# UiO - FYS-STK4155: PROJECT 2: Deep Neural Networks for Regression and Classification cases

**Fábio Rodrigues Pereira**
fabior@uio.no - github: @fabiorod

November 18, 2020

**Abstract**

This project aims to study many different Gradient Descent (GD), Momentum Gradient Descent (GDM), Stochastic Gradient Descent (SGD), Mini-batch Stochastic Gradient Descent (Mini-SGD), Mini-batch Momentum Stochastic Gradient Descent (Mini-SGDM), Neural Network (NN), and Deep Neural Networks (DNN) techniques applied on a GeoTIF terrain data containing region heights from a place near Stavanger in Norway. We are interested in predicting this region's measurements and finding a model that would overcome the accuracy metrics obtained from traditional methods as Ordinary least squares (OLS) and Ridge regression (RIDGE). From project 1, the best testing Mean Squared Error (MSE) achieved for OLS and RIDGE, respectively, was 1.11 and 1.44. In this project 2, we go through all parameters for the models above, like the number of hidden layers, the number of neurons, activation functions, cost functions, weights initialization methods, Learning Rate ($\eta$), decay, lambda 'l2' regularization, batch size, epochs, training elapsed time, and see that the best testing MSE score achieved for an SGD was 2.99, for a MiniSGDM was 4.18, and for a DNN was 7.46. These scores do not perform equally well as the more traditional regression methods, but some practical benefits are considered, for example, the unnecessary design matrix to feed the DNN models and more efficient/lower training/learning time systems.

Furthermore, this project 2 also aims to study the application of Neural Networks on MNIST and Breast Cancer data-sets. Now, the essence of the problems is not regression anymore, but classification. The former, MNIST, is a classical data-set with multi-classes targets among 0-9, and the latter, Breast Cancer, is a typical data-set with binary target classes between 0-1. Hence, in this part of project 2, we go through all parameters for the models Multi-Layer Perceptron (MLP) and Logistic Regression. For the MNIST data-set, we will see that the best testing accuracy prediction of 97.2 percent was achieved using an MLP model with the Sigmoid activation function and Accuracy-score cost function. For the Breast Cancer data-set, the best testing accuracy prediction of 99 percent was acquired using an MLP model with the Softmax activation function and Accuracy-score cost function.

# Contents

# 1 Introduction

Supervised machine learning techniques for regression and classification scenarios have gained consistent developments throughout the last decade. Many different private and academic contributions have helped create more reliable, efficient, and precise methods to acquire equally or superior significance as the old traditional systems.

Artificial Neural Networks (ANN) techniques such that Multi-layer Perceptron (MLP) have proved an incredible potential. However, these models need to be studied and perfected even more by professionals and researchers. Our job here is to develop, analyze, and investigate different generalized techniques applied in as many different real domains as imaginable, with as much efficiency and accuracy as possible. This duty is the motivation of our scientific report, which will explore the subject of Neural Networks and see how this method can be enhanced by assessing more reliable results.

For that, this report will first investigate GeoTIF image data of an area near Stavanger in Norway on Gradient Descent (GD) models, essentially Momentum Gradient Descent (GDM), Stochastic Gradient Descent (SGD), Mini-batch Stochastic Gradient Descent (Mini-SGD), Mini-batch Momentum Stochastic Gradient Descent (Mini-SGDM). After that, we will apply Feed-Forward and Back-Propagation techniques in a model so-called Multi-Layer Perceptron (MLP) that fitts as a base for Neural Networks (NN) or Deep Neural Networks (DNN) principles. Therefore, we will examine the prediction accuracy results of its heights as a function of the model's parameters.

After that, we will cover a classification case called MNIST, which classifies handwritten digit images into one of its ten classes (0–9). The problem we are dealing with is essentially a Multi-class classification that we will be using the MLP model with selected activation and cost functions.

In the end, we will analyze the breast cancer data, a binary classification problem with response variable 0 for patients not containing breast cancer and 1 for breast cancer patients. Our MLP model with selected activation and cost functions will also be applied to this problem. We will likewise apply a binary classification model called Logistic Regression on this data-set and then compare it to the MLP model.

The following topics of this dissertation will cover these methods under the studied models. First, this report will approach the theories utilized in the discussion section. After, it will divide the discussion and result topics into different subsections, each one dealing with a different subject and with a separate python test file in our GitHub repository. In the end, a conclusion will discuss the pros and cons of the methods and possible improvements and perspectives for future work.

## 1.1 Source code

All codes utilized in this project is written in Python v.3.8, and found in the GitHub repository at https://github.com/fabiorodp/UiO-FYS-STK4155/tree/master/Project2. The repository contains:

- data/SRTM_data_Norway_1.tif - Data utilized in this project.
- package/ - Directory containing the collection of python's methods utilized in the project.
- report/ - Directory containing the written report in latex and pdf.
- partA.py - Test script for part A of the project.
- partBandC.py - Test script for part B and C of the project.
- partD.py - Test script for part D of the project.
- partE.py - Test script for part E of the project.

# 2 Theory

## 2.1 Gradient Decedent

Gradient descent (GD) or Batch gradient descent (BGD) is a first-order method for finding a local or global minimum of a multi-variable differentiable function. To do that, it takes negative gradient steps of a function at a current point. We denote here ($\boldsymbol{w_0}$) as the initial/start point, and the processes are described as $w_{n+1} = w_n - \eta \nabla_{\boldsymbol{w}} f(\boldsymbol{w_n})$, where $\eta \in R$ is the Learning Rate which must be small enough; $\nabla f(w_n)$ is the gradient of the cost function (MSE); $\boldsymbol{w}$ is the set of coefficients or weights.

It is important to remark that the Leaning rate ($\eta$) is the step size or amount of the gradient that will update the weights of the model during training. Brilliant illustrations were extracted from [3][p. 157] to visually explain the effect of a too-small Learning rate or too large learning rate:

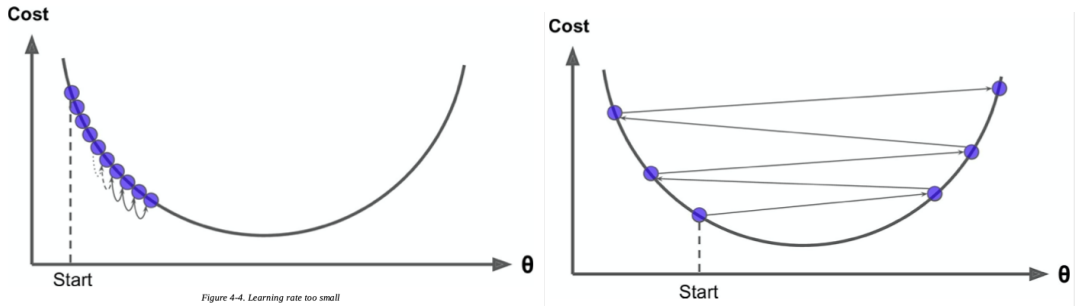



Figure 1: Left-illustration visually demonstrates the effect of a too-small learning rate. Right-illustration visually demonstrates the effect of a too-large learning rate.

In summary, the algorithm will need many iterations to achieve the local or global minimum if the learning rate is too low. On the other hand, if the learning rate is too large, the algorithm diverges and never achieves the local or global minimum.

The MSE cost-function is utilized to compute the linear regression case gradient, which also is excellent for these cases because it is a convex continuous function. This continuity and convexity properties mean that the gradient descent will always go in the local or global minimum direction as illustrated under:

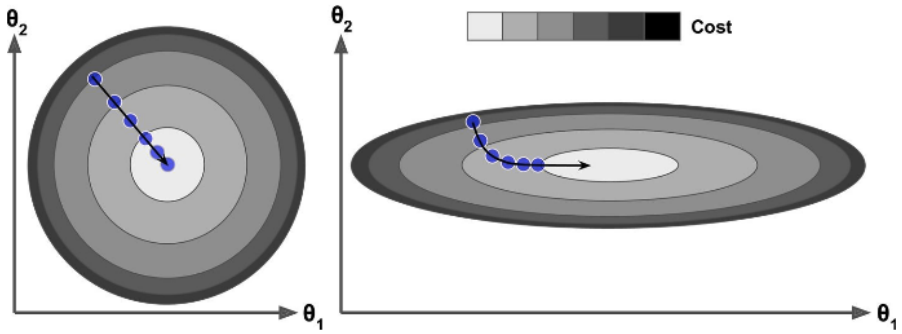



Figure 2: Figures extracted from [3][p. 159]. Left-side figure shows a fast Gradient Descent on a training set where features 1 and 2 have the same scale. Right-side figure shows a slower Gradient Descent on a training set where feature 1 has smaller values than feature 2.

From Calculus, the gradient of MSE can be calculated as:

$$\begin{aligned} w_{n+1} &= w_n - \eta \nabla_{\boldsymbol{w}} f(\boldsymbol{w_n}) \\ &= w_n - \eta * (\frac{2}{m} * \boldsymbol{X}.T * (\boldsymbol{X} * \boldsymbol{w} - \boldsymbol{z})) \end{aligned}$$

where $\boldsymbol{X}$ is the design matrix with explanatory variables; $\boldsymbol{z}$ is the set with response variables; $n = [0, 1, 2, ..., m], n \in N$ is the number of steps/interactions/epochs.

The algorithm called BGDM in package/ directory, when the parameter gamma=0, performs this method, which initializes its' initial points or weights randomly. Then, the gradient is calculated for every epoch and decreased from those weights described above. It is essential to state that the BGDM computes an entire batch's gradient, which means a unique block containing all training data samples and features (design matrix $\boldsymbol{X}$).

### 2.1.1 Learning Rate ($\eta$) with decay/schedule

The Learning Rate ($\eta$) with decay/schedule changes during learning between epochs/interactions. There are many different types of $\eta$ schedules; however, we will focus on the most used one called time-based. For that, a parameter decay is introduced, and it will alter $\eta$ depending on the learning rate of the previous time iteration according to the mathematical formula:

$$\eta_{n+1} = \frac{\eta_n}{1 + decay * epoch}$$

where $\eta_{n+1}$ is the learning rate with decay; $\eta_n$ is the previous learning rate; decay is a constant parameter; epoch is the step index.

### 2.1.2 Stochastic Gradient Decedent

Stochastic Gradient Decedent (SGD) is a stochastic approximation of the gradient descent. It replaces the gradient from (BGD), calculated from an entire data set $\boldsymbol{X}$, by a gradient calculated from a randomly selected feature vector of a sample from the data set $\boldsymbol{X}$. Put differently, the main difference between BGD and SGD is that the former updates its weights after the entire data-set gradient computation. The latter updates its weights after every unique sample and its features.

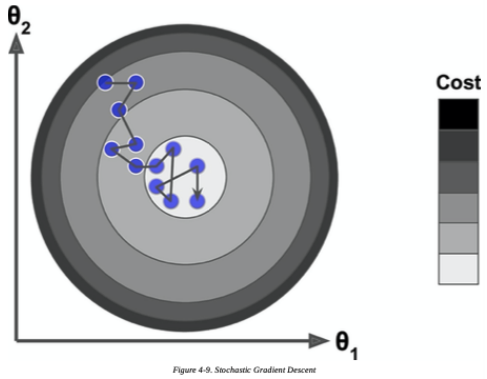



Figure 3: Figure extracted from [3][p. 163] illustrating the effect of the SGD during training.

Therefore, SGD is faster than BGD. However, as Fig.3 shows, due to SGD's stochastic nature, the gradient is much less regular, as stated by *Aurélien Géron* in [3][p. 163] "*instead of gently decreasing until reaches the minimum, the cost function will bounce up and down, decreasing only on*

*average. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down. So once the algorithm stops, the final parameter values are good, but not optimal*".

The algorithm called MiniSGDM in package/ directory, when parameters gamma=0 and batch-size=1, performs the method above.

#### 2.1.3 Mini-batch Stochastic Gradient Decedent

Mini-batch Stochastic Gradient Decedent (Mini-SGD) differs from SGD in the number of samples and its features picked randomly. The former uses blocks of mini-batches containing more than 1 sample and its features from the data-set x, then the number of blocks of mini-batches will be the total of samples divided by the fixed size of the mini-batches. This process reduces the number of times the weights update turning the system faster than BGD and SGD. Therefore, Mini-batch SGD has tremendous benefits over BGD and SGD, especially in high-dimensional optimization problems, because it reduces the time and computation's cost resulting in a faster fitting for the model without losing the accuracy.

The algorithm called MiniSGDM in package/ directory, when parameters gamma=0 and batch-size $> 1$, performs the method above.

#### 2.1.4 BGD, SGD and Mini-batch SGD with momentum

BGD, SGD and Mini-batch SGD can be used with a momentum parameter ($0 < \gamma <= 1$) that plays the role of a memory of the gradient steps' direction [4], as follows:

$$\boldsymbol{v}_{n+1} = \gamma \boldsymbol{v}_n + \eta_{n+1} \nabla_{\boldsymbol{w}} f(\boldsymbol{w_n})$$

where $\boldsymbol{w}_{n+1} = \boldsymbol{w}_n - \boldsymbol{v}_n$; $\boldsymbol{v}_n$ is a running average of recently encountered weights [4].

The algorithms called BGDM and MiniSGDM in package/ directory, when parameter $0 < \gamma <= 1$, perform the method above.

### 2.2 Deep Neural Networks

The deep neural network (DNN) is part of the neural network (NN) and machine learning (ML). *Sebastian Raschka* and *Vahid Mirjalili* define deep learning in [2][p. 542] as "*understood as a set of algorithms that were developed to train artificial neural networks with many layer most efficiently*".

Before start our study of DNN we need to understand the process for a simple NN. In [2][p. 547] has a brilliant picture illustrating the Adaptive Linear Neuron (Adaline algorithm) which resembles in part the processes of a NN:

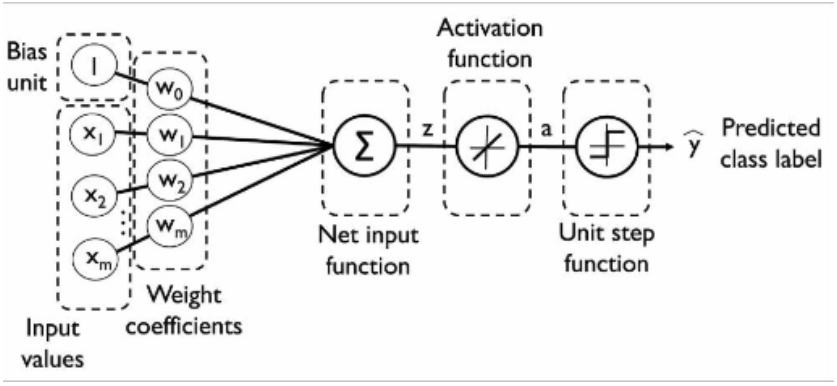



Figure 4: Figure extracted from [2][p. 547] illustrating Adaline process.

Our project's scope is not to explain the Adaline process, but, in a straightforward explanation, the process consists in, for every epoch, a matrix multiplication between training data and weights plus bias, which the result is called net input. Then it is submitted to an activation function to compute the gradient update and update the weights for the next epoch or obtain the prediction. For more details about Adaline, see [2][Chapter 2].

This process is very similar to the NN, but only considers an input layer connected to the output layer without any hidden layer. The difference between this process and the NN process is that NN has a hidden-layer between the input and output layers. In other words, Adaline has two layers, and NN has three layers in total. Regarding DNN, it has an input layer, two or more hidden layers, and output layer. The illustration for NN follows:

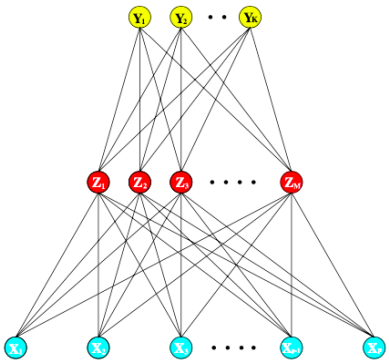


Figure 5: Figure extracted from [1][p. 393] illustrating the connections of a Neural Network. The circles represent nodes/neurons. Green-circles, red-circles, and Yellow-circles are the neurons for the input, hidden, and output layers, respectively.

The way that NN connects the input-layer to the hidden-layer and output-layer is called Feed-forward. The method to update the weights is called back-propagation.

### 2.2.1 Feed-Forward

The computations of the connections between input, hidden, and output layers are made by a technique called feed-forward (FFNN), illustrated in Fig.5. Basically, for each training instance (mini-batch containing vectors of features denoted by $\mathbf{x}$), the algorithm computes the net input ($\mathbf{z}$) throughout matrix-vector multiplications between inputs ($\mathbf{a}^{[input]}$) and weights ($\mathbf{w}$) plus bias ($\mathbf{b}$). The net input ($\mathbf{z}$) will be submitted to an activation function, and its results/output are denoted by ($\mathbf{a}^{[output]}$), such that:

$$\mathbf{x} = \mathbf{a}^{[input]} \rightarrow \sum_{j=1}^{M} \mathbf{w}_{ij}\mathbf{a}^{[input]} + \mathbf{b}_i = \mathbf{z} \rightarrow f(\mathbf{z}) = \mathbf{a}^{[output]}$$

where $i :=$ each node in the single hidden-layer; $j :=$ each node in the input/output layer.

Now, suppose we have a DNN with multiple hidden-layers. Hence, the FFDNN mathematical generalization will be:

$$\mathbf{x} = \mathbf{a}^{[0]} \rightarrow f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^{3} f^{l}\left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l-1}\left(\ldots f^{1}\left(\sum_{n=1}^{N_0} w_{mn}^{1}\mathbf{a}^{[0]} + b_m^1\right)\ldots\right) + b_k^2\right) + b_1^3\right] = a_i^{l+1}$$

which illustrates FFDNN property [4] of a Multi-layer perceptrons (MLP).

### 2.2.2 Back-propagation

After the FF process, the output layer contains the predictions of the model. Next, the model compares the predictions and the real targets by applying a specific loss-function. Based on the loss-function results, the system will update the weights and bias, improving the performance by mini-batch stochastic gradient descent. Typically, the model initializes the weights and bias randomly, resulting in bad predictions initially. Still, as soon as the weights and bias are updated, the loss decreases to a local or global minimum, and the predictions get more accurate.

As said, back-propagation is the method of how the model will update the weights and bias. Now, the computations occur from the output layer to the input layer in a backward process. Utilizing the chosen loss-function (C) and by the chain rule [3][p.396], the output error $\delta_L$ is:

$$\delta^{[l]} = \frac{\partial C}{\partial a^{[l]}} * \frac{\partial a^{[l]}}{\partial z^{[l]}},$$

and the error for the other hidden-layers is:

$$\delta^{[l]} = \delta^{[l-1]} W^{[l-1].T}[f(z^{[l]})]'_W,$$

where $f^{'}$ is the derivative of the activation function utilized.

