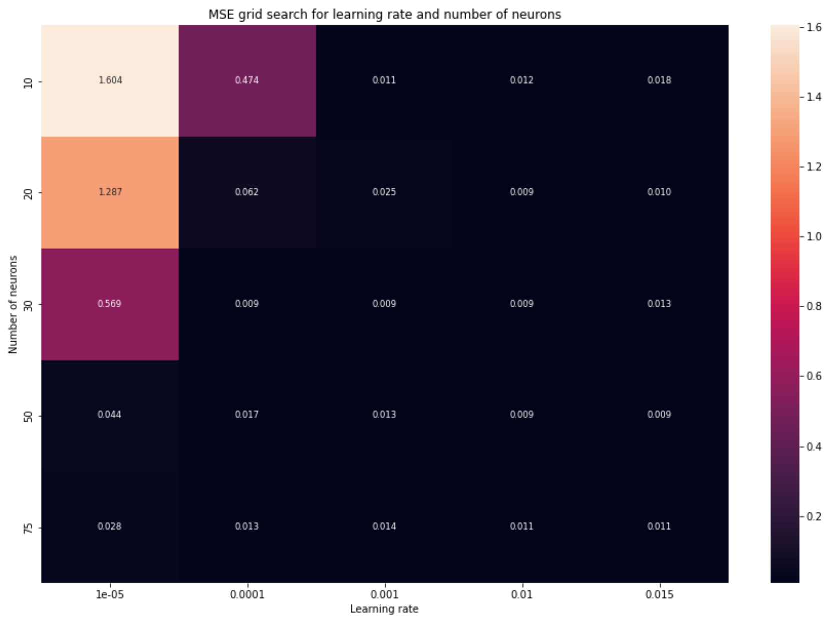

Figure 13: Grid search for the optimal MSE as function of the learning rate η and the number of neurons in the single hidden layer.

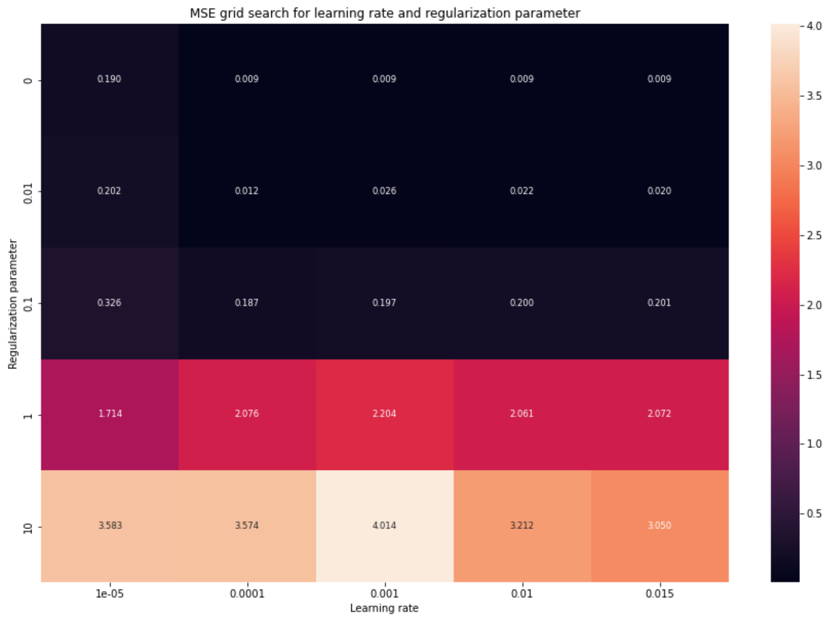

Figure 14: Grid search for the optimal MSE as function of the learning rate η and the regularization parameter λ while considering a sigmoid activation function.