Hence, the update of the weights and biases are:

$$W^{[l-1]} = W^{[l-1]} - \eta a^{[l-1].T}\delta^{[l]}$$
$$b^{[l-1]} = b^{[l-1]} - \eta\delta^{[l]},$$

where $\eta$ is the learning step.

After every epoch, the processes FF and back-propagation are performed forwards predicting the target and backward updating the weights and bias. The mini-batch trick is also applied to reduce the cost of the computations.

### 2.2.3 The vanishing or exploding gradient problems

As discussed before, the back-propagation algorithm goes from the output layer to the input layer, calculating the cost gradient and updating weights and biases by the gradient descent steps.

Unfortunately, these gradients may vanish or explode [3][p. 369] somehow. Typically, it gets smaller and smaller as its progress throughout lower layers, resulting in not updating the weights and biases and never converging to a good solution. Conversely, the gradient can also grow bigger and bigger, diverging the weights and biases updates. These are investigated as the vanishing and exploding gradient problems [3][p. 369], respectively.

The researchers tackle these problems by applying different activation functions for hidden-layers (Sigmoid, Tanh, ReLu, etc.) or initializing the weights and biases with several methods (Gaussian distribution, Xavier method, etc.). We will cover these topics in the next sub-sections.

### 2.2.4 Sigmoid

Sigmoid is a mathematical function with a shape of an "S"-curve widely used in logistic regression, NN and DNN because returns values between 0 and 1, such that:

$$f(z) = \frac{1}{1 + e^{-z}} = \frac{e^z}{e^z + 1}$$

The derivative of the Sigmoid is:

$$f^{'}(z) = f(z)(1 - f(z))$$

This function replaced the originally step function of the MLP, because the latter contains only flat segments in which the gradient technique does not work [3][p.351]. The former, Sigmoid, has a well-defined nonzero derivative everywhere, allowing the Gradient descent to work correctly in every step. Still, it may have problems because of its saturation when results are near zero, vanishing the gradient descent step. Therefore, another technique was introduced, hyperbolic Tanh, trying to avoid that.

### 2.2.5 Hyperbolic tanh

Like the Sigmoid, the hyperbolic tangent function tanh is S-shaped, continuous, and differentiable. This method was implemented to avoid vanishing the gradient descent step due to its range between [-1, 1]. Besides, the hyperbolic Tanh helps to speed up convergence because of its centered values on 0 [3][p. 351].

The mathematical representation:

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Its derivative:

$$f^{'}(z) = 1 - f(z)^2$$

### 2.2.6 Rectified Linear Unit (ReLu)

ReLu is a continuous and differentiable function, except when z=0, when the slope changes abruptly, and the gradient descent bounce around [3][p. 351]. Nevertheless, the function works excellently as an activation function and has faster computations than Sigmoid or Tanh. Besides, it does not have a maximum output value helping overcoming problems with the Gradient descent.

The mathematical representation:

$$f(z) = \begin{cases} 0, & z < 0 \\ z, & \text{otherwise} \end{cases}$$

### 2.2.7 Softmax

Softmax function returns the estimated probability of the corresponding prediction class. It is a generalized approach to support multiple class predictions without training and combining various binary classifiers.

The mathematical representation:

$$\sigma(z)_i = \frac{e_i^z}{\sum_{j=1}^{K} e^{z_j}},$$

for $i = 1, ..., K$ and $z = (z_1, ..., z_K) \in R^K$.

### 2.2.8 Xavier and He weights initialization method

According to [6] paper, the authors verified and discussed the variance of the output layer on Neural Networks. They noticed that the output layer typically has a higher variance than the input layer has, and it might be a problem.

Thus, for the Feed-forward and back-propagation processes flow correctly, it is necessary the same variance for the input, output layers as well as for the gradients [3][p. 371].

For that, it was proposed to initialize the parameters of the model using a specific strategy of each type of activation function, as follows:

*Table 11-1. Initialization parameters for each type of activation function*

| Activation function | Uniform distribution [–r, r] | Normal distribution |
|---|---|---|
| Logistic | $r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$ | $\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$ |
| Hyperbolic tangent | $r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$ | $\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$ |
| ReLU (and its variants) | $r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$ | $\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$ |

Figure 6: Types of Xavier initialization. Picture extracted from [3][p. 371]

## 2.3 Logistic Regression

Logistic regression models the probability that input data ($\mathbf{x}$) leads to outcomes ($y_i$), for $i = 1, 2, ..., m$, where $m$ is the number of possible labels for the outcome classes and $y_i \in [0, 1]$. By using the Sigmoid function, it expresses these probabilities of two outcomes labels, i.e. 0 and 1, as follows:

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

where $\beta$ are the coefficients that will be estimated by the model.

We define $D = (x_i, y_i)$ and assume that the samples are independent identically distributed (i.i.d.). The total likelihood for all possible outcomes in $D$ is approximated by the product of its individual probabilities for a particular outcome $y_i$:

$$P(D|\beta) = \prod_{i=1}^{n} \left[p(y_i = 1|x_i, \beta]^{y_i} \left[1 - p(y_i = 1|x_i, \beta)\right]^{1-y_i}\right.$$

From the Maximum Likelihood Estimation principle, the log of the above equation is taken, leading to the log-likelihood below:

$$\log P(D|\beta) = \sum_{i=1}^{n} \left[y_i \log p(y_i = 1|x_i, \beta) + (1 - y_i) \log(1 - p(y_i = 1|x_i, \beta))\right]$$

By reordering the expression above and multiplying by -1, we end up with the cross-entropy expression:

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

which is the cost-function for the logistic regression.

Next, it is minimized the cross-entropy which is the same as maximizing the log-likelihood:

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

and the second derivative:

$$\frac{\partial^2 C(\beta)}{\partial \beta \partial \beta^T} = \sum_{i=1}^{n} x_i x_i^T p(y_i = 1|x_i, \beta)(1 - p(y_i = 1|x_i, \beta)).$$

Hence, the first and second derivative matrices form, also known as Jacobian and Hessian matrices form, are:

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

where W is the diagonal matrix with $p(y_i = 1|x_i, \beta)(1 - p(y_i = 1|x_i, \beta)$, X is the design matrix, y is the target vector containing $y_i$ values and $p$ is the vector of probabilities.

# 3 Discussion

## 3.1 Part A

### 3.1.1 Ordinary Least Squares, Ridge regression and Stochastic Gradient Descent on terrain's heights GeoTIF image-data

We start part A of this project with a re-capitulation of Project 1, part F and G, where a GeoTIF terrain image, containing the heights of a region near Stavanger/Norway, was extracted and examined on Ordinary Least Squares (OLS) and Ridge regression models. On that occasion, we generated two sets of explanatory variables ($\boldsymbol{x1}$, $\boldsymbol{x2}$), with values between 0 and 1, and utilized them on the design matrix ($\boldsymbol{X}$) function that performed a polynomial transformation with a degree equals 10 for OLS and 11 for Ridge. The response variable set ($\boldsymbol{z}$) had space (number of samples or size of the sliced image) of 15 (15x15 pixels). Lastly, the testing MSE (mean squared error) scores obtained were 1.11 and 1.44 for OLS and Ridge, respectively.

Now, we perform a Stochastic Gradient Descent (SGD) utilizing the same generated data as above. It is essential to say that we do not scale the data-sets because of possible crashes during the gradient computations. The gradient_descent.py under the Project2/package/ directory has the algorithms for GDM and Mini-SGDM.

Our first experiment for SGD is a combination of parameters incorporated at the Project2/package/ directory, file name studies.py and class name *SearchParametersMiniSGDM*. We run the *SearchParametersMiniSGDM* class to find the best constant learning rate $\eta = [0.02, 0.01, 0.005, 0.001, 0.0005]$ as a function of the number of interactions epochs=[100, 500, 1000, 5000, 10000]. The model applied is the MiniSGDM, which is an SGD when the number of batches is equal to 1, gamma=0, lambda=0, and decay=0.

The results are as follows:

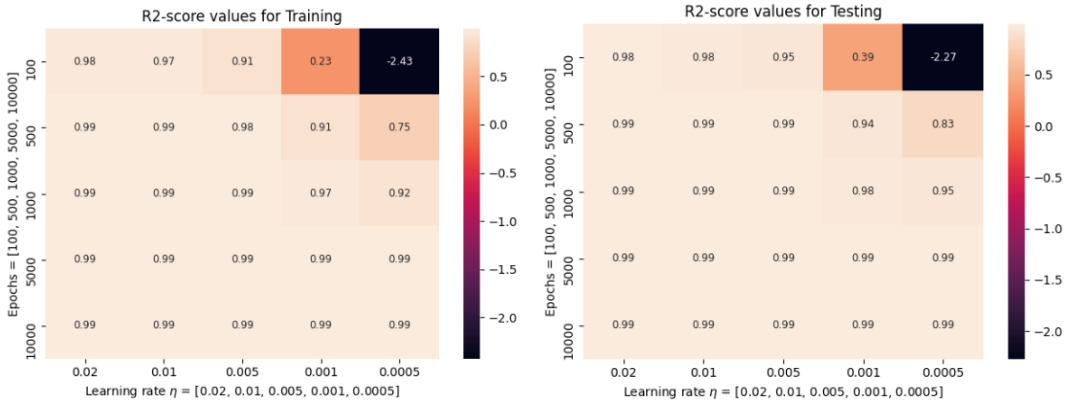



Figure 7: Heat-map containing the R2-score training and testing results of a Grid-Search with Learning rates = [0.02, 0.01, 0.005, 0.0001, 0.0005] and epochs = [100, 500, 1000, 5000, 10000].

As one can see, the design matrix ($\boldsymbol{X}$) with degree equals to 10 explains the response variable ($\boldsymbol{z}$) at a rate of 99 percent, according to the r2-scores. In other words, these results suggest that the input polynomials translate almost entirely the output heights of the terrain region studied.

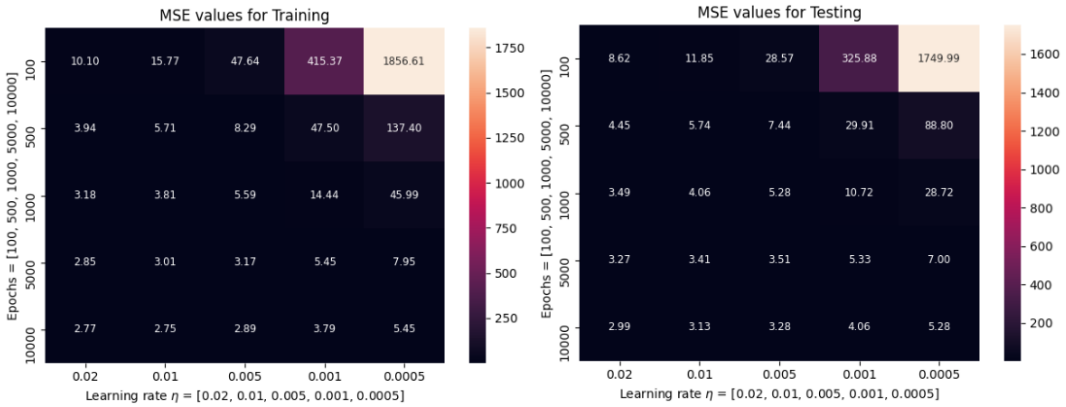



Figure 8: Heat-map containing the MSE training and testing results of a Grid-Search with Learning rates = [0.02, 0.01, 0.005, 0.0001, 0.0005] and epochs = [100, 500, 1000, 5000, 10000].

The results from training and testing MSE heat-maps indicate as the best scores are 2.77 and 2.99 for learning rates 0.02 and epochs 10.000. Notice that as the number of epochs increases, the MSE scores decreases indicating that the gradient of the loss-function is in the direction of the minimum local/global as expected.

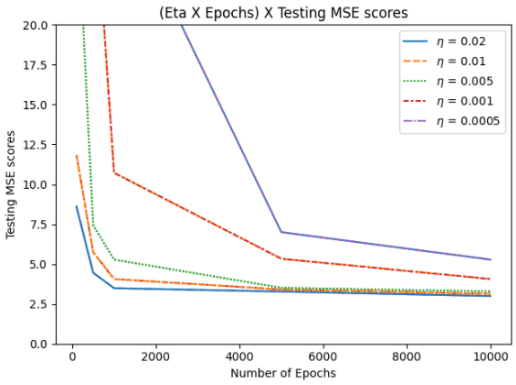



Figure 9: Line-plot showing (Eta x Epochs) X Testing MSE scores.

Line-plots above evince that the accuracy for the learning rates converges to its best score near 1.000 epochs. Therefore, we choose the number of epochs equals 1.000 as the parameter's benchmarks on futures experiments.

### 3.1.2 Tuning computation's cost (mini-batches)

The SGD method with a unique mini-batch is costly for computations while fitting the model. The algorithm stochastically chooses a single vector of features from the sample space, calculates the gradient, and updates the weights. This process is repeated for as many samples in the design matrix ($\boldsymbol{X}$) for every epoch. It turns out that the number of computations sky rocks and the entire fitting takes a long time to complete, as shown on the heat-map below:

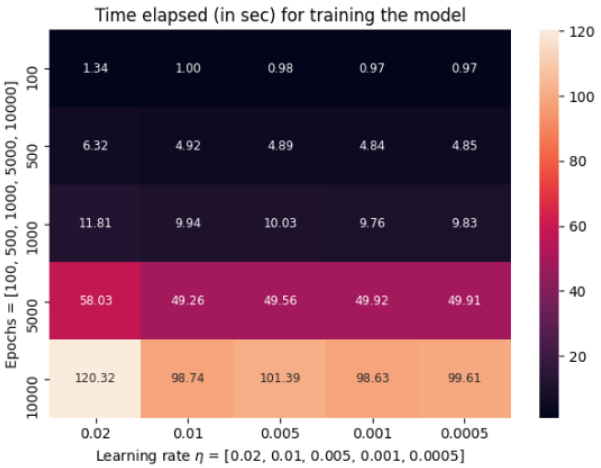



Figure 10: Heat-map containing the fitting time (in seconds) of the previous Grid-Search on Fig.1 and Fig.2.

It is evident that as the number of epochs is high, the computations' costs are higher. For example, our model's fitting time takes approx. 1 seconds to run when the epoch number is 100. On the other hand, when the epoch is 10.000, the fitting time takes approx. 100 seconds.

To avoid the cost of this expensive computation, we introduce the mini-batches technique. Instead of stochastically choosing only one vector of features from the training sample, the algorithm chooses feature vectors blocks.

Therefore, we perform a *SearchParametersMiniSGDM* for the parameters etas=[0.01, 0.005, 0.001, 0.0005] and batch_sizes=[1, 5, 10, 15, 20], using our bench-marked epoch value equals to 1.000, and analyse the testing MSE performance and fitting's time on the heat-maps below:

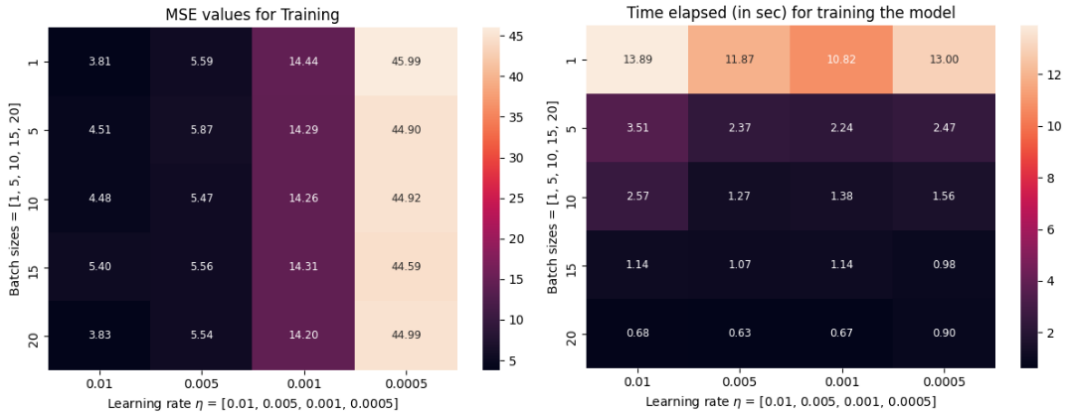



Figure 11: Left heat-map shows the testing MSE scores for each search. Right heat-map shows the fitting's time for each search.

For a single vector of features (batch_size=1), the elapsed fitting's times are above 10 seconds. Under blocks of feature vectors (mini-batches), the elapsed fitting times have a tremendous reduction without losing much the accuracy. For example, the case with learning rate=0.01 and batch-size=1 got MSE equals 3.81, and fitting's time equals 13.89 seconds, however for the same learning rate=0.01, but batch_size=20, the fitting's time was approx. 20 times faster and with almost the same accuracy of 3.83.

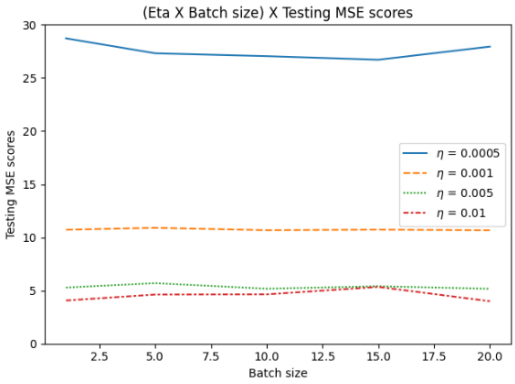



Figure 12: Line.plot containing (Eta x Batch size) X testing MSE.

Fig.11 shows that the constant learning rate $\eta$=0.01 performs better than the other learning rates. It has constant accuracy for all batch sizes tested. Besides, we see that batch=10 is the sweet spot for $\eta$=0.01 because it is where the accuracy starts to worsen.

Hence, we define our bench-marked parameters epochs=1.000 and batch_size=10 for futures examinations.

### 3.1.3 Learning rate decay and tuning decay

There are many ways to optimize the learning rate to find more accurate predictions. As studied in Gradient Descent theory, the learning rate ($\eta$) is a hyper-parameter that manages the model's change in response to the MSE each time the model's coefficients (weights) are updated.

Now, we apply a technique that reduces the learning rate value according to the number of epochs, studied in Learning rate decay. To do that, we introduce a constant parameter called decay.