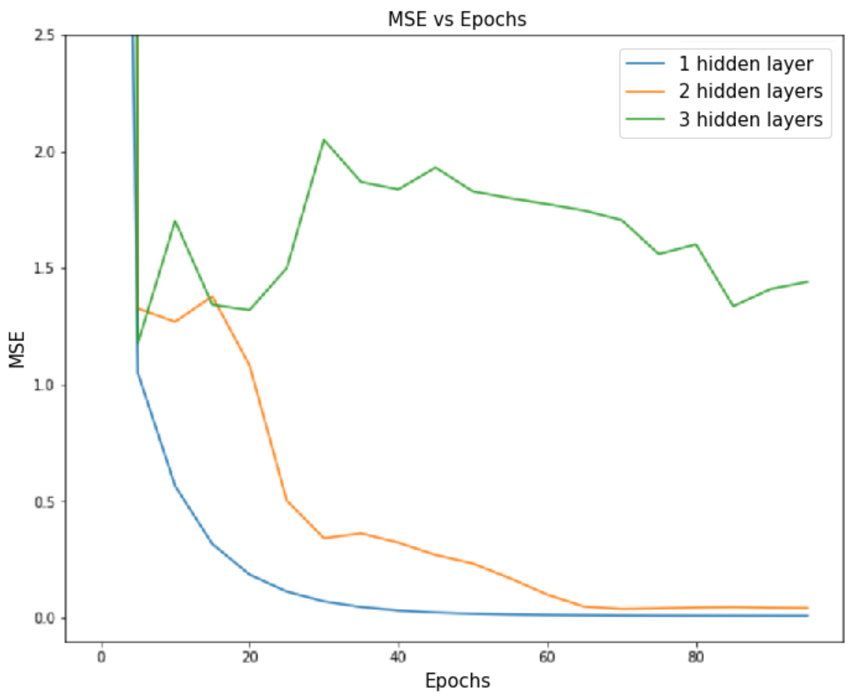

Figure 15: Convergence analysis for Neural Networks with different number of hidden layers with 30 neurons each. To avoid numerical overflow in the training of the networks with multiple layers, we considered $\eta = 1e - 4$.

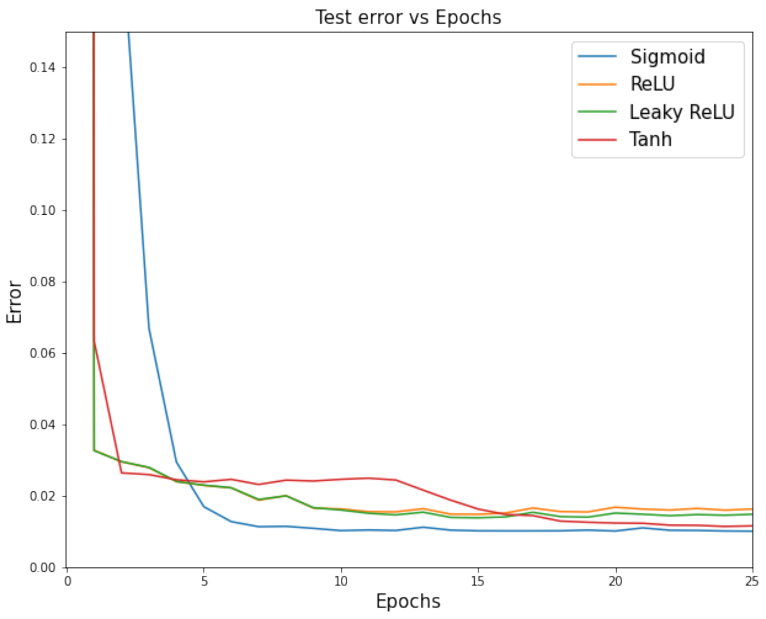

Figure 16: Convergence analysis for Neural Networks with a single hidden layer and different activation functions over 25 epochs.

Finally we test out more variations on the model to understand how they would influence the performance. We compare our previous result obtained with a constant learning rate with the introduction of an adaptive η, which is linearly updated at every epoch following this schedule:

$$\eta_i = \eta_0 + (\eta_n - \eta_0)\frac{i}{n}$$

where η_0 is the initial learning rate, η_n is the final learning rate and $i = 1, ..., n$ where n is the total number of epochs.

Figure 17 shows us how in the second case the convergence is quicker.
Using a learning schedule is beneficial to the results of our model.

We also consider different initialization strategies for the weights and biases in the model.
In Figure 18 we consider weights generated by a normal distribution of mean 0 and variance both 1 and $1/\sqrt{n_{in}}$ where n is the number of inputs in the node.
As explained by M. Nielsen [4], this smaller value for the variance of weights should reduce the likelihood of a neuron saturating and consequently speed up convergence.