Thus, we run a *SearchParametersMiniSGDM* with the parameters etas=[0.35, 0.3, 0.25, 0.2, 0.01] and decays=[10**-1, 10**-2, 10**-3, 10**-4, 10**-5], using our bench-marked epoch=1.000 and batch-size=10, to find the best testing MSE score for an eta as a function of a decay value:

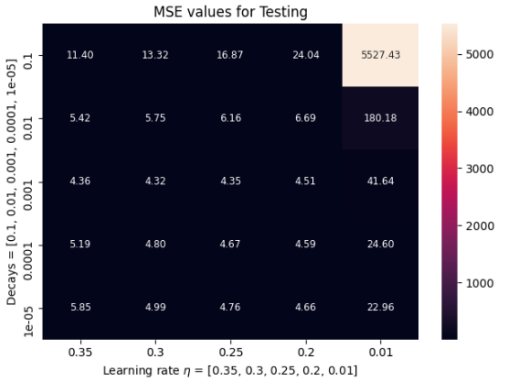


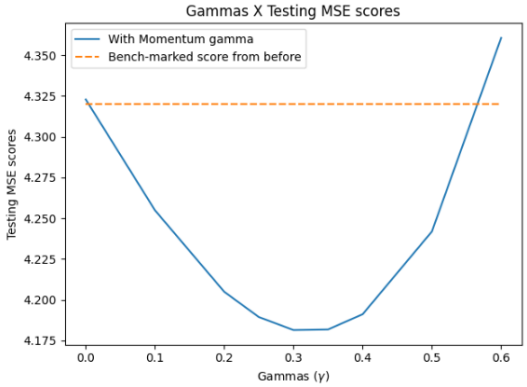

Figure 13: Heat-map showing the testing MSE scores for learning rates as function of decays.

With this decay technique, we were able to increase the learning rate without breaking the gradient calculation. Therefore, the best testing MSE achieved was 4.32 for learning rate=0.3 and decay=0.001.

Hence, we include decay=0.001 and eta0=0.3 to our bench-marked parameters epochs=1.000, batch_size=10. We will try to beat the testing MSE of 4.32 obtained in this section by introducing mini-batch SGD momentum (gamma) and 'l2' regularization in the following sub-sections.

### 3.1.4 SGD with momentum and tuning gamma

This subsection will try to overcome the testing MSE of 4.32 by applying another type of Mini-batch Stochastic Gradient Descent, so-called Mini-batch SGD with momentum. A new parameter gamma is introduced, as explained in the theory section in "BGD, SGD and Mini-batch SGD with momentum". We use different values of gamma=[0, 0.1, 0.2, 0.25, 0.3, 0.35, 0.4, 0.5, 0.6] and run the model MiniSGDM with the bench-marked parameters obtained from the previous subsections, decay=0.001, eta0=0.3, epochs=1.000 and batch_size=10.




```
Gamma: 0   |   MSE test: 4.322867060388854
Gamma: 0.1   |   MSE test: 4.254854716954629
Gamma: 0.2   |   MSE test: 4.20487493800941
Gamma: 0.25   |   MSE test: 4.189362240428546
Gamma: 0.3   |   MSE test: 4.1814057511429015
Gamma: 0.35   |   MSE test: 4.181753837532903
Gamma: 0.4   |   MSE test: 4.191106541708218
Gamma: 0.5   |   MSE test: 4.241878011653097
Gamma: 0.6   |   MSE test: 4.360691173481241
```

Figure 14: Left line-plot containing the MSE values as function of gammas. Right output of accuracy values for gammas.

As shown, the Mini-batch Stochastic Gradient Descent with momentum overcome our bench-marked score of 4.32, reaching 4.18 when gamma parameter is 0.3.

Hence, we include gamma=0.3 to our bench-marked parameters decay=0.001, eta0=0.3, epochs=1.000, batch_size=10. We will try to beat the testing MSE of 4.1814 obtained in this section by introducing 'l2' regularization (Ridge lambda) in the next sub-section.

### 3.1.5 'L2' regularization and tuning lambda

This subsection will try to overcome the testing MSE of 3.65 by applying the l2 regularization (Ridge). A new parameter lambda is introduced, as explained in the theory section of Project 1. We use different values of lambda=[0, 10**-9, 10**-6, 10**-4] and run the model MiniSGDM with the bench-marked parameters obtained from the previous subsections, gamma=0.3, decay=0.001, eta0=0.3, epochs=1.000 and batch_size=10.

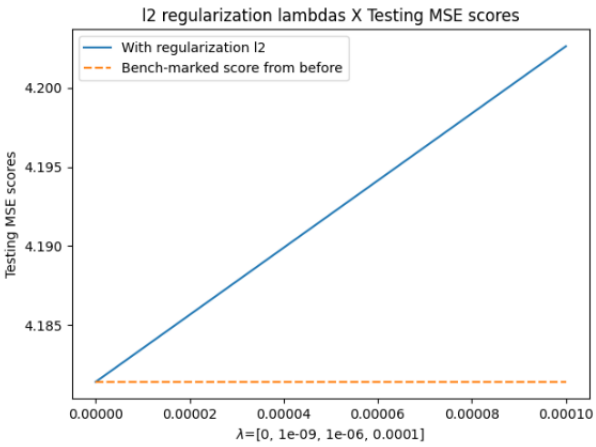



Figure 15: Line-plot containing the MSE values as function of lambdas.

As exhibited, the 'l2' regularization did not make any improvement overcoming our bench-marked testing accuracy of 4.1814.

## 3.2 Part B and C

In part B and C of this report, we will discuss the application of our own Feed Forward Deep Neural Network (DNN) with the back-propagation method on a regression problem. The data-set applied on DNN will be the GeoTIF terrain image utilized in part A. It is essential to mention that the descriptive data-set will not be submitted to a design matrix function, as we did in part A, because the DNN should identify possible linearity by itself. Also, the targets will not be scaled, as we did not do that on "Project 1" and part A. These choices are because of comparisons propose as the performances between OLS, Ridge, Mini-batch SGDM and DNN will be distinguished, and we want to feed the models with almost the same complexities as fed before.

The cost-function employed is the Mean Squared Error (MSE) due to the regression nature of this kind of problem. Also, as explained on the DNN theory topic, the MSE function matches very well on models that use gradient descent because of MSE's convexity and continuity aspects.

Moreover, the activation function for the hidden layers will be Sigmoid on the first experiments and then Tanh and Relu. The output layer's activation function will be "identity" because DNN for regression cases does not use an activation function for output values.

The way that the models' weights and biases are initialized is through Gaussian distribution, but another method so-called "Xavier method" will be explored. Both approaches are explained in detail in the theory section.

Finally, the number of epochs tested in this section will be fixed at 500 because of computation costs. The batch_size will also be fixed at the samples' length in training data due to the best metrics than mini-batches.

### 3.2.1 Number of Neurons Vs Learning rates for 1 hidden layer Sigmoid

At the beginning of part B of this report, we will employ a one-layer neural network to analyze different vital topics regarding the number of neurons, learning rate, epoch, cost error, decay, and 'l2' regularization.

The Python function called one_hidden_layer_sigmoid at file partBandC.py in the directory Project2/ will perform the following experiments and return the results we will discuss here.

First, we would like to find the best number of neurons as a function of a learning rate ($\eta$) value. For that, n_neurons = [10, 20, 30, 40, 50, 75, 100, 125, 150] and etas = [0.99, 0.9, 0.8, 0.7, 0.6, 0.5] are defined and performed on our class *SearchParametersDNN* in studies.py at directory package/. This class method fits our Multi-layer perceptron (MLP) function with a combination of the previous parameters, one by one, to identify which one returns the best metrics as follows:

```
Best testing MSE is 10.085788258400251 at indexes (1, 6), with parameters:
n_neurons: 100
Learning rate (eta): 0.9
```

Figure 16: Output for printing best combination of metrics.

This first output shows that the best combination of parameters is neurons = 100 and learning rate = 0.9, which obtained testing MSE at 10.0857.


Figure 17: Heat-maps containing the R2-score training and testing results.

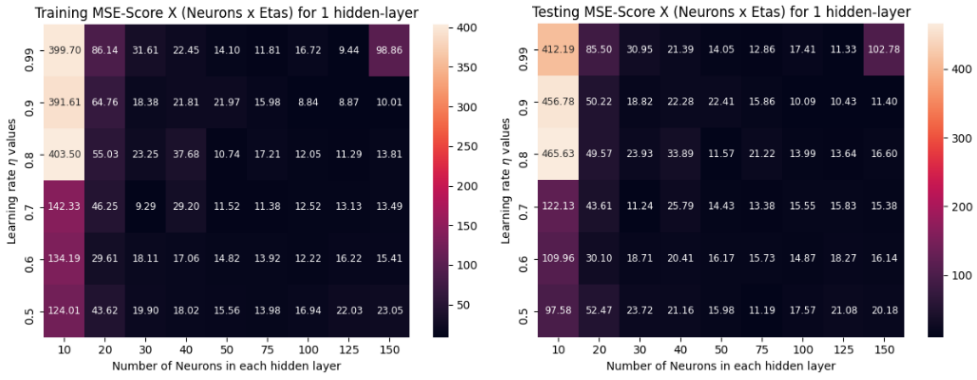



Figure 18: Heat-maps containing the MSE-score training and testing results.

The results are promising because the model reached r2-score around 99 percent for only one hidden-layer. The best training and testing MSE accuracies were 8.84 and 10.09, respectively, at neurons = 100 and $\eta = 0.9$. Observe that when the neurons and learning rates are smaller or larger than 100 and 0.9, the model gets worse scores, indicating that the parameters are optimized.

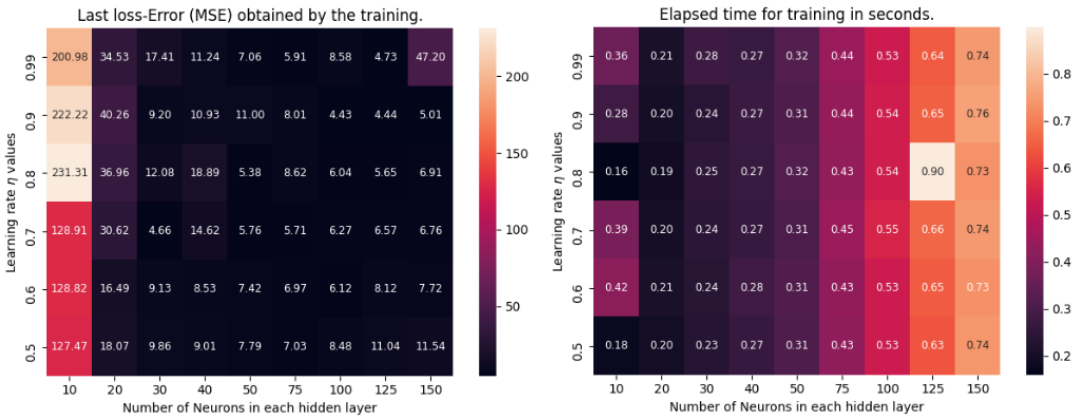



Figure 19: Heat-maps containing the loss-error for the last training epoch (left) and elapsed training times (right).

The loss-error is the MSE score measured for every Feed-Forward process for every epoch during training. As shown above, the best loss-error for the last training epoch at neurons = 100 and $\eta$=0.9 likewise was the smallest/best with a 4.43 score. This score is overestimated because the training and testing MSE scores were higher in Figure 17. Besides, as the number of neurons is high, the time elapsed for training is more considerable.

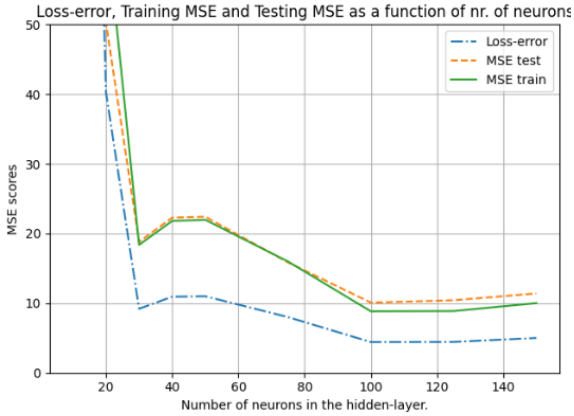



Figure 20: Line-plot containing the loss-error, training and testing MSE scores for $\eta$=0.9 and different neurons values.

The previous figure shows that our parameter's sweet spot is 100 for the number of neurons. This parameter is optimal because it reached the best score in the searching and where the bias and variance of the model are the lowest since loss-error, training and testing lines are in the smallest score and closest among each other. Besides, an increase in the variance happens when the complexity number of neurons increases above 100, where the scores worsen, and the difference among loss-error, training and testing increases.

### 3.2.2 Accuracy analysis and epochs analysis

Considering our best model from the previous sub-topic, we can analyze each epoch's accuracy score's evolution by the succeeding line-plots:

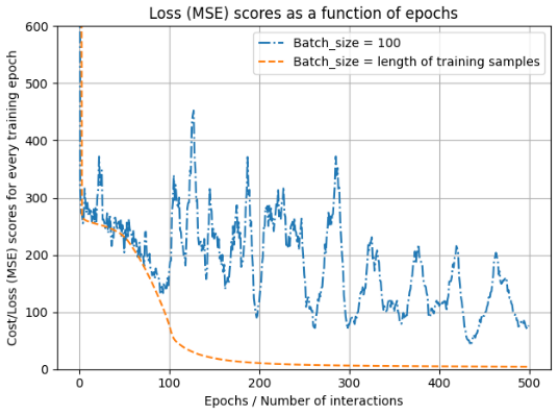



Figure 21: Line-plot containing the cost/loss-error for every training epoch.

At the beginning of the fitting, the accuracy is terrible because of the randomness initiation of the layers' weights and biases. As long as the epochs are processed, our gradient descent does an excellent task, converging and decreasing to the local or global minimum score.

It is imperative to say that we were using the parameter batch_size equals the number of samples of the training data. This parameter makes the cost curve above being smooth due to the batch gradient descent essence, as explained in theory.

However, when we define batch_size equals 100, less than the number of samples of the training data, then the cost curve will differ, being choppy due to the Stochastically nature of the mini-batches choices. Nonetheless, the gradient descent still performs very well, converging the cost to its minimum as expected. This option of batch_size gains a lot in speedy of the computation, likewise explained in theory section", and it is ideal for extensive explanatory data with a considerable number of samples and features.

### 3.2.3 Tuning the learning rate, decay and "l2" regularization

We will tune the parameters learning rate ($\eta$), decay and 'l2' Ridge regularization lambda. Remember that the explanations about each parameter are in the theory section. For this optimization, we choose different values for etas = [0.99, 0.95, 0.9, 0.85, 0.8, 0.7, 0.6, 0.5], decays = [0, 10 ** -9, 10 ** -8, 10 ** -7, 10 ** -6, 10 ** -5, 10 ** -4, 10 ** -3, 10 ** -2, 10 ** -1] and lambdas = [0, 10 ** -9, 10 ** -8, 10 ** -7, 10 ** -6, 10 ** -5, 10 ** -4, 10 ** -3, 10 ** -2, 10 ** -1], and then perform the MLP model for each combination of parameters. The printed output shows the best parameters and testing MSE score for the best combination:

```
Best testing MSE is 10.085788258400251 at indexes (0, 0, 2), with parameters:
Eta: 0.9
Decay: 0
Lambda: 0
```

Figure 22: Printed output containing the best parameters of the searching.

This output confirms that the learning rate equal to 0.9 acquired before is optimized. Besides, the decay and lambda parameters did not help the MLP model overcome the testing MSE score of 10.085.

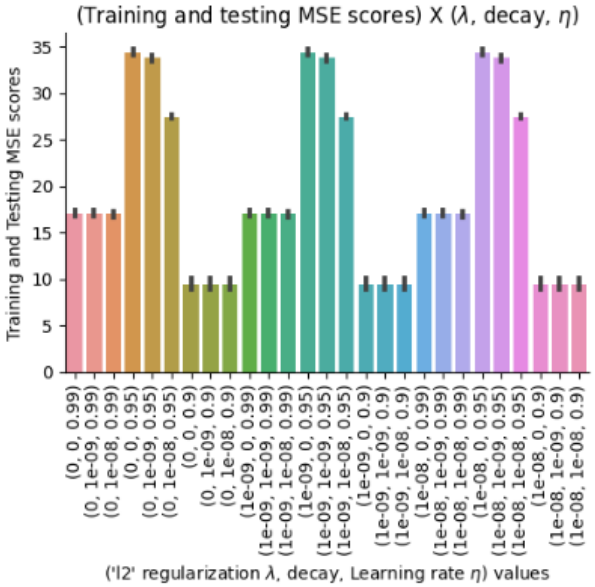



Figure 23: Bar-plot containing selected best scores for the $\eta$, decay and $\lambda$ parameter search. The upper extreme of the black bars is the testing MSE scores, and the lower extreme of the same black bars is the training MSE scores. The colored bars are the average between training and testing MSE scores.

The bar-plotting above shows that the 'l2' regularization and decay did not produce any furtherance in the best score. Although, parameters ($\lambda$=1e-9, decay=1e-9, $\eta$=0.9) and ($\lambda$=1e-8, decay=1e-8, $\eta$=0.9) achieved pretty close scores to ($\lambda$=0, decay=0, $\eta$=0.9). Since $\lambda$ attenuates the variance and avoids over-estimations and over-fitting of the model, those sets of parameters with $\lambda$ different from 0 might be better choice.

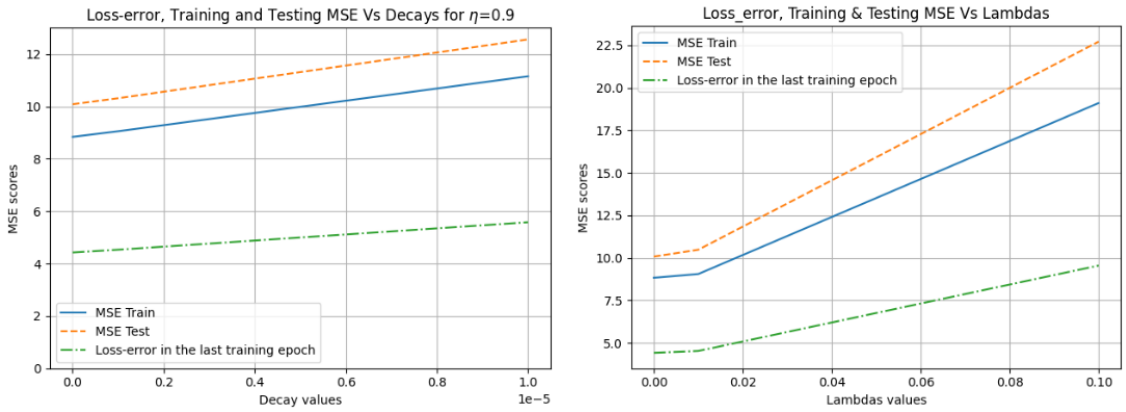



Figure 24: Line-plots containing loss-error, training and testing MSE as a function of decays (left), loss-error, training and testing MSE as a function of lambdas (right).

The line-plots above confirm that the MSE scores did not improve by applying decay and lambda parameters.

### 3.2.4 Terrain image data on 2 hidden-layers MLP-DNN

In this sub-section, we will continue with the same data-set of GeoTIF and analyze it on an MLP with two hidden layers. The aim here is to examine if we can achieve better accuracy scores with a higher number of hidden layers.

The standard parameters will still be epochs=500, batch_size = len(X_train), learning_rate = 'constant', decays = 0.0, lmbds = 0.0, bias0 = 0.01, init_weights = 'normal', act_function = 'sigmoid', output_act_function = 'identity', cost_function = 'mse', random_state = 10.

First, a combination of parameters between the number of neurons equal to [5, 10, 20, 50, 100, 150, 200, 300, 400, 500] per hidden-layer and learning rates equal to [0.1, 0.09, 0.08, 0.05] is performed with results as follows:

```
Best testing MSE is 9.37034822222356 at indexes (2, 6), with parameters:
n_neurons: 500
Learning rate (eta): 0.08
```

Figure 25: Output containing the best testing MSE score and its parameters.

The output shows that two hidden layers MLP model produces its best testing MSE score when each hidden layer has 500 neurons and eta=0.08. Catch sight on the testing MSE score obtained here (9.37) is lower and better than the score attained for one hidden layer (10.80) with 100 neurons and eta=0.9, printed in the previous section's output.

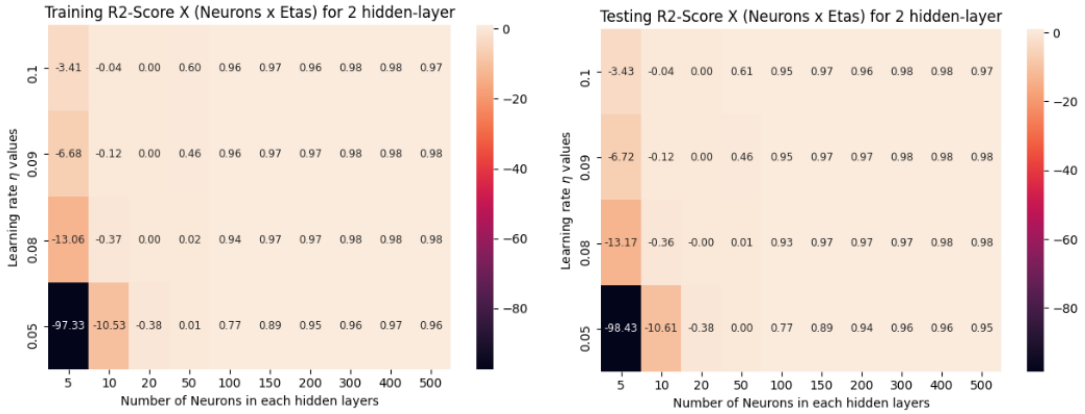



Figure 26: Heat-plots containing the training (left) and testing (right) R2-scores.

The R2-scores obtained for two hidden layers are very close to the ones attained from one hidden layer, but the number of neurons needs to be higher for the former in order to achieve the same accuracy of 0.98 from the latter.

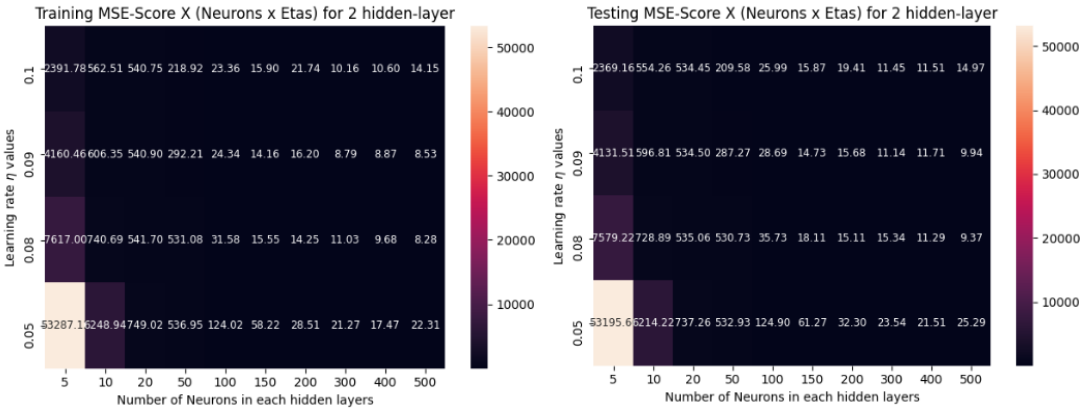



Figure 27: Heat-plots containing the training (left) and testing (right) MSE-scores.

The best training and testing MSE scores of 8.28 and 9.37 for two hidden layers are better than 8.84 and 10.09 for one hidden layer. This lower MSE values might owing to the fact that two hidden layers provide more significant data and complexity to the learning process, thus more accurate results. However, one must be aware that this improvement can be over-estimated or over-fitted, demanding more attention.

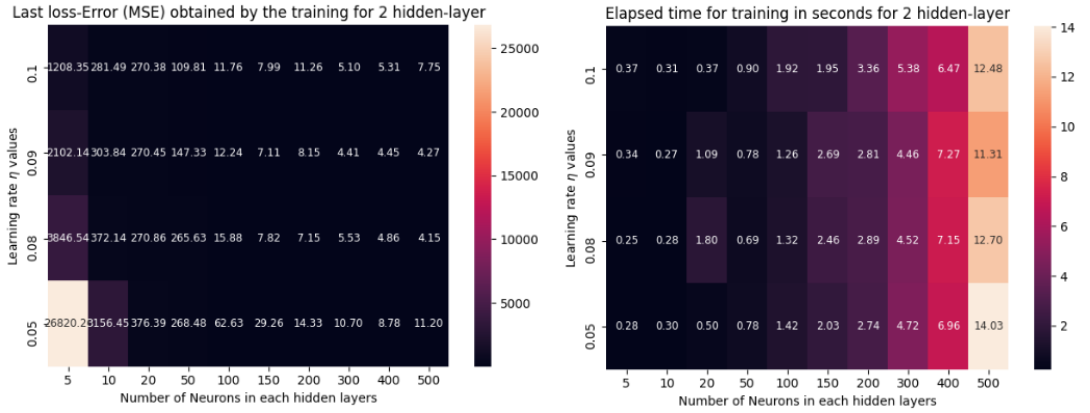



Figure 28: Heat-plots containing the last training loss-cost (left) and the time elapsed for training (right).

The best last training loss-error of 4.15 for two hidden layers are better than 4.43 for one hidden layer, confirming that two hidden layers perform better than one hidden layer. However, the elapsed time for training the best two hidden layers was 12.70 seconds, much higher than 0.54 seconds needed for one hidden layer.

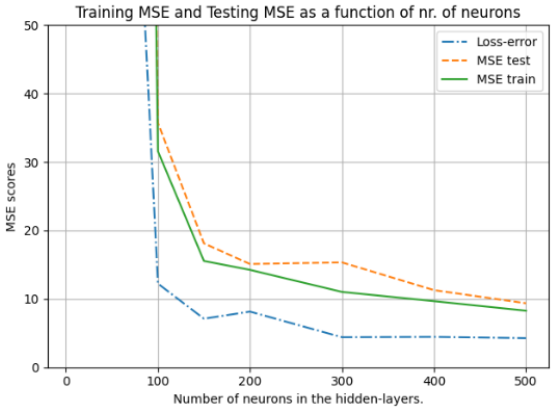



Figure 29: Line-plots containing the last training loss-cost, training and testing MSE scores for 100 neurons in each hidden layers and eta=0.08.

The main difference observed in the line-plots between two hidden layers, and one hidden layer is that the variance has not impacted the former, and the accuracy metric is still going under by 500 neurons. The latter has reached the lower metric score at 100 neurons, and after that, the values started to increase, showing the variance has come.


Figure 30: Line-plots containing the cost/loss for every epoch.

In our case, one can observe that the mini-batch strategy batch_size=100, for two hidden layers, produces fewer chop bounces than for one hidden layer. However, the former takes a long time to converge to its minimum, around 300-400 epochs, while the latter takes 100-200 epochs.

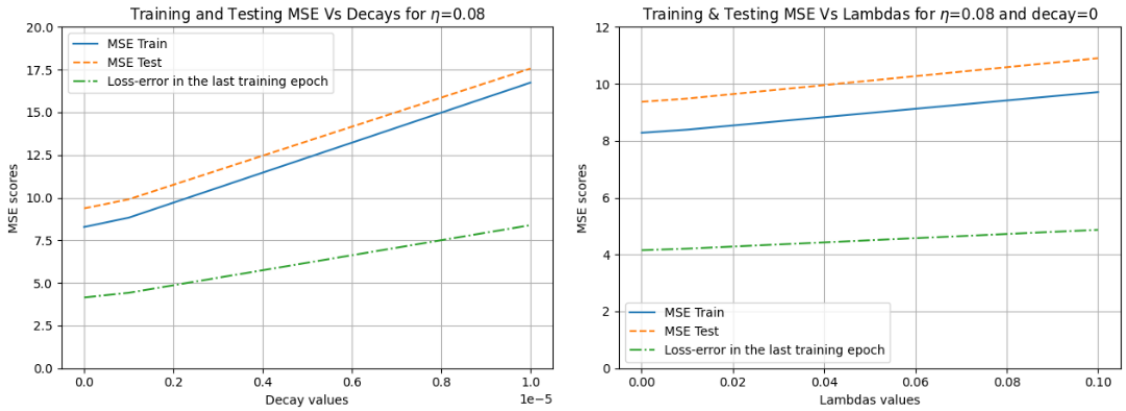



Figure 31: Line-plots containing learning rate = 0.08 as a function of decays = [0, 10 ** -9, 10 ** -8, 10 ** -7, 10 ** -6, 10 ** -5] (left) and lambdas = [0, 10 ** -5, 10 ** -4, 10 ** -3, 10 ** -2, 10 ** -1] (right).

Again, as happened with our one hidden layer MLP model, the parameters decay and lambda does not improve the model's accuracy score.

Finally, we perform a combination of parameters etas = [0.1, 0.09, 0.08, 0.05], decays = [0, 10 ** -9, 10 ** -8, 10 ** -7, 10 ** -6, 10 ** -5] and lambdas = [0, 10 ** -5, 10 ** -4, 10 ** -3, 10 ** -2, 10 ** -1].

```
Best testing MSE is 9.37034822222356 at indexes (0, 0, 2), with parameters:
Eta: 0.08
Decay: 0
Lambda: 0
```

Figure 32: Output containing the result of a combination of parameters $\eta$, decay and $\lambda$.

The output above confirms that the parameters decay and lambda does not improve our two hidden layer MLP model's accuracy.

### 3.2.5 Tanh activation function

Successively, we will perform the same two layer experiments done before on GeoTIF data, but now with the hyperbolic Tanh activation function for hidden layers.

The standard parameters will still be epochs=500, batch_size = len(X_train), learning_rate = 'constant', decays = 0.0, lmbds = 0.0, bias0 = 0.01, init_weights = 'normal', act_function = 'tanh', output_act_function = 'identity', cost_function = 'mse', random_state = 10.

A combination of parameters between the number of neurons equal to [5, 10, 25, 50, 100, 150, 200, 300, 400, 500] per hidden-layer and learning rates equal to [0.3, 0.1, 0.07, 0.05, 0.04, 0.03, 0.02, 0.01] is performed with results as follows:

```
Best testing MSE is 8.196480541850725 at indexes (4, 8), with parameters:
n_neurons: 400
Learning rate (eta): 0.04
```

Figure 33: Output with the best testing MSE and parameters neurons and etas.

The output shows that the hyperbolic tanh for two hidden layers got its best testing MSE score of 8.19, which is better than the best testing score for one hidden layers Sigmoid (10.08) or two hidden layer Sigmoid (9.37) experiments. This might have happened because hyperbolic tanh helps the gradient descent to not vanish, as explained in the theory section of "the vanishing or exploding gradient problems".

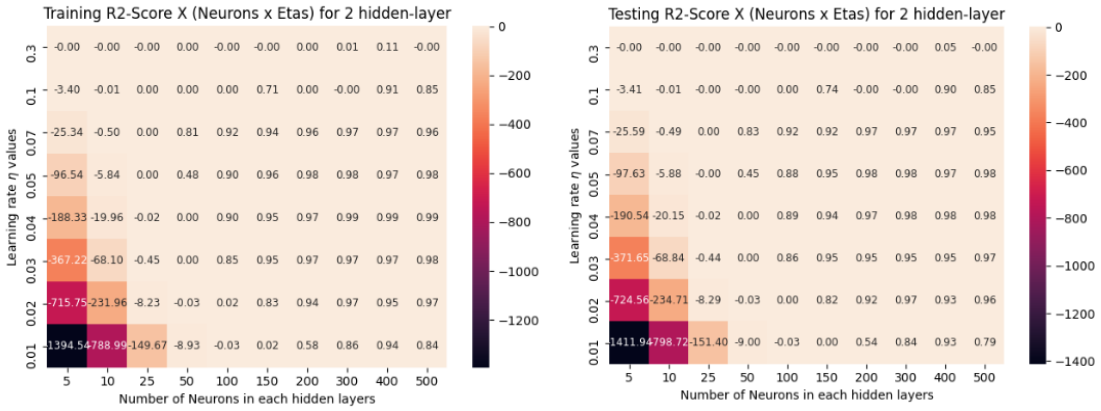



Figure 34: Heat-maps containing the training (left) and testing (right) R2-Scores.

The R2-scores continues very high at 0.98, as obtained for two hidden layers Sigmoid or one hidden layer Sigmoid, showing that the MLP model is able to identify the linearity of the training data with a very good precision.

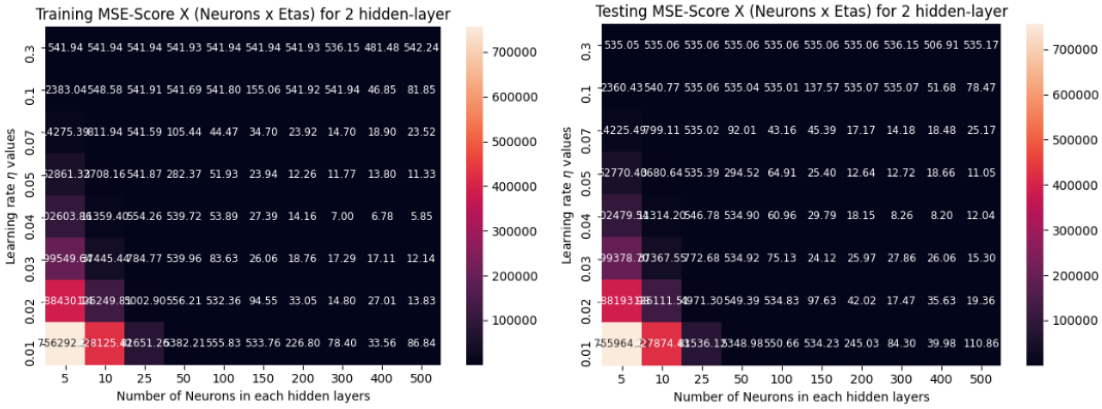



Figure 35: Heat-maps containing the training (left) and testing (right) MSE-Scores.

The best training and testing MSE scores of 6,78 and 8,19 are better than the ones achieved on two hidden layers Sigmoid (8.28 and 9.37) or one hidden layer Sigmoid (8.84 and 10.08), confirming that the Tanh activation function performed better than Sigmoid.

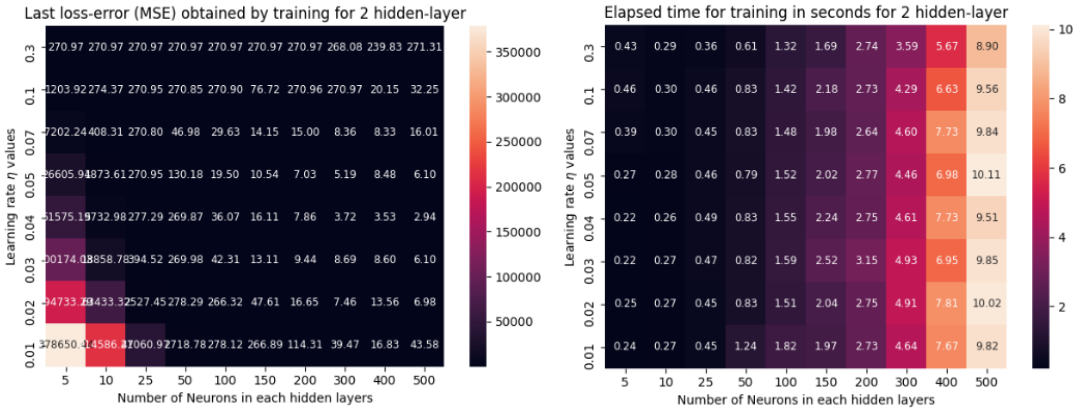



Figure 36: Heat-maps containing the loss-error at last epoch (left) and training elapsed times (right).

Also, observe that the loss-error for the last epoch during training with Tanh got (2,94) smaller scores than for two hidden layers Sigmoid (4.15) or for one hidden layer Sigmoid (4.43).

Moreover, the elapsed time for training the best two hidden layers Tanh was 7.73 seconds, faster than 12.70 seconds for two hidden layers Sigmoid, but slower than 0.54 seconds for one hidden layer Sigmoid.

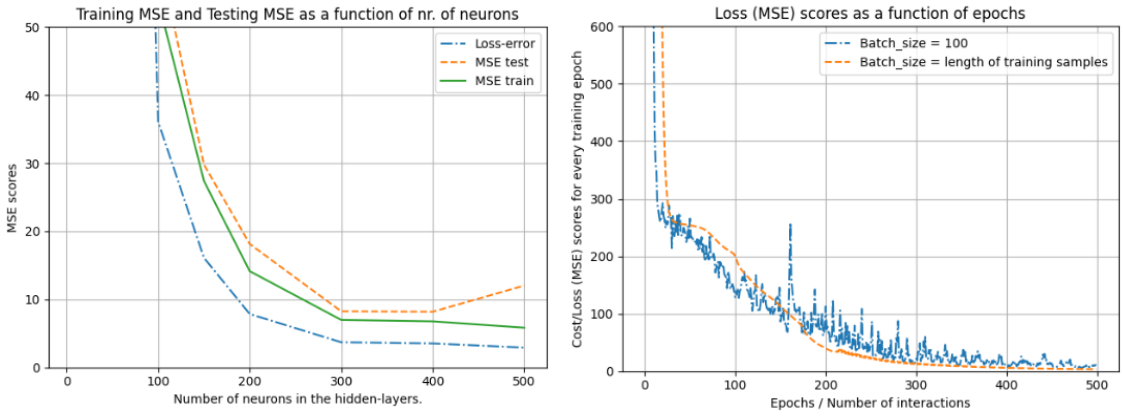



Figure 37: Left line-plots containing the last training loss-cost, training and testing MSE scores for 400 neurons in each hidden layers and eta=0.04. Right line-plots containing the cost/loss for every epoch.