We find out that this is not the case for our data, as introducing the different variance value results in a worse performance by the model.

As for the biases, we consider both a constant initial value and bias generated by a standard normal distribution: as shown in Figure 19, this doesn't seem to have any observable effect on the results of the model.

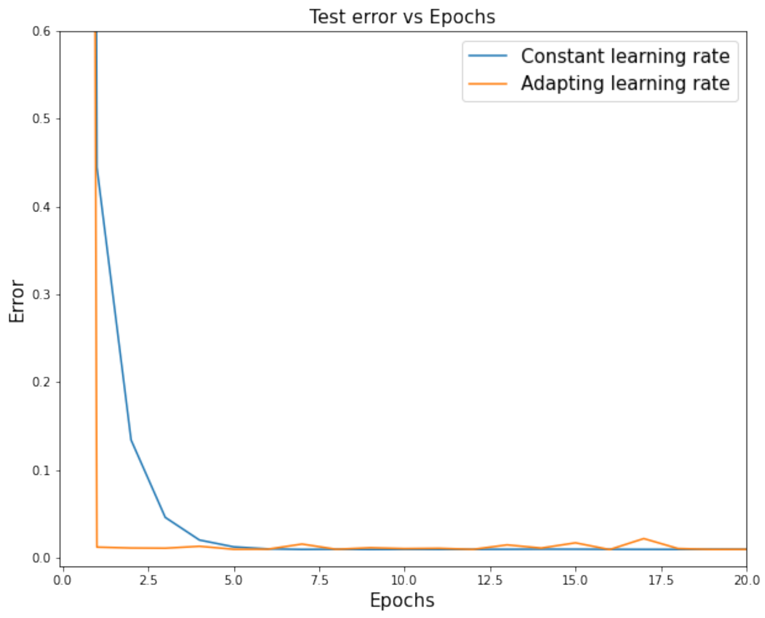

Figure 17: Convergence analysis for Neural Networks with constant learning rate $\eta = 0.001$ and with an updated learning rate linearly decreasing from 0.01 to 0.0001.

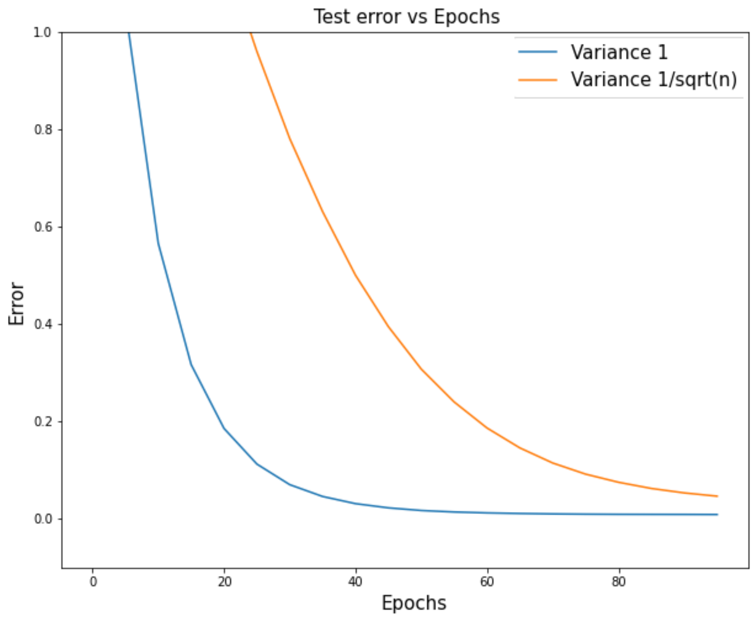

Figure 18: Convergence analysis for Neural Networks with weights generated from a normal distribution with different variances.