Left line-plots above shows that the model reaches its minimum accuracy around 300 or 400 epochs, where we can notice an increase of the variance. Thus, be variance has come later than for one hidden layer Sigmoid (converged at, but earlier than for two hidden layers Sigmoid.

Regarding the right line-plots above, both batch and mini-batch strategies reached its minimum around 300-400 epochs with their typical behavior, a smooth decreasing line for the first and a bounced line for the second.

Now, a parameter combination between learning rate = [0.1, 0.07, 0.06, 0.05, 0.04, 0.03], decays = [0, 10 ** -9, 10 ** -8, 10 ** -7, 10 ** -6, 10 ** -5] and lambdas = [0, 10 ** -5, 10 ** -4, 10 ** -3, 10 ** -2, 10 ** -1] is performed on MLP, obtaining the following results:

```
Best testing MSE is 7.469313829690998 at indexes (0, 4, 4), with parameters:
Eta: 0.04
Decay: 1e-06
Lambda: 0
```

Figure 38: Output with the best testing MSE and parameters etas, decay and lambdas.

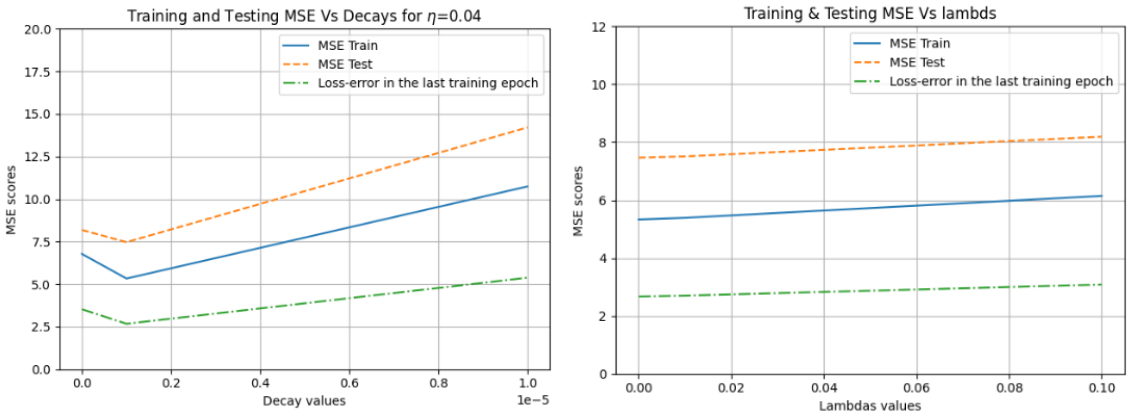



Figure 39: Line-plots containing learning rate = 0.04 as a function of decays = [0, 10 ** -9, 10 ** -8, 10 ** -7, 10 ** -6, 10 ** -5] (left) and lambdas = [0, 10 ** -5, 10 ** -4, 10 ** -3, 10 ** -2, 10 ** -1] (right).

One can see that the decay parameter got a positive effect resulting in an accuracy improvement on testing MSE from 8.19 to 7.46. However, the lambda parameter has not positively impacted the results.

### 3.2.6 ReLu activation function

In succession, we will perform the same two layer experiments done before on GeoTIF data, but now with the ReLu activation function for hidden layers.

The standard parameters will still be epochs=500, batch_size = len(X_train), learning_rate = 'constant', decays = 0.0, lmbds = 0.0, bias0 = 0.01, init_weights = 'normal', act_function = 'relu', output_act_function = 'identity', cost_function = 'mse', random_state = 10.

A combination of parameters between the number of neurons equal to [7, 8, 9, 10, 14, 15] per hidden-layer and learning rates equal to [0.001, 0.0009, 0.0008, 0.0007, 0.0006, 0.0005, 0.0004, 0.0003, 0.0002, 0.0001] is performed with results as follows:

```
Best testing MSE is 14.182477527211068 at indexes (5, 2), with parameters:
n_neurons: 9
Learning rate (eta): 0.0005
```

Figure 40: Output with the best testing MSE and parameters neurons and etas.

The output shows that the ReLu for two hidden layers got its best testing MSE score of 14.18, which is worse than the best testing score for two hidden layers Tanh (8.19) or for one hidden layers Sigmoid (10.08) or for two hidden layer Sigmoid (9.37) experiments.

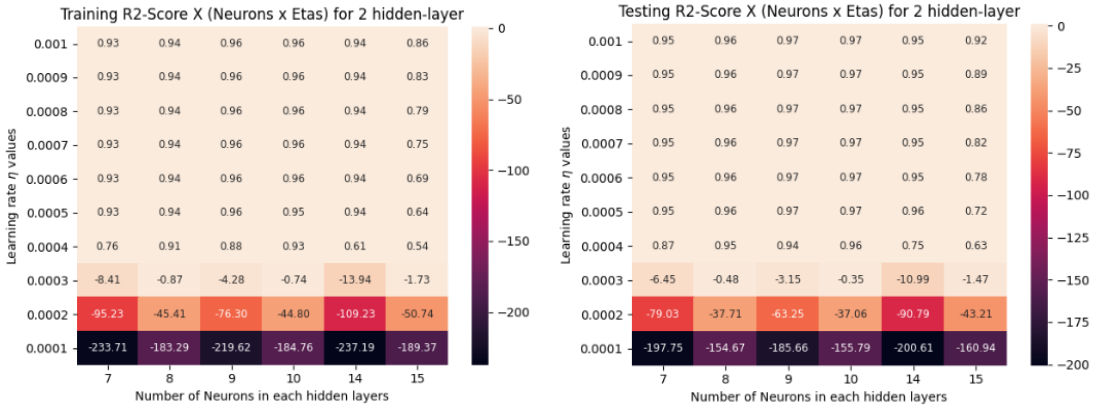



Figure 41: Heat-maps containing the training (left) and testing (right) R2-Scores.

The best R2-scores are slightly lower around 0.97 for two hidden layers ReLu, while two hidden layers Tanh, two hidden layers Sigmoid, and one hidden layer Sigmoid got 0.98.

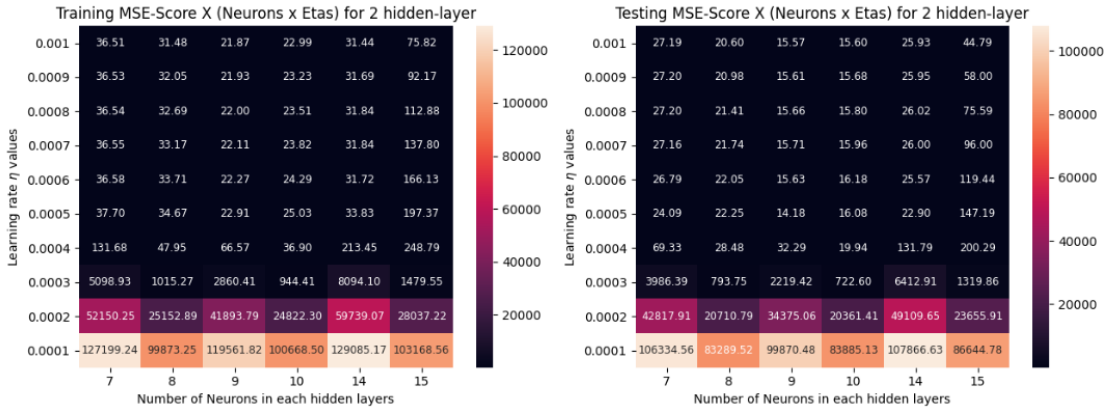



Figure 42: Heat-maps containing the training (left) and testing (right) MSE-Scores.

The best training and testing MSE scores of 21.87 and 15.57 are worse than the ones achieved on two hidden layers Tanh (6,78 and 8,26), two hidden layers Sigmoid (8.28 and 9.37), and one hidden layer Sigmoid (8.84 and 10.09).

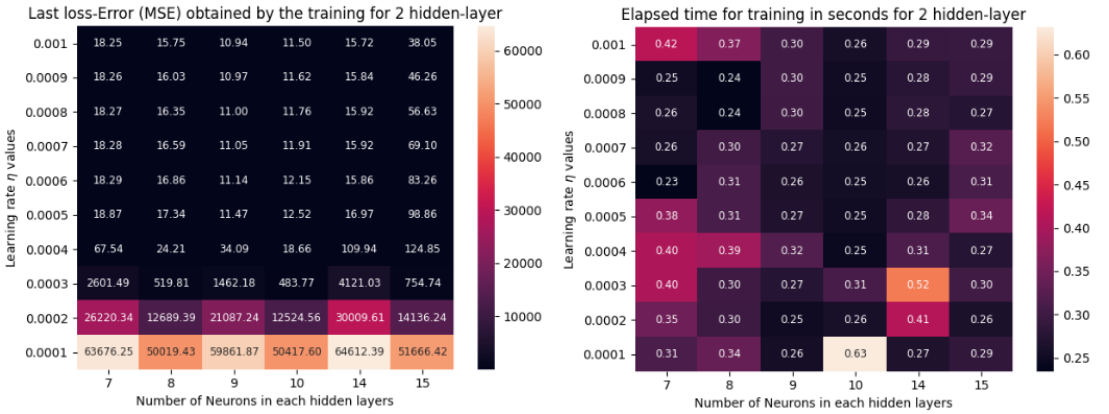



Figure 43: Heat-maps containing the loss-error at last epoch (left) and training elapsed times (right).

The loss-error for the last epoch during training with ReLu got 10.94, which is higher/worse than 2,94 for two hidden layers Tanh, 4.15 for two hidden layers Sigmoid and 4.43 for one hidden layer Sigmoid.

On the other hand, the elapsed time for training the best scores on ReLu is way faster at 0.30 seconds, comparing to 7.73 seconds for two hidden layers Tanh, 12.70 seconds for two hidden layers Sigmoid, and 0.54 seconds for one hidden layer Sigmoid.

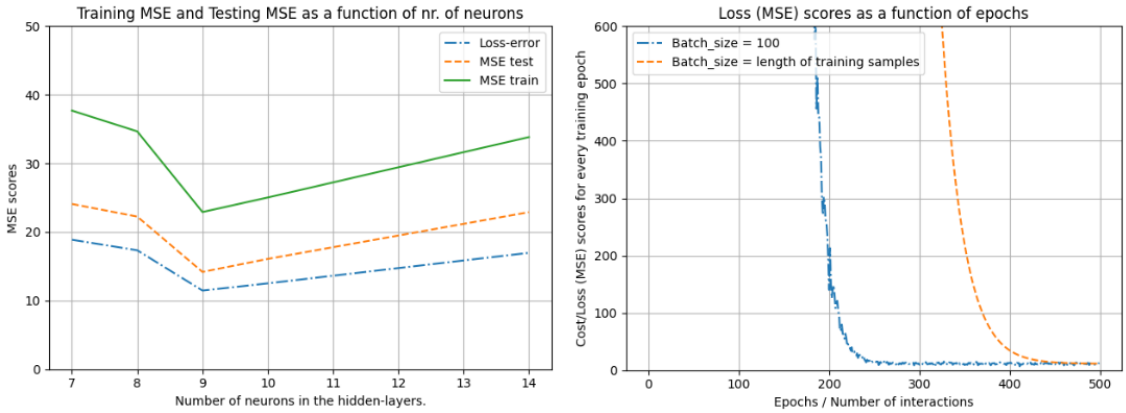



Figure 44: Left line-plots containing the last training loss-cost, training and testing MSE scores for 9 neurons in each hidden layers and eta=0.0005. Right line-plots containing the cost/loss for every epoch.

Left line-plots above shows that the model reaches its minimum accuracy with around 9 neurons per hidden layer, where we can notice an increase of the variance after that.

Right line-plots above exhibits that the model reaches its minimum training accuracy around 200 or 300 epochs, which is faster than two hidden layers Tanh, both batch and mini-batch strategies, which reached its minimum around 300-400 epochs. Still, a smooth decreasing line for the batch step and a bounced line for mini-batch step were verified.

Finally, a parameter combination between learning rate = [0.0006, 0.0005, 0.0004], decays = [0, 10 ** -8, 10 ** -7, 10 ** -6, 10 ** -5, 10 ** -4] and lambdas = [0, 0.01, 0.1, 0.5, 0.9, 0.95, 0.99] is performed on MLP, obtaining the following results:

```
Best testing MSE is 13.508104904343172 at indexes (6, 3, 1), with parameters:
Eta: 0.0005
Decay: 1e-06
Lambda: 0.99
```

Figure 45: Output with the best testing MSE and parameters etas, decay and lambdas.

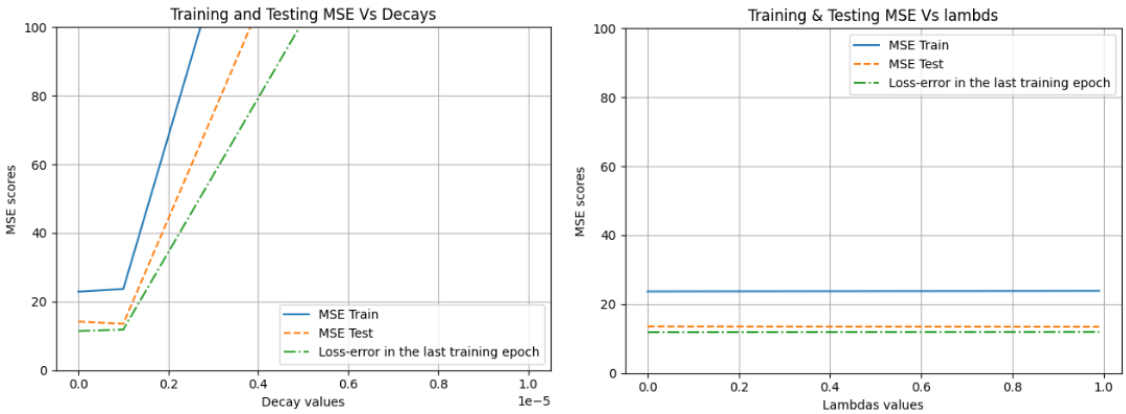



Figure 46: Line-plots containing learning rate = 0.0005 as a function of decays = [0, 10 ** -8, 10 ** -7, 10 ** -6, 10 ** -5, 10 ** -4] (left) and lambdas = [0, 0.01, 0.1, 0.5, 0.9, 0.95, 0.99] (right).

One can see that the decay and lambda parameters got a positive effect resulting in an accuracy improvement on testing MSE from 14.18 to 13.50.

### 3.2.7 Initializing weights using Xavier method

At the end of this part B and C, we will carry out the same two layer experiments done before on GeoTIF data, but now using the Xavier method for weights initialization and choosing Sigmoid, Tanh, and ReLu as the activation function.

The standard parameters will still be epochs=500, batch_size = len(X_train), bias0 = 0.01, init_weights = 'xavier', output_act_function = 'identity', cost_function = 'mse', random_state = 10.

A combination of parameters between the number of neurons equal to [7, 9, 15, 50, 75, 100, 250, 500] per hidden-layer and learning rates equal to [0.001, 0.00075, 0.0005, 0.0004] is conducted using ReLu activation function:

```
Best testing MSE is 13.692239540093444 at indexes (2, 7), with parameters:
n_neurons: 500
Learning rate (eta): 0.0005
```

Figure 47: Output with the best testing MSE and parameters neurons and etas.

Attention to the best testing MSE score of 13.69 which is better/smaller than 14.18 obtained for normal weights initialization on MLP with ReLu.

Next, a parameter combination between learning rate = [0.001, 0.00075, 0.0005, 0.0004], decays = [0, 10 ** -8, 10 ** -7, 10 ** -6, 10 ** -5] and lambdas = [0, 10 ** -2, 10 ** -1, 0.5, 0.9, 0.99] is executed on MLP with ReLu and two hidden layers of 500 neurons, coming by the following results:

```
Best testing MSE is 13.687256474037913 at indexes (2, 2, 2), with parameters:
Eta: 0.0005
Decay: 1e-07
Lambda: 0.1
```

Figure 48: Output with the best testing MSE and parameters etas, decay and lambdas.

The regularization of the learning rate ($\eta$) combined to the 'l2' regularization ($\lambda$) got a positive effect reducing the best testing MSE score from 13.69 to 13.68. This reduced score is considered

more reliable because its regularization techniques decrease the model's variance and over-fitting, giving less over-estimated metrics.

Regarding to MLP with Tanh activation function, a combination of parameters between the number of neurons equal to [5, 10, 25, 50, 100, 150, 200, 300, 400, 500] per hidden-layer and learning rates equal to [0.3, 0.1, 0.07, 0.05, 0.04, 0.03, 0.02, 0.01] is run:

```
Best testing MSE is 28.87321224090392 at indexes (6, 9), with parameters:
n_neurons: 500
Learning rate (eta): 0.02
```

Figure 49: Output with the best testing MSE and parameters neurons and etas.

Observe that the best testing MSE score of 28.87 is higher/worse than 8.19 acquired for Gaussian weights initialization on MLP with Tanh. Xavier's initialization choice might not be a good idea in this case, except for a possible better parameter values than the ones we have examined here.

Then, a parameter combination between learning rate = [0.3, 0.1, 0.07, 0.05, 0.04, 0.03, 0.02, 0.01], decays = [0, 10 ** -9, 10 ** -8, 10 ** -7, 10 ** -6, 10 ** -5] and lambdas = [0, 10 ** -5, 10 ** -4, 10 ** -3, 10 ** -2, 10 ** -1] is driven on MLP with Tanh and two hidden layers with 500 neurons, getting the subsequent outcome:

```
Best testing MSE is 25.511312704225908 at indexes (0, 4, 4), with parameters:
Eta: 0.02
Decay: 1e-06
Lambda: 0
```

Figure 50: Output with the best testing MSE and parameters etas, decay and lambdas.

The decay parameter improved the best testing score from 28.87 to 25.51, however it is still far worse than 7.46 got for Gaussian weights initialization.

Finally, a combination of parameters between the number of neurons equal to [5, 10, 20, 50, 100, 150, 200, 300, 400, 500] per hidden-layer and learning rates equal to [0.5, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001] is performed on MLP using Sigmoid activation function:

```
Best testing MSE is 228.19404341124954 at indexes (2, 7), with parameters:
n_neurons: 300
Learning rate (eta): 0.05
```

Figure 51: Output with the best testing MSE and parameters neurons and etas.

The Xavier's initialization has not helped our model to increase the performance which got 228.19 as best testing MSE score, way higher/worse than 9.37 for the Gaussian initialization on two hidden layers Sigmoid.

In addition, a parameter combination between learning rate = [0.5, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001], decays = [0, 10 ** -9, 10 ** -8, 10 ** -7, 10 ** -6, 10 ** -5] and lambdas = [0, 10 ** -5, 10 ** -4, 10 ** -3, 10 ** -2, 10 ** -1, 0.5] is performed on MLP and Sigmoid, obtaining the following results:

```
Best testing MSE is 225.6514619144 at indexes (5, 0, 2), with parameters:
Eta: 0.05
Decay: 0
Lambda: 0.1
```

Figure 52: Output with the best testing MSE and parameters etas, decay and lambdas.

Again, this result did not have any improvement from 9.37 for the Gaussian initialization on two hidden layers Sigmoid.

## 3.3 Part D

In part D, we will classify a given handwritten digit image, so-called MNIST, into one of its ten classes (0–9). The problem we are dealing with is essentially a Multi-class classification that we will be using the MLP model with Softmax as activation function and accuracy score as cost function. Afterward, we will replace the Softmax activation function with the Sigmoid and compare the results.

After that, we will analyze breast cancer data, a binary classification problem with response variable 0 for patients not containing breast cancer, and 1 for breast cancer patients. Our MLP model with Softmax activation function and accuracy score cost function will be applied for this problem to predict breast cancer incidence as a function of selected features (columns of the design matrix X). In the end, the Softmax will be substituted for Sigmoid, and the results will be compared.

### 3.3.1 MNIST data-set on MLP Classifier with one hidden layer and Softmax

At the beginning, we will employ a one-layer neural network to analyze different vital topics regarding the number of neurons, learning rate, epoch and 'l2' regularization for predicting handwritten digit image among 0-9.

The Python file partD.py in the directory Project2/ will perform the following experiments and return the results we will discuss here.

First, we would like to find the best number of neurons as a function of a learning rate ($\eta$) value. For that, n_neurons = [5, 10, 50, 100, 200, 300, 400, 500] and etas = [0.2, 0.1, 0.09, 0.05, 0.01] are defined and performed on the *classification_study* function. This function fits our Multi-layer perceptron (MLP) model with a combination of the previous parameters, one by one, to identify which one returns the best metrics as follows:

```
Best testing Accuracy is 0.725 at indexes (0, 7), with parameters:
n_neurons: 500
Learning rate (eta): 0.2
```

Figure 53: Output with the best testing accuracy for parameters neurons vs etas, for epoch=50.

The standard parameters used on the MLP model were epochs=50, batch_size = 50, learning_rate = 'constant', decays = 0.0, lmbds = 0.0, bias0 = 0.01, init_weights = 'normal', act_function = 'softmax', output_act_function = 'softmax', cost_function = 'accuracy_score', random_state = 10.

This first output shows that the best combination of parameters is neurons = 500 and learning rate = 0.2, which obtained testing accuracy at 72,5 percent.

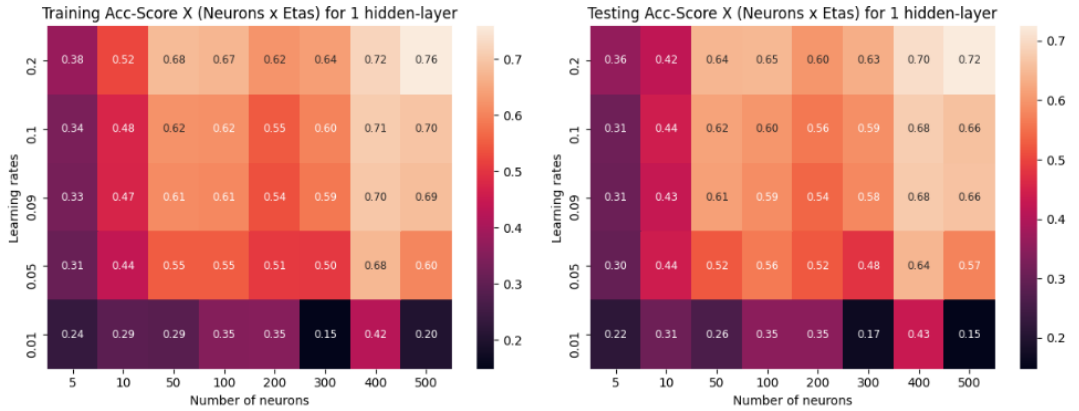



Figure 54: Heat-maps containing the training (left) and testing (right) Accuracy-scores.

One can see that for learning rate 0.2, the training and testing accuracy scores increases as long as the number of neurons increases, reaching the best scores at 500 neurons. Because of expensive computations cost, we did not analyze more than 500 neurons, however, if it would have done, it might have gotten better accuracies. Besides, one needs to be careful about over-fitting when increasing too much the number of neurons.

The best training and testing accuracies were 0.76 and 0.72, respectively, both for one hidden layer with 500 neurons and learning rate at 0.2.

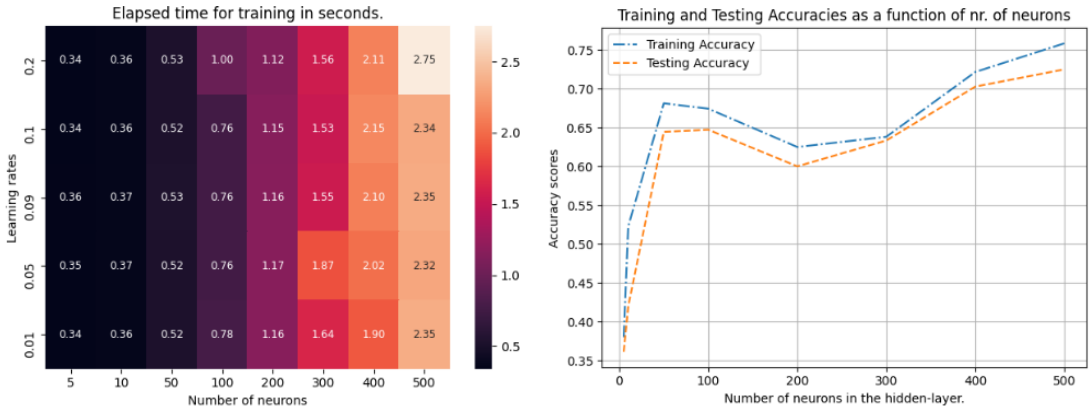



Figure 55: Heat-maps containing the training elapsed times (left), and line-plots containing the training and testing accuracies for the best parameters.

The time elapsed for training the best model was 2.75 seconds, as we see in the heat-map above. Additionally, in the line-plots above, the accuracy scores reach a top at around 100 neurons, decrease, and again reach another peak in 500 neurons. This decrease between 100 and 300 neurons might indicate the variance's effects, giving an over-estimated score for 300-500 neurons or higher.

Now, a parameter combination between learning rate = [0.2, 0.1, 0.09, 0.05, 0.01], epochs = [10, 20, 40, 60, 80, 100, 120, 140, 160, 180, 200] and lambdas = [0, 10**-9, 10 ** -7, 10 ** -5, 10 ** -3, 10 ** -1, 0.5] is performed on MLP, obtaining the following results:

```
Best testing accuracy is 0.7861111111111111 at indexes (5, 6, 1), with parameters:
Eta: 0.1
Epoch: 120
Lambda: 0.1
Training time: 6.084824800491333
```

Figure 56: Output with the best testing accuracy for parameters etas, epochs and lambdas.

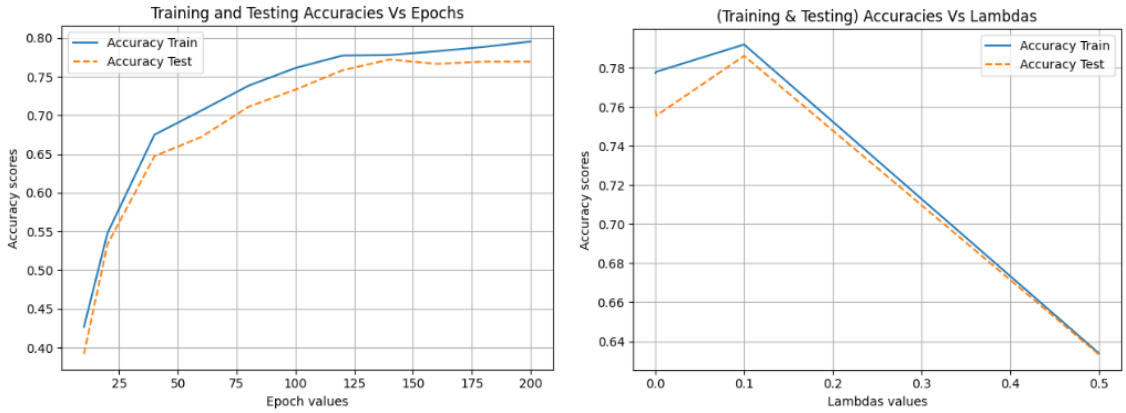



Figure 57: Line-plots containing learning rate = 0.1 as a function of epochs (left) and lambdas (right).

One can see that the epoch parameter got a positive effect resulting in an accuracy improvement on training and testing from 0.76 and 0.72 for epoch=50 to 0.8 and 0.76 for epoch=200. Again, as the number of epochs is higher, it might increase the model's variance and consequently overestimate the results.

The lambda parameter has also positively increased the results when its value is 0.1.

On the other hand, the elapsed time for training has increased/worsen from 2.75 to 7.82 seconds.

### 3.3.2 MNIST data-set on MLP Classifier with two hidden layer and Softmax

In succession, we will perform a two hidden layer experiments on MNIST data. The standard parameters will still be the same as the previous subsection and a combination of parameters between the number of neurons equal to [5, 10, 25, 50, 75, 100, 250, 500] per hidden-layer and learning rates equal to [0.2, 0.1, 0.09, 0.05, 0.01, 0.005] is performed with results as follows:

```
Best testing Accuracy is 0.525 at indexes (0, 3), with parameters:
n_neurons: 50
Learning rate (eta): 0.2
```

Figure 58: Output with the best testing accuracy for parameters neurons vs etas, for epoch=50.

The output shows that the two hidden layer model got its best testing accuracy score of 0.525, which is worse than the best testing score of 0.725 for the one hidden layer experiment.

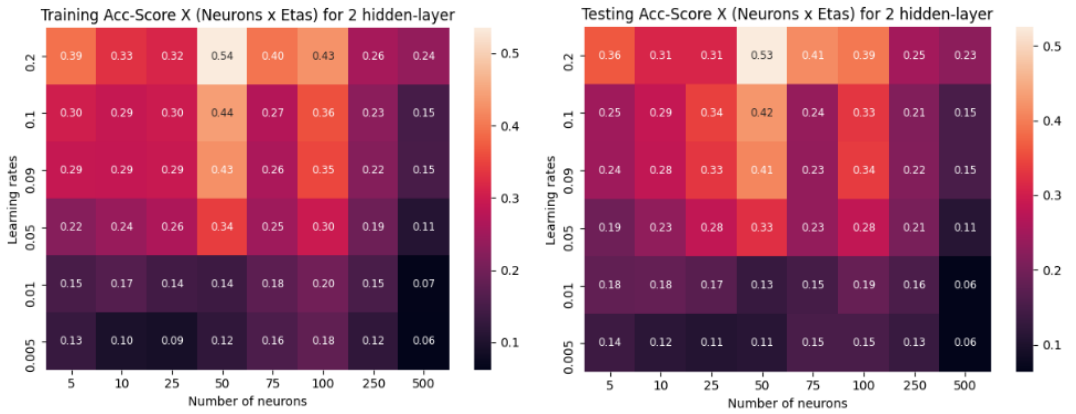



Figure 59: Heat-maps containing the training (left) and testing (right) accuracy-scores.

The best training and testing accuracies of 0.54 and 0.53 are worse than the ones achieved on one hidden layer (0.76 and 0.72).

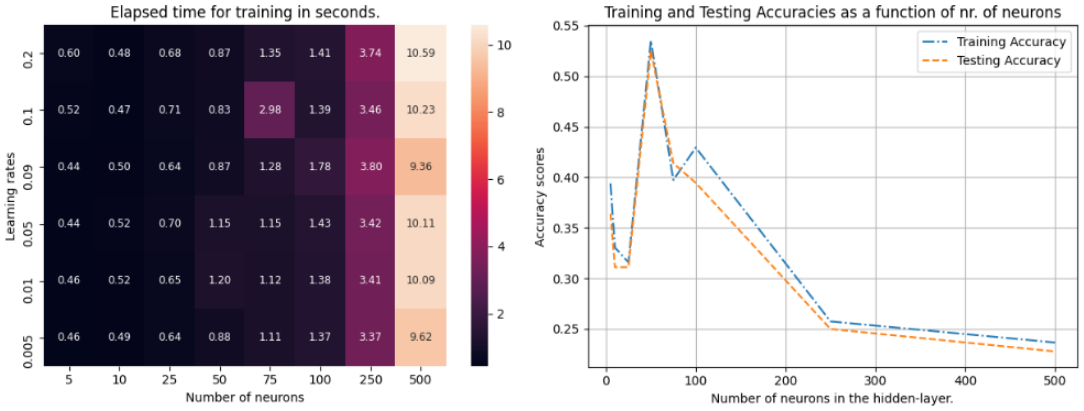



Figure 60: Heat-maps containing the training elapsed times (left), and line-plots containing the training and testing accuracies for the best parameters.

The elapsed time for training the best scores on two hidden layers is 0.87 seconds (heat-map left), for about 50 neurons in each hidden layer and 50 epochs. If we increase the number of neurons, the model collapses, and the metrics worsen very quickly, as shown in the line-plots (right).

Then, a parameter combination between learning rate = [0.2, 0.1, 0.09, 0.05, 0.01, 0.005], epochs = [10, 20, 40, 60, 80, 100, 120, 140, 160, 180, 200] and lambdas = [0, 10**-9, 10 ** -7, 10 ** -5, 10 ** -3, 10 ** -1, 0.5] is driven, getting the subsequent outcome:

```
Best testing accuracy is 0.7694444444444445 at indexes (4, 10, 0), with parameters:
Eta: 0.2
Epoch: 200
Lambda: 0.001
Training time: 3.9131481647491455
```

Figure 61: Output with the best testing accuracy for parameters etas, epochs and lambdas.

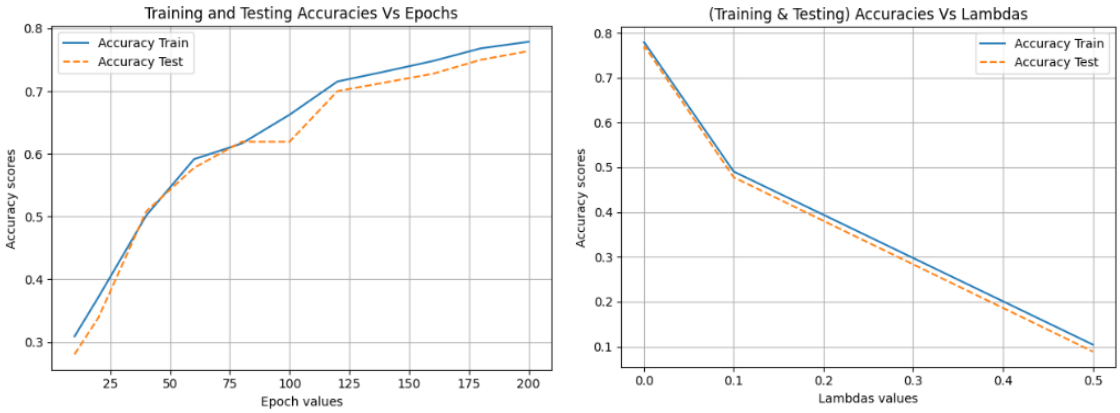



Figure 62: Line-plots containing learning rate = 0.2 as a function of epochs (left) and lambdas (right).

Notice that as the epoch parameter increases, the training and testing accuracies improve reaching its peak at 0.79 and 0.77 for epoch=200 and lambda=0.001. This score is quite the same as the best one achieved for one hidden layer model 0.8 and 0.78.

### 3.3.3 MNIST data-set on MLP Classifier with one hidden layer and Sigmoid

We will carry out one hidden layer experiments on MNIST data but choose Sigmoid instead of the Softmax activation function, and this Sigmoid Neural Networks is very similar to a Logistic Regression, except that the latter does not use FFNN and back-propagation.

The standard parameters will still be the same as the previous subsections. A combination of parameters between the number of neurons equal to [5, 10, 25, 50, 75, 100, 250, 500] per hidden-layer and learning rates equal to [0.2, 0.1, 0.09, 0.05, 0.01, 0.005] is conducted with results as observes:

```
Best testing Accuracy is 0.9722222222222222 at indexes (0, 5), with parameters:
n_neurons: 100
Learning rate (eta): 0.2
```

Figure 63: Output with the best testing accuracy for parameters neurons vs etas, for epoch=50.

This output shows that the best combination of parameters is neurons = 100 and learning rate = 0.2, which obtains an extremely good testing accuracy at 97.2 percent.

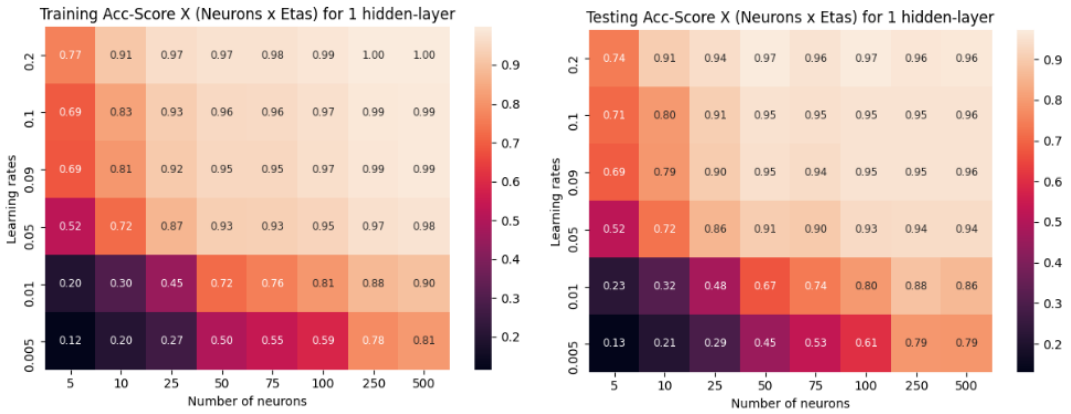



Figure 64: Heat-maps containing the training (left) and testing (right) Accuracy-scores, for epoch=50.