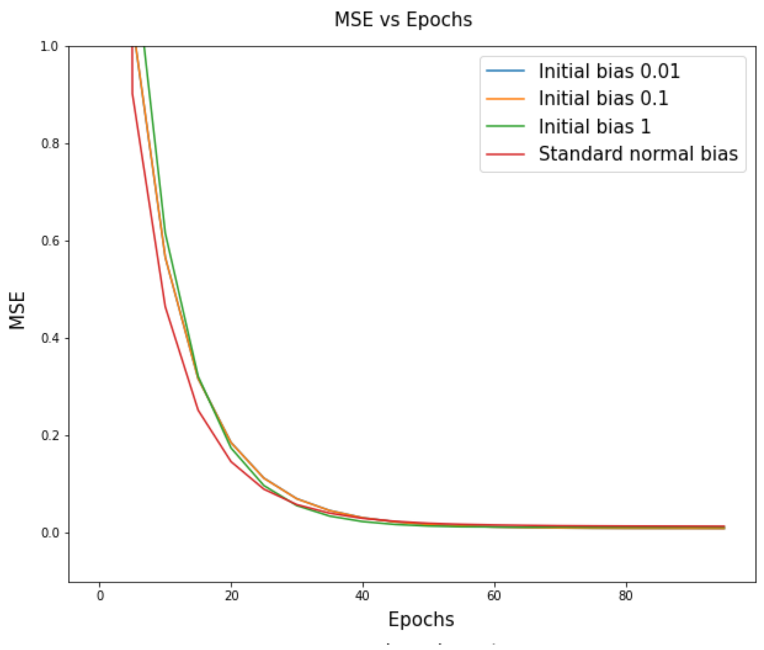

Figure 19: Convergence analysis for Neural Networks with bias initialized as different constant values and from a standard normal distribution.

3.2 Classification

In the following section we will apply a FFNN and a logistic regression in order to classify whether a patient is affected by a malignant tumor or not.
In order to evaluate the efficiency of the methods we use the so-called *accuracy* score.
The accuracy is the number of correctly guessed targets t_i divided by the total number of targets n, that is

$$Accuracy = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n}$$

where I is the indicator function 1 if $t_i = y_i$ and 0 otherwise. Here y_i represents the output of classification processes.

3.2.1 Breast Cancer Dataset

We now start our classification analysis on the *Wisconsin Breast Cancer Data*. Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass on 569 patients.
Considering 30 predictor variables such as *mean Radius*, *mean Perimeter* and *mean Area*, we will try to implement a FFNN and a logistic regression to classify tumor in *bening* and *malignant*.

3.2.2 Standardization

Firstly, we have split the data into training and test set with a proportion of 0.2 for the test set. Then, we have standardized predictors values in both the training and test data by the mean and standard deviation of the training set.
It is important to do to so in order to have common scaled features, which will then influence the scale of the weights and biases in our network.

3.2.3 Neural Networks

We perform a similar analysis as the one previously shown for the neural network in the regression setting.

Since we are considering two classes, the output layer for our network will always contain a single neuron with a sigmoid activation function: this way, the output will be between 0 and 1 and could be interpreted as the probability of the datapoint belonging to class 1 (the tumor being malignant).
In particular, our prediction will be 0 if the output is < 0.5 and 1 otherwise.

In the classification case, the cost function we use is the cross-entropy.

The results of tuning the parameters for the method, considering varying number of neurons and hidden layers, as well as different activation functions is shown in the following figures (20, 21, 22, 23).

The final chosen model has a single hidden layer with 20 neurons and sigmoid activation function with learning rate $\eta = 0.01$ and without introducing any regularization.

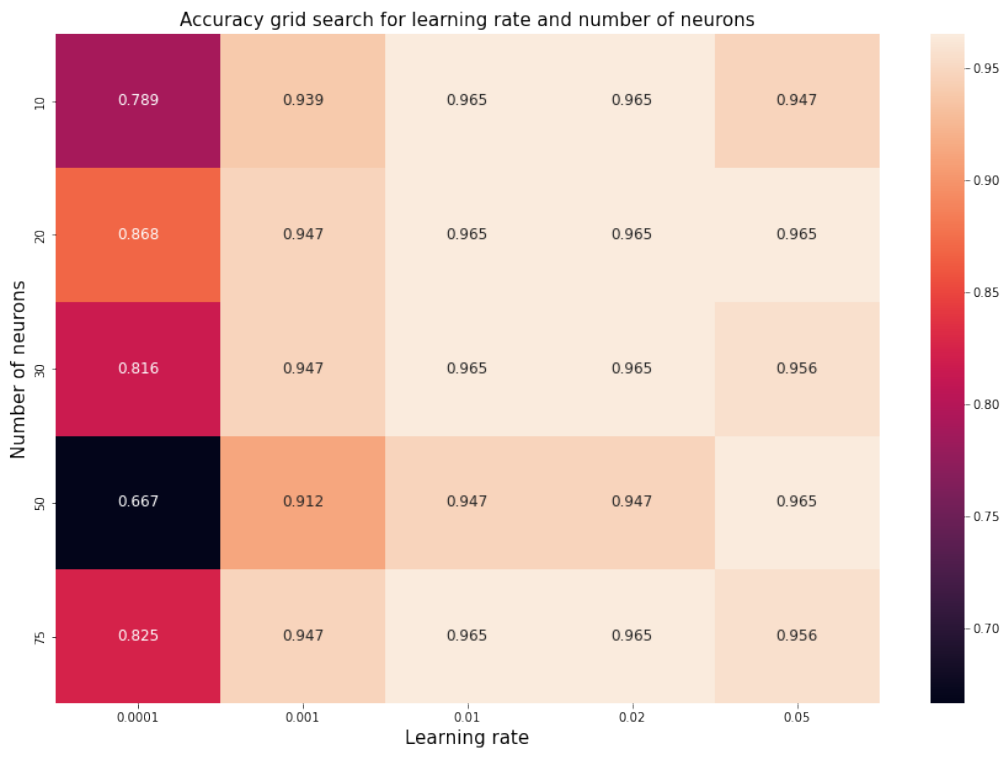

Figure 20: Grid search for the optimal accuracy as function of the learning rate and number of neurons in the single hidden layer. We consider 200 epochs and a sigmoid activation function. The chosen parameters are $\eta = 0.01$ and 20 neurons.

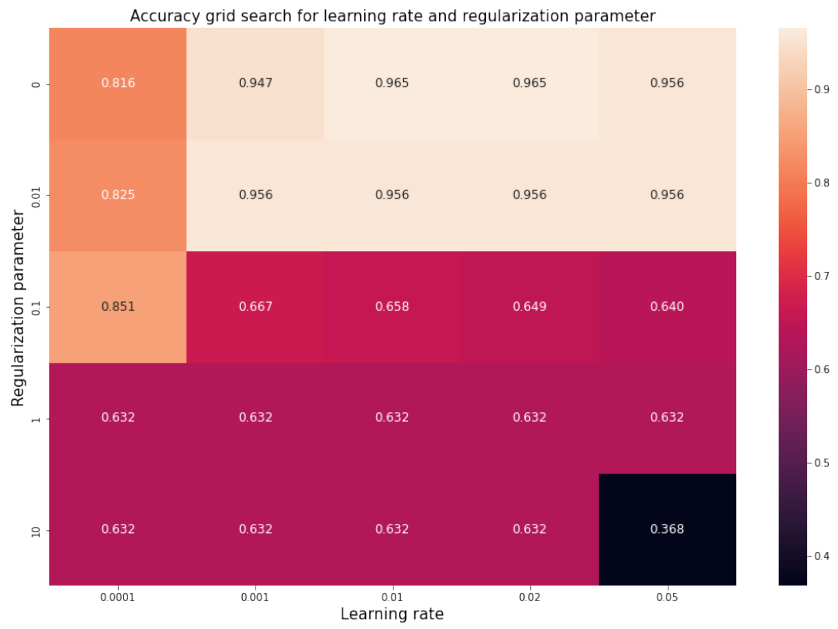

Figure 21: Grid search for the optimal accuracy as function of the learning rate and regularization parameter. The chosen parameters are $\eta = 0.01$ and $\lambda = 0$.