The training and testing heat-maps confirm the outstanding precision of the MLP model with Sigmoid activation function, acquiring the perfect score on many occasions of the training, and the best testing score of 0.972 for 100 neurons and 0.2 learning rate. This accuracy was much better than the ones achieved in the previous subsection for Softmax activation function.

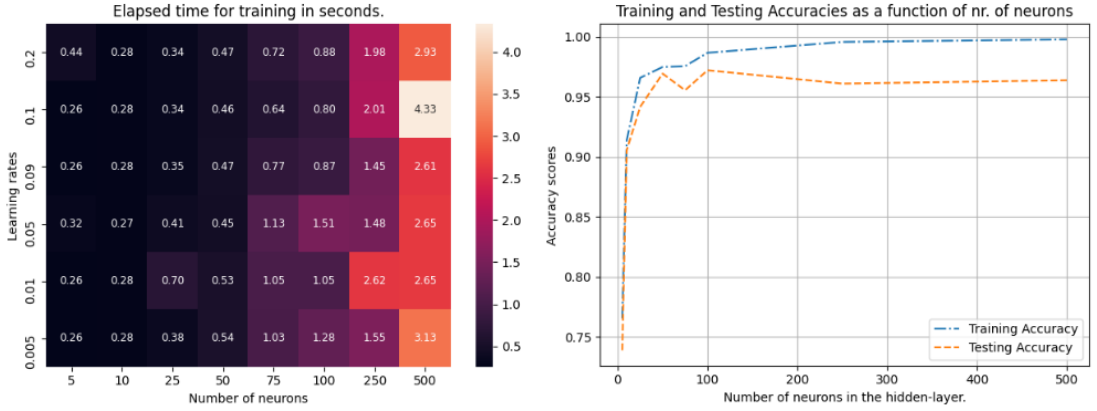



Figure 65: Heat-maps containing the training elapsed times (left), and line-plots containing the training and testing accuracies for the best parameters.

Besides, the elapsed time for training with the Sigmoid activation function was way lower with much better scores than for Softmax. In the case here, the model got the best accuracy with 0.88 seconds of training. Additionally, the number of neurons higher than 100 shows an increase in the model's variance, indicating that 100 is an optimal parameter value.

At last, a parameter combination between learning rate = [0.2, 0.1, 0.09, 0.05, 0.01, 0.005], epochs = [10, 20, 40, 60, 80, 100, 120, 140, 160, 180, 200] and lambdas = [0, 10 ** -9, 10 ** -7, 10 ** -5, 10 ** -3, 10 ** -1, 0.5] is seen:

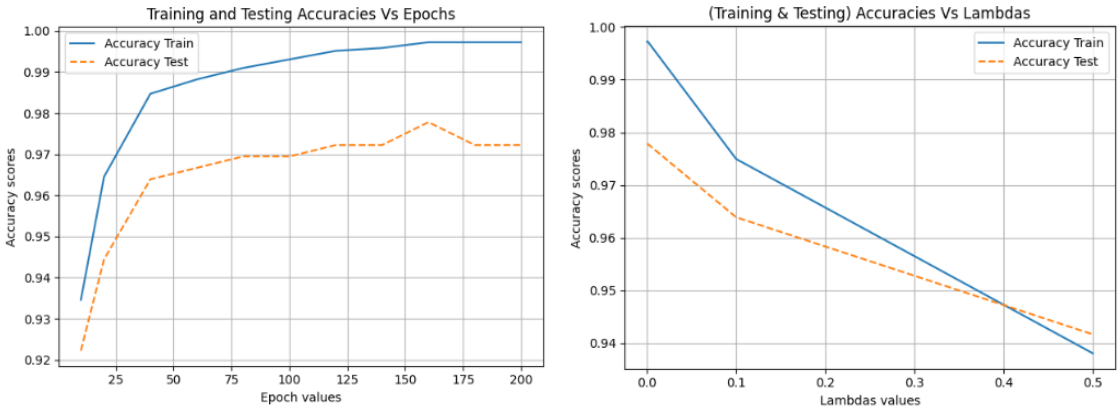



Figure 66: Line-plots containing nr_neurons = 100, learning rate = 0.2 as a function of epochs (left) and lambdas (right).

The epoch=160 and eta=0.2 proved an increase in accuracy from 0.972 (epoch=50) to 0.977 (epoch=160). Nevertheless, the left line-plots show that this accuracy of 0.977 for epoch=160 might be over-estimated due to the large distance between training and testing lines. The sweet spot should have been around 45 epochs, where the accuracy is still very precise, and the training and testing lines are near each other. The lambda plotting (right) shows that this parameter could not increase the model's performance.

### 3.3.4 MNIST data-set on MLP Classifier with two hidden layer and Sigmoid

The last analyzes for MNIST data-set will carry out two hidden layers with Sigmoid activation function. The standard parameters will still be the same as the previous subsections. A combination of parameters between the number of neurons equal to [5, 10, 25, 50, 75, 100, 250, 500] per hidden-layer and learning rates equal to [0.2, 0.1, 0.09, 0.05, 0.01, 0.005] is conducted:

```
Best testing Accuracy is 0.9638888888888889 at indexes (0, 3), with parameters:
n_neurons: 50
Learning rate (eta): 0.2
```

Figure 67: Output with the best testing accuracy for parameters neurons and etas, for epoch=50.

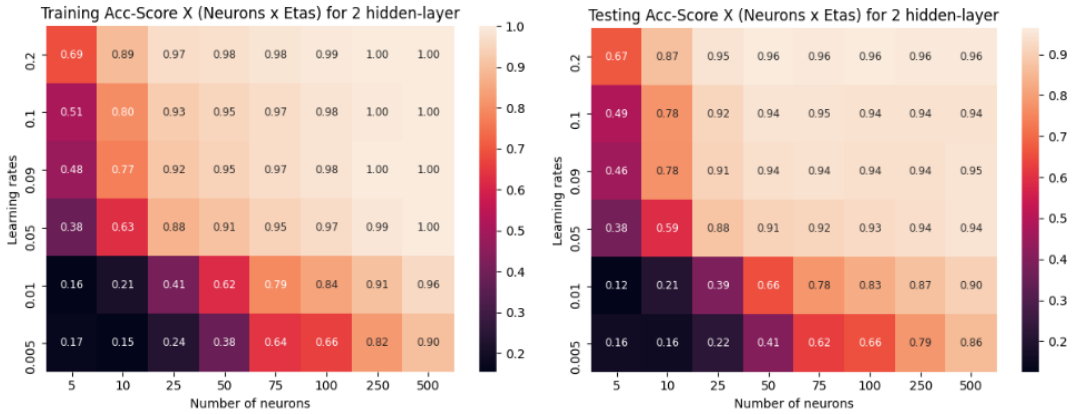



Figure 68: Heat-maps containing the training (left) and testing (right) accuracy-scores, for epoch=50.

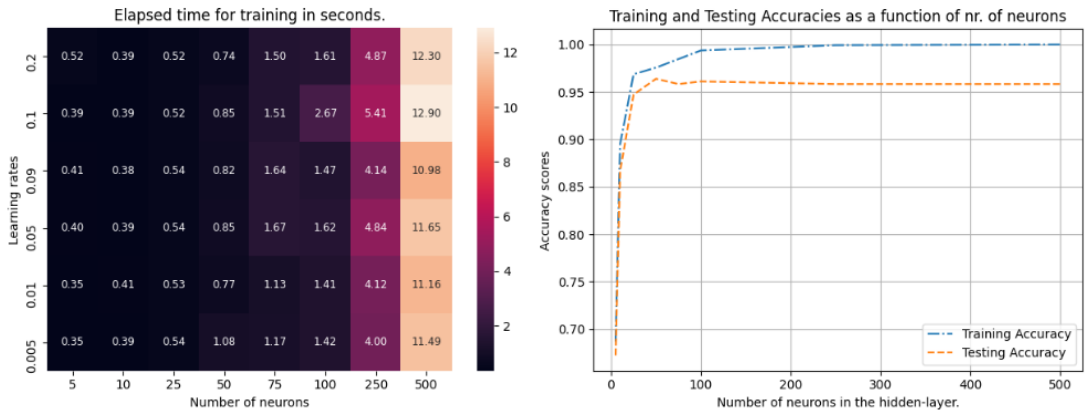



Figure 69: Heat-maps containing the training elapsed times (left), and line-plots containing the training and testing accuracies for the best parameters.

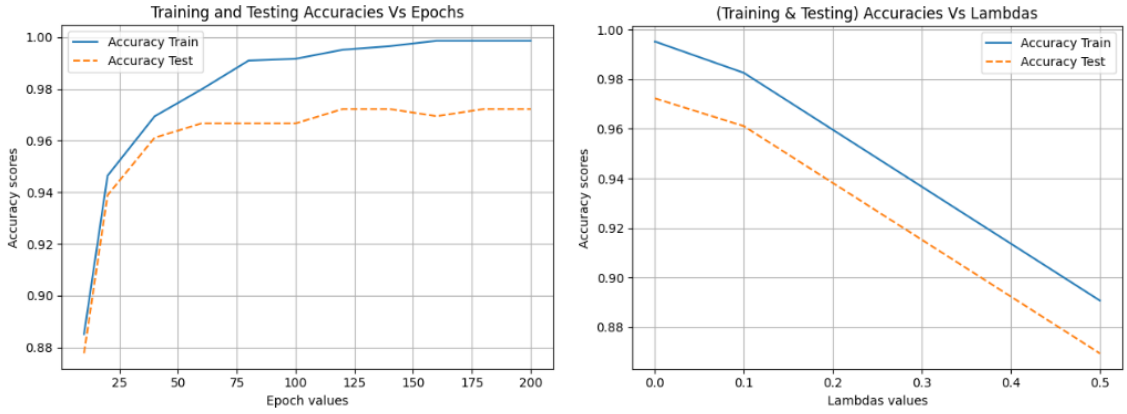



Figure 70: Line-plots containing (nr_neurons = 50, learning rate = 0.2) as a function of epochs (left) and lambdas (right).

Again, the training and testing accuracies were very high, approaching perfection. However, we must be aware of the effects of the model's variance when the number of epochs is higher than 50. There is a visible increase in the distance of the training and testing lines on the left plot, indicating the variance's effect, which causes an overestimation of the accuracy. Therefore, the accuracy for one hidden layer and two hidden layers (Sigmoid) are almost identical, with the same best parameters.

### 3.3.5 Breast Cancer data-set on MLP Classifier

In this subsection, we will analyse the Breast Cancer data-set on MLP model using Softmax and Accuracy-score activation and cost functions. The details of the Breast Cancer data is introduced in the preamble of the part D section.

We will employ a one-layer neural network to analyze different vital topics regarding the number of neurons, learning rate, epoch and 'l2' regularization for predicting the incidence of breast cancer in patients.

The Python file partD.py in the directory Project2/ will perform the following experiments and return the results we will discuss here.

In the beginning, we would like to find the best number of neurons as a function of a learning rate ($\eta$) value. For that, n_neurons = [1, 2, 3, 4, 5] and etas = [2.0, 1.0, 0.5, 0.1] are defined and performed on the *classification_study* function. This function fits our Multi-layer perceptron (MLP) model with a combination of the previous parameters, one by one, to identify which one returns the best metrics as follows:

```
Best testing Accuracy is 0.9736842105263158 at indexes (1, 3), with parameters:
n_neurons: 4
Learning rate (eta): 1.0
```

Figure 71: Output with the best testing accuracy for parameters neurons vs etas, for epoch=50 and batch_size=25.

One can see that the accuracy score achieved was extremely high, about 97,36 percent, for an unseen set (testing data). The best parameters were 4 for the number of neurons and 1 for the learning rate value, which means that the leaning rate has not any influence on the training.

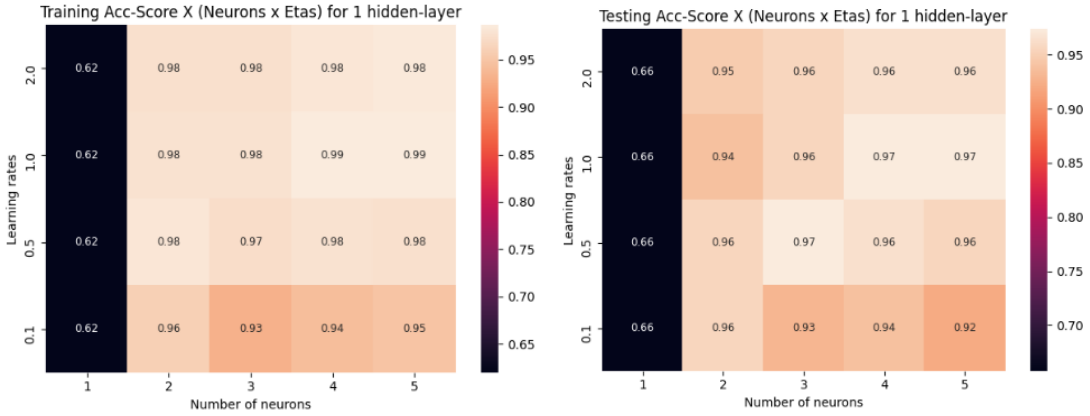



Figure 72: Heat-maps containing the training (left) and testing (right) Accuracy-scores, for nr_neurons=4, epoch=50 and batch_size=25.

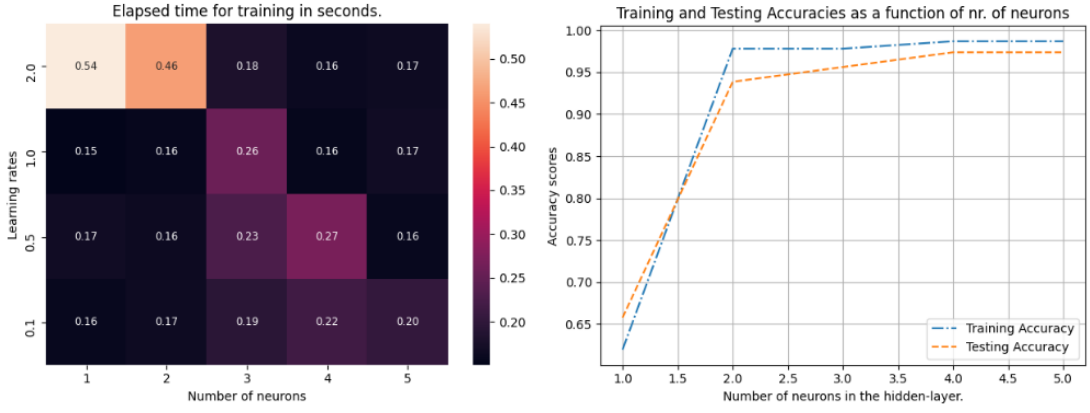



Figure 73: Heat-maps containing the training elapsed times (left), and line-plots containing the training and testing accuracies for the best parameters.

The training and testing accuracies of 0.99 and 0.97 express almost the perfection of the predictions. Notice that the model has no problem learning and predicting correctly even though with a

very low number of neurons, 4 in our best scenario (line-plots). The training elapsed time is also shallow at about 0.16 seconds, indicating that the MLP model with Softmax and Accuracy activation and cost functions do an outstanding job.

Next, a parameter combination between learning rate = [2.0, 1.0, 0.5, 0.1], epochs = [5, 6, 7, 8, 9, 10] and lambdas = [0, 10 ** -9, 10 ** -7, 10 ** -5, 10 ** -3, 10 ** -1] is seen:

```
Best testing accuracy is 0.9912280701754386 at indexes (5, 1, 0), with parameters:
Eta: 2.0
Epoch: 6
Lambda: 0.1
Training time: 0.023122072219848633
```

Figure 74: Output with the best testing accuracy for parameters etas, epochs and lambdas.

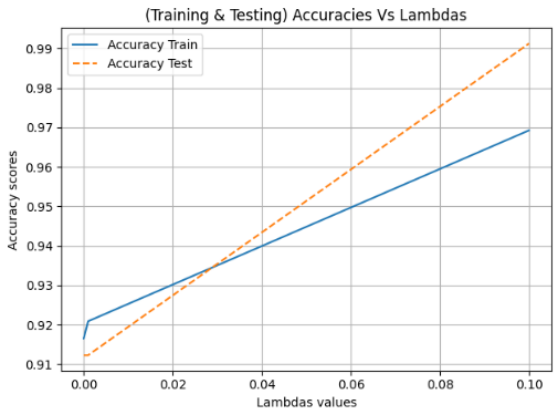



Figure 75: Line-plots containing nr_neurons = 4, learning rate = 2 and epochs = 6 as a function of 'l2' regularization lambdas = [0, 10**-9, 10 ** -7, 10 ** -5, 10 ** -3, 10 ** -1].

One can see that with parameters neurons=4, eta=2, epoch=6, and 'l2' regularization lambda=0.1, our MLP model with Softmax and Accuracy activation and cost function, respectively, reached almost the perfect score nearby 99.12 percent. The exceptional point here that we would highlight is the training time that is incredibly low at around 0.02 seconds for a fantastic accuracy of almost 100 percent.

There is no necessity to increase the number of hidden layers, epochs, initialize the weights with another method, or trying another activation or cost function due to the phenomenal metrics achieved above. We have gotten almost the perfect score for unseen data predictions, with an incredible computation efficiency of 0.02 seconds, and it would be tough to beat that.

However, we test the Sigmoid function as a replacement for the Softmax activation function for comparison purposes. For that, a parameter combination between number of neurons = [1, 2, 3, 4, 5] and learning rate = [2.0, 1.0, 0.5, 0.1] is performed:

```
Best testing Accuracy is 0.9824561403508771 at indexes (4, 0), with parameters:
n_neurons: 1
Learning rate (eta): 0.6
```

Figure 76: Output with the best testing accuracy for parameters neurons vs etas, for epoch=50 and batch_size=25.


Figure 77: Heat-maps containing the training (left) and testing (right) Accuracy-scores.