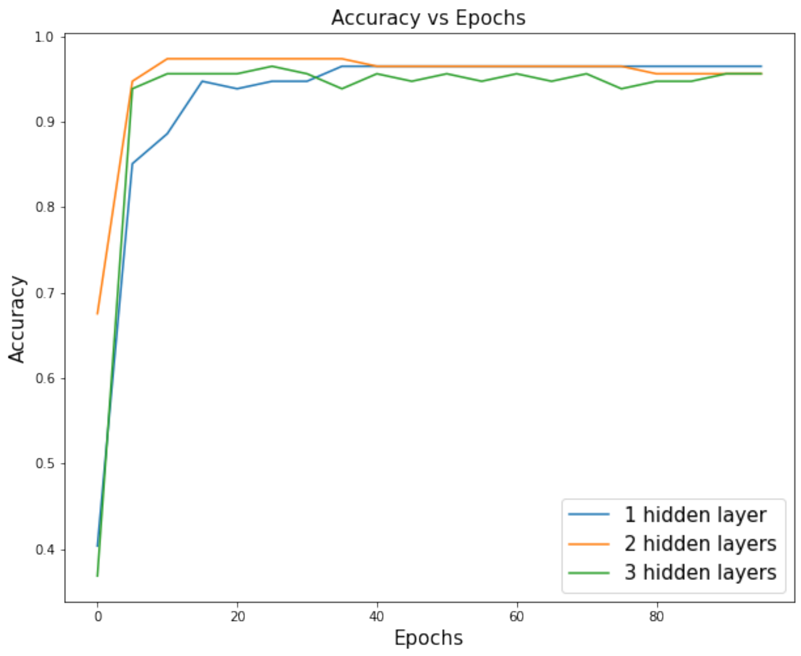

Figure 22: Convergence analysis for Neural Networks with different number of hidden layers with 20 neurons each.

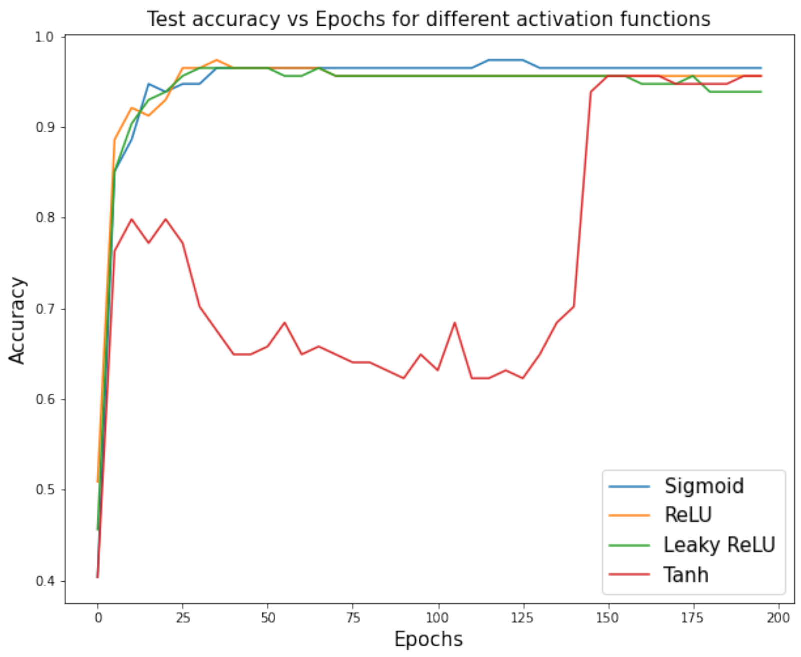

Figure 23: Convergence analysis for Neural Networks with a single hidden layer and different activation functions over 200 epochs.

3.2.4 Logistic Regression

We now want to compare the results of the FFNN in the classification setting with another method. To do this, we implemented a logistic regression code using the SGD algorithm with the introduction of a l_2 penalty term.

In Figure 24 we look for the optimal values of the parameters.

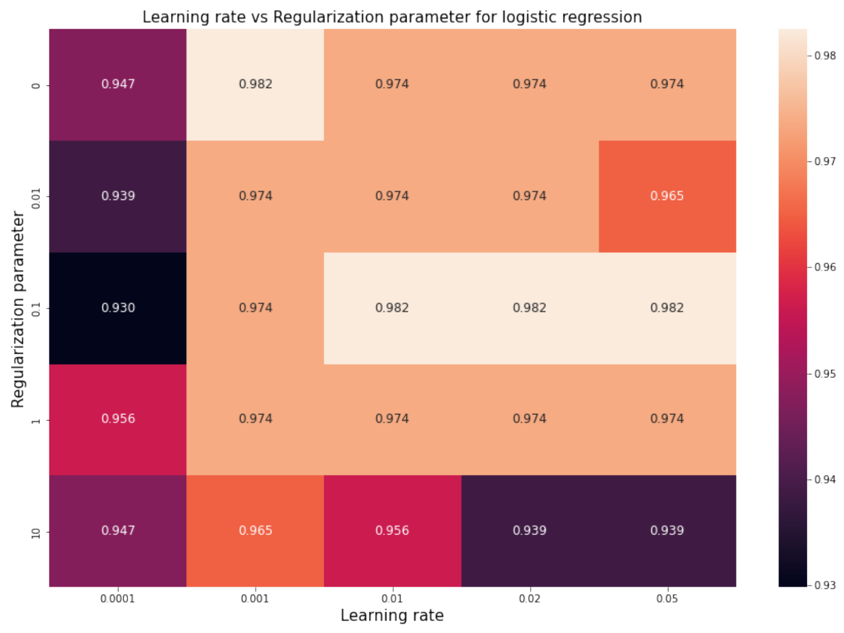

Figure 24: Grid search for the optimal accuracy as function of the learning rate and regularization parameter. The chosen parameters are $\eta = 0.01$ and $\lambda = 0.1$.

4 Conclusion

The aim of this project was to implement, study and compare the performance of different methods for regression and classification.

We started by implementing gradient optimization methods such as Gradient Descent and mini-batch Stochastic Gradient Descent. We then implemented a code for a Feed-Forward Neural Network with a flexible number of layers and nodes and with different activation functions, using the back propagation algorithm for training. This network was tested for both regression and classification problems.

In the regression setting we considered the fitting of a second order polynomial function with an added normal distributed noise term. The final comparison is done on 1000 generated points and the results for the different methods tested are reported in the following table:

Model	**Test error**
OLS with GD	0.0111
OLS with SGD	0.0115
Ridge Regression with SGD	0.0099
Feed Forward Neural Network	0.0093
MLPRegressor	0.0113
TensorFlow Model	0.0090

We considered both the methods we implemented and two models obtained from python libraries: `MLPRegressor` from `Scikit-learn` and the neural network from `TensorFlow`. As presented, the resulting MSE are similar across all models, which proves that the methods we implemented are suitable for regression problems.

Future expansions of this study could implement the same methods on real datasets for regression, to verify if in other settings the use of SGD compared to plain GD or the introduction of regularization parameters shows a more sizable difference in the results.

In the classification setting, on the other hand, we compared the performance of a Feed Forward Neural Network and logistic regression with mini-batch SGD on the Wisconsin Breast Cancer Dataset.
The accuracy results obtained with different models are reported in the following table:

Model	**Accuracy**
FFNN	0.9649
Logistic regression	0.9824
MLP Classifier	0.9736

The final comparison shows that the logistic regression model performs slightly better than the neural network. We also compared our results with the one provided by the `MLPClassifier` tool in the `scikit-learn` python library: the similarity in the results once again testifies for the suitability of the models implemented for problems in the classification setting.

5 Appendix

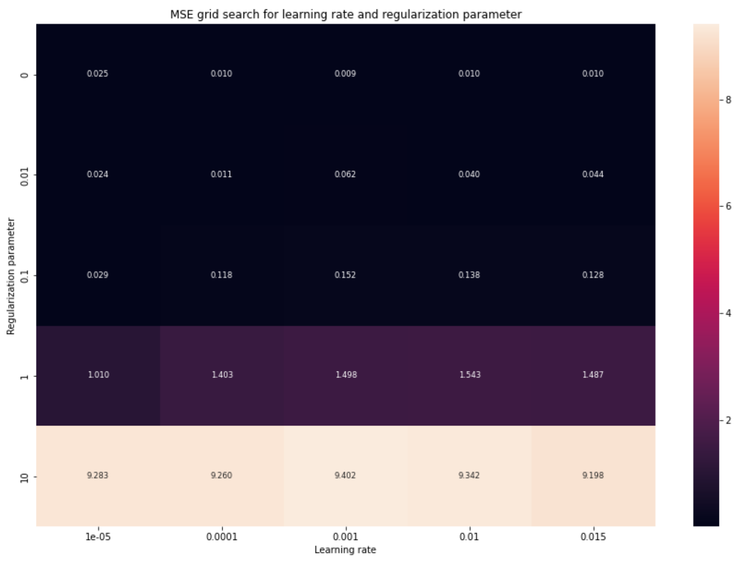

Figure 25: Grid search for the optimal MSE as function of the learning rate η and the regularization parameter λ while considering a hyperbolic tangent activation function.

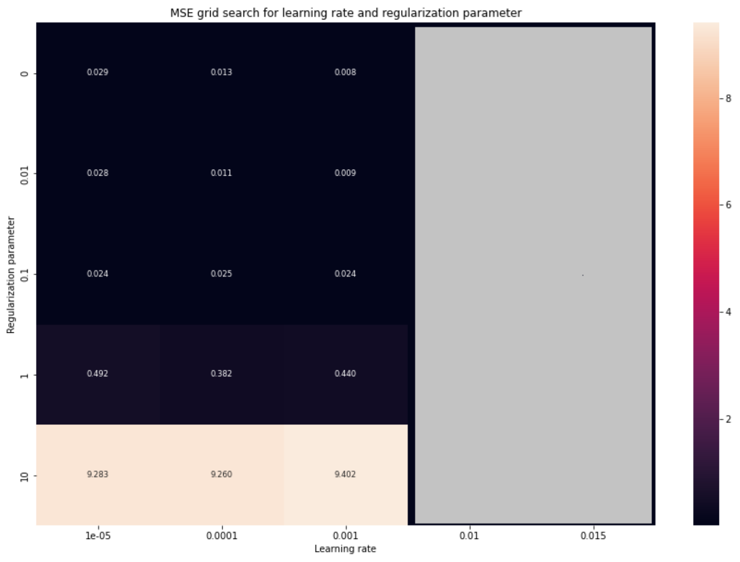

Figure 26: Grid search for the optimal MSE as function of the learning rate η and the regularization parameter λ while considering a ReLU activation function. The grey squares indicate that the model did not terminate due to numerical overflow.

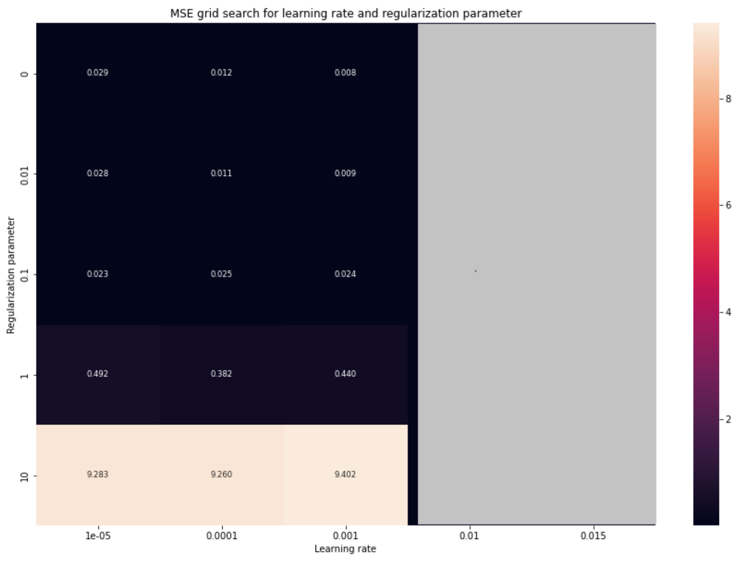

Figure 27: Grid search for the optimal MSE as function of the learning rate η and the regularization parameter λ while considering a leaky ReLU activation function.

References

[1] W.S. McCulloch , W. Pitts, A logical calculus of the ideas immanent in nervous activity, Bull. Math. Bioph. 5, 1943, pp. 115–133.

[2] A. Géron, Hands-on Machine Learning with Scikit-Learn, Keras,and Tensor-Flow. O'Rally, second edition, Sebastopol, CA 2019.

[3] M. Ledum, A Computational Environment for Multiscale Modelling, chapter 7. MA thesis, Universitetet i Oslo, `https://www.duo.uio.no/handle/10852/61196`, 2017.

[4] M. Nielsen, Neural networks and deep learning, `http://neuralnetworksanddeeplearning.com/index.html`, Chapter 3 - *weights initialization*, 2019.

[5] M. Hjorth-Jensen, 2022, GitHub, `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html`, chapter 6 7, 13.