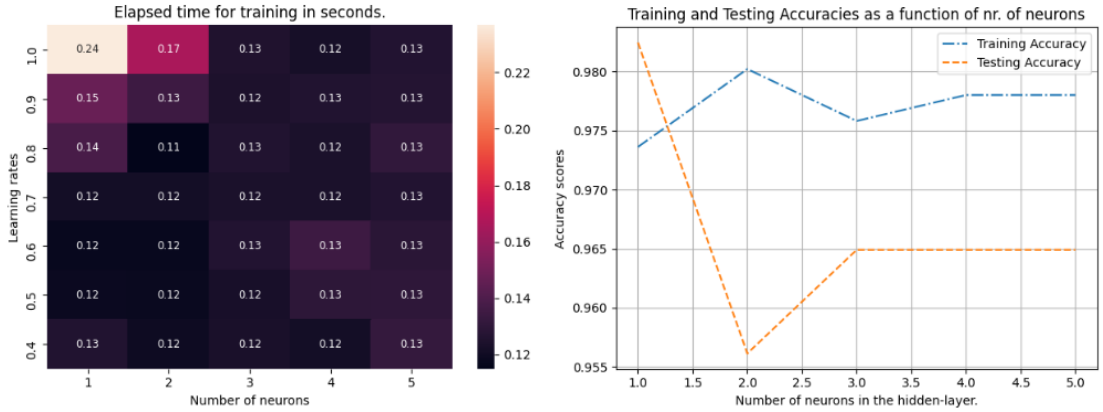



Figure 78: Heat-maps containing the training elapsed times (left), and line-plots containing the training and testing accuracies for the best parameters.

Notice that the results here with Sigmoid are not superior that 0.99 testing accuracy achieved on MLP with Softmax.

At last, a parameter combination between learning rate = [1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4], epochs = [2, 3, 4, 5] and lambdas = [0, 10 ** -9, 10 ** -7, 10 ** -5, 10 ** -3, 10 ** -1] is performed:

```
Best testing accuracy is 0.9736842105263158 at indexes (0, 3, 0), with parameters:
Eta: 1.0
Epoch: 5
Lambda: 0
Training time: 0.012618064880371094
```

Figure 79: Output with the best testing accuracy for parameters etas, epochs and lambdas, for number of neurons = 1 and batch_size=25.

Again, the best testing accuracy score of 0.9736, with Sigmoid activation function, is not superior that 0.9912 testing accuracy achieved on MLP with Softmax.

## 3.4 Part E

In the part E of this report, we will use our Logistic Regression (LR) algorithm, that uses the SGD techniques, to predict the incidence of the breast cancer in patients. A class function so-called LR is present at the file gradient_descent.py in directory Project2/package/, which performs the model.

First, we would like to find the best number of epochs as a function of a learning rate ($\eta$) value. For that, epochs = [5, 10, 25, 50, 100, 250, 500] and etas = [3, 2, 1, 0.1, 0.05, 0.01] are defined and performed on the *classification_study* function. This function fits our Logistic Regression (LR) model with a combination of the previous parameters, one by one, to identify which one returns the best metrics as follows:

```
Best testing Accuracy is 0.9385964912280702 at indexes (2, 4), with parameters:
Epoch: 100
Learning rate (eta): 1
```

Figure 80: Output with the best testing accuracy for parameters epochs vs etas, for batch_size=25 and random_state=10.

This first output shows that the best combination of parameters is epoch = 100 and learning rate = 1, which obtained the best testing accuracy at 93,85 percent.

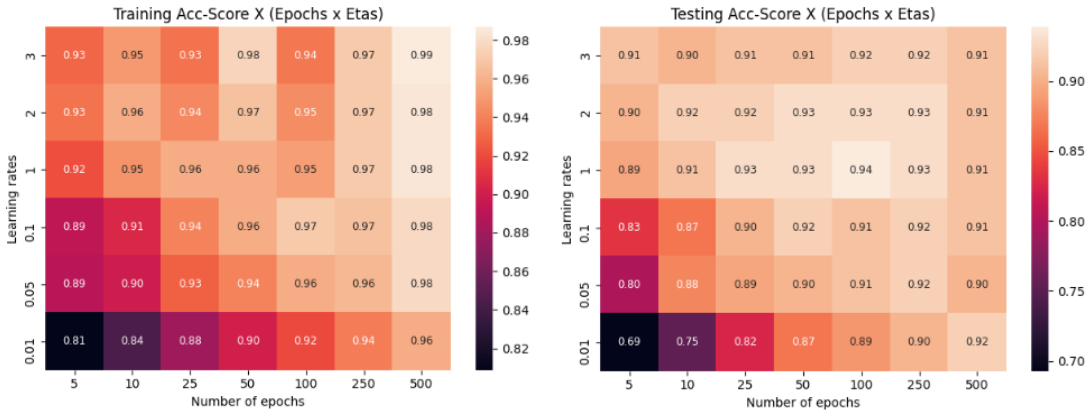



Figure 81: Heat-maps containing the training (left) and testing (right) Accuracy-scores, for batch_size=25 and random_state=10.

The best training and testing accuracies were 0.95 and 0.94, respectively, for epoch=100 and learning rate = 1.

One can see that for learning rate at 1, the testing accuracy scores increase as long as the number of epochs increases, until a certain point (100 epochs) where it reaches its best testing scores at 0.94. After 100 epochs, we see that the accuracy diminishes, indicating over-fitting.

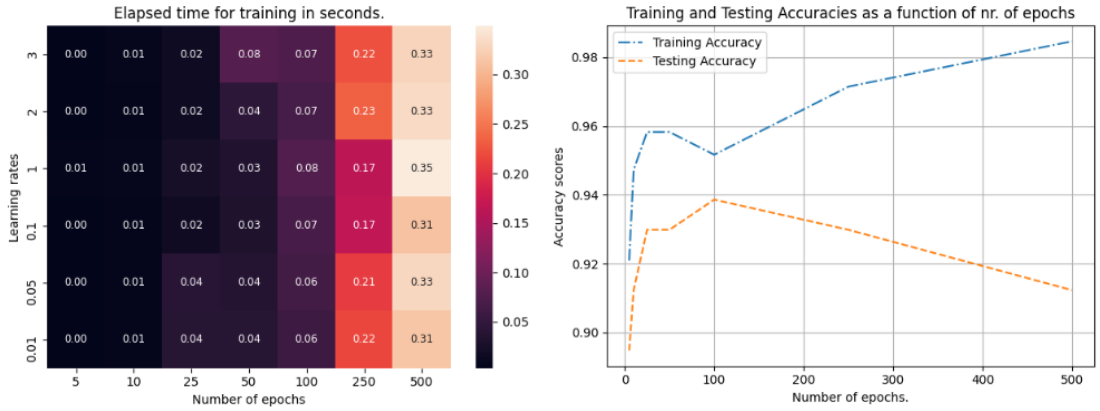



Figure 82: Heat-maps containing the training elapsed times (left), and line-plots containing the training and testing accuracies as a function of the number of epochs.

The line-plot in the right of the picture above confirms that the sweet spot for the number of epochs is 100, where the training and testing lines are closest. After this point, it is clear that the training and testing lines progress in opposite directions, confirming the model's variance's negative effect.

The heat-map in the left side of the figure above shows the elapsed time for training. It the case of the best testing accuracy, the elapsed training time was 0.08 seconds.

Now, a parameter combination between learning rate = [3, 2, 1, 0.1, 0.05, 0.01], epochs = [5, 10, 25, 50, 100, 250, 500] and lambdas = [0, 10**-9, 10 ** -7, 10 ** -5, 10 ** -3, 10 ** -1, 0.5] is performed on LR, obtaining the following results:

```
Best testing accuracy is 0.9385964912280702 at indexes (0, 4, 2), with parameters:
Eta: 1
Epoch: 100
Lambda: 0
Training time: 0.07767081260681152
```

Figure 83: Output with the best testing accuracy for parameters etas, epochs and lambdas.

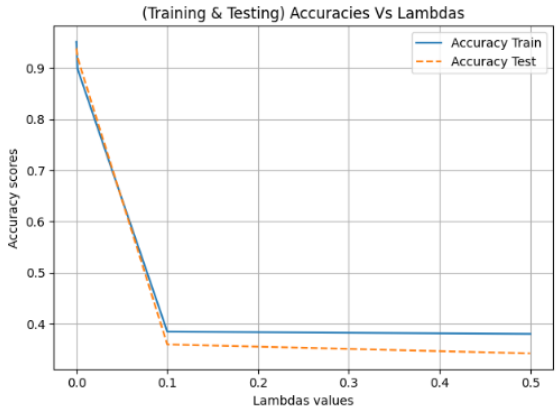



Figure 84: Line-plots containing learning rate = 1 and epochs=100 as a function of lambdas.

The lambda parameter has not positively improved the results.

# 4 Conclusion and a critical evaluation of the various algorithms

After an exhaustive study of the various models and techniques, we are prepared to decide which method is most appropriate for each type of circumstance.

| MODEL | ALGO NAME | DATA SET | PIXELS | SCALED X ? | POLY DEGREE | ACT. FUNCTION | WEIGHTS INITIALIZATION | NR. HIDDEN LAYERS | NR. NEURONS | ETA | DECAY | GAMMA | LAMBDA | BATCH SIZE | EPOCHS | SEED | Training Time | Training R2-score | Testing R2-score | Training MSE | Testing MSE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OLS | OLS | GeoTIFF | 15X15 | Yes | 10 | do not have | do not have | do not have | do not have | do not have | do not have | do not have | do not have | do not have | do not have | 10 | - | - | - | - | 1.11 |
| Ridge | Ridge | GeoTIFF | 15X15 | Yes | 11 | do not have | do not have | do not have | do not have | do not have | do not have | do not have | 0.0001 | do not have | do not have | 10 | - | - | - | - | 1.44 |
| SGD | MiniSGDM | GeoTIFF | 15X15 | No | 10 | do not have | do not have | do not have | do not have | 0.02 | 0 | 0 | 0 | 1 | 10.000 | 10 | 120.32 s | 0.99 | 0.99 | 2.77 | 2.99 |
| MiniSGD | MiniSGDM | GeoTIFF | 15X15 | No | 10 | do not have | do not have | do not have | do not have | 0.01 | 0 | 0 | 0 | 10 | 1.000 | 10 | 2.57 s | 0.99 | 0.99 | 4.19 | 4.48 |
| MiniSGD | MiniSGDM | GeoTIFF | 15X15 | No | 10 | do not have | do not have | do not have | do not have | 0.3 | 0.001 | 0 | 0 | 10 | 1.000 | 10 | 2.61 s | 0.99 | 0.99 | 3.99 | 4.32 |
| MiniSGDM | MiniSGDM | GeoTIFF | 15X15 | No | 10 | do not have | do not have | do not have | do not have | 0.3 | 0.001 | 0.3 | 0 | 10 | 1.000 | 10 | 2.82 s | 0.99 | 0.99 | 3.8 | 4.18 |
| MiniSGDM | MiniSGDM | GeoTIFF | 15X15 | No | 10 | do not have | do not have | do not have | do not have | 0.3 | 0.001 | 0.3 | 1e-09 | 10 | 1.000 | 10 | 2.77 s | 0.99 | 0.99 | 3.8 | 4.18 |
| NN | MLP | GeoTIFF | 15X15 | Yes | do not have | Sigmoid/Identity | Gaussian | 1 | 100 | 0.9 | 0 | 0 | 0 | len(samples) | 500 | 10 | 0.54 s | 0.99 | 0.99 | 8.84 | 10.08 |
| DNN | MLP | GeoTIFF | 15X15 | Yes | do not have | Sigmoid/Identity | Gaussian | 2 | 500 | 0.08 | 0 | 0 | 0 | len(samples) | 500 | 10 | 12.70 s | 0.98 | 0.98 | 8.28 | 9.37 |
| DNN | MLP | GeoTIFF | 15X15 | Yes | do not have | Tanh/Identity | Gaussian | 2 | 400 | 0.04 | 0 | 0 | 0 | len(samples) | 500 | 10 | 7.73 s | 0.99 | 0.98 | 6.78 | 8.19 |
| DNN | MLP | GeoTIFF | 15X15 | Yes | do not have | Tanh/Identity | Gaussian | 2 | 400 | 0.04 | 1e-06 | 0 | 0 | len(samples) | 500 | 10 | 7.85 s | 0.99 | 0.98 | 6.34 | 7.46 |
| DNN | MLP | GeoTIFF | 15X15 | Yes | do not have | ReLu/Identity | Gaussian | 2 | 9 | 0.0005 | 0 | 0 | 0 | len(samples) | 500 | 10 | 0.27 s | 0.96 | 0.97 | 22.91 | 14.18 |
| DNN | MLP | GeoTIFF | 15X15 | Yes | do not have | ReLu/Identity | Gaussian | 2 | 9 | 0.0005 | 1e-06 | 0 | 0.99 | len(samples) | 500 | 10 | 0.33 s | 0.97 | 0.97 | 22.56 | 13.50 |
| DNN | MLP | GeoTIFF | 15X15 | Yes | do not have | ReLu/Identity | Xavier | 2 | 500 | 0.0005 | 0 | 0 | 0 | len(samples) | 500 | 10 | - | - | - | - | 13.69 |
| DNN | MLP | GeoTIFF | 15X15 | Yes | do not have | ReLu/Identity | Xavier | 2 | 500 | 0.0005 | 1e-07 | 0 | 0.1 | len(samples) | 500 | 10 | - | - | - | - | 13.60 |
| DNN | MLP | GeoTIFF | 15X15 | Yes | do not have | Tanh/Identity | Xavier | 2 | 500 | 0.02 | 0 | 0 | 0 | len(samples) | 500 | 10 | - | - | - | - | 28.87 |
| DNN | MLP | GeoTIFF | 15X15 | Yes | do not have | Tanh/Identity | Xavier | 2 | 500 | 0.02 | 1e-06 | 0 | 0 | len(samples) | 500 | 10 | - | - | - | - | 25.51 |
| DNN | MLP | GeoTIFF | 15X15 | Yes | do not have | Sigmoid/Identity | Xavier | 2 | 300 | 0.05 | 0 | 0 | 0 | len(samples) | 500 | 10 | - | - | - | - | 228.19 |
| DNN | MLP | GeoTIFF | 15X15 | Yes | do not have | Sigmoid/Identity | Xavier | 2 | 300 | 0.05 | 0 | 0 | 0.1 | len(samples) | 500 | 10 | - | - | - | - | 225.65 |
| | | | | | | | | | | | | | | | | | | | | | |
| | Bench-mark to be beaten. | | | | | | | | | | | | | | | | | | | | |
| | The best model, when computation's efficiency and precision metrics are compared and analyzed. | | | | | | | | | | | | | | | | | | | | |
| | Consistent precision metrics, but not much efficient in terms of computation's cost. | | | | | | | | | | | | | | | | | | | | |

Figure 85: Fitting different models and parameters, and assessing predictions for the GeoTIF image data.

Starting with the GeoTIF terrain data-set, the table above demonstrated that the Stochastic Gradient Descent (SGD) model presented the best MSE testing score of 2.99 among Gradient Descent (GD) and Neural Networks (NN) varieties. This score is pretty close to the ones achieved in the traditional methods of Ordinary Least Square (OLS) of 1.11 and Ridge Regression (RIDGE) of 1.44. However, the SGD model proves to be very inefficient in computation cost, as evidenced by the elapsed training time of 120 seconds. The other types of GD are quicker in terms of training but less accurate. Furthermore, the GD model's types might be more precise than the NN model's diversities because the former applies a polynomial transformation on its input data, likewise OLS and RIDGE, while the latter uses random numbers between 0 and 1. The NN species' most significant benefit is that the descriptive data does not need a polynomial transformation to feed the model.

| MODEL | ALGO NAME | DATA SET | SCALED X ? | ACT. FUNCTION | WEIGHTS INITIALIZATION | NR. HIDDEN LAYERS | NR. NEURONS | ETA | LAMBDA | BATCH SIZE | EPOCHS | SEED | Training Time | Training Accuracy | Testing Accuracy |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NN | MLP | MNIST | Yes | Softmax | Gaussian | 1 | 500 | 0.2 | 0.0 | 50 | 50 | 10 | 2,75 | 0.76 | 0.72 |
| NN | MLP | MNIST | Yes | Softmax | Gaussian | 1 | 500 | 0.1 | 0.1 | 50 | 120 | 10 | 6,08 | 0.79 | 0.786 |
| DNN | MLP | MNIST | Yes | Softmax | Gaussian | 2 | 50 | 0.2 | 0.0 | 50 | 50 | 10 | 0,87 | 0.54 | 0.53 |
| DNN | MLP | MNIST | Yes | Softmax | Gaussian | 2 | 500 | 0.2 | 0.001 | 50 | 200 | 10 | 3,91 | 0.77 | 0.76 |
| NN | MLP | MNIST | Yes | Sigmoid | Gaussian | 1 | 100 | 0.2 | 0.0 | 50 | 50 | 10 | 0.88 | 0.99 | 0.972 |
| NN | MLP | MNIST | Yes | Sigmoid | Gaussian | 1 | 100 | 0.2 | 0.0 | 50 | 160 | 10 | 2.64 | 0.99 | 0.977 |
| DNN | MLP | MNIST | Yes | Sigmoid | Gaussian | 2 | 50 | 0.2 | 0.0 | 50 | 50 | 10 | 0.74 | 0.98 | 0.96 |
| DNN | MLP | MNIST | Yes | Sigmoid | Gaussian | 2 | 50 | 0.2 | 0.0 | 50 | 120 | 10 | 2.57 | 0.99 | 0.97 |
| | | | | | | | | | | | | | | | |
| | The best model, when computation's efficiency and precision metrics are compared and analyzed. | | | | | | | | | | | | | | |

Figure 86: Fitting different models and parameters, and assessing predictions for the MNIST data.

In its turn, the MNIST's results show that the Multi-Layer Perceptron (MLP) model using Sigmoid as the activation function and Accuracy-score as the cost function is more precise than Softmax with Accuracy-score. In fact, the handwriting images' predictions got an excellent testing accuracy of 97 percent with Sigmoid, while only 78 percent with Softmax. It is essential to state that an MLP model with Sigmoid and Accuracy-score resembles a Logistic Regression model, except

that the former utilizes NN techniques as Mini-batch, Feed-Forward, and Back-Propagation, while the latter only uses Mini-batch and gradient descent.

| MODEL | ALGO NAME | DATA SET | SCALED X? | ACT. FUNCTION | WEIGHTS INITIALIZATION | NR. HIDDEN LAYERS | NR. NEURONS | ETA | LAMBDA | BATCH SIZE | EPOCHS | SEED | Training Time | Training Accuracy | Testing Accuracy |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NN | MLP | Breast Cance | Yes | Softmax | Gaussian | 1 | 4 | 1 | 0.0 | 25 | 50 | 10 | 0.16 | 0.99 | 0.97 |
| NN | MLP | Breast Cance | Yes | Softmax | Gaussian | 1 | 4 | 2 | 0.1 | 25 | 6 | 10 | 0.02 | 0.99 | 0.99 |
| NN | MLP | Breast Cance | Yes | Sigmoid | Gaussian | 1 | 1 | 0.6 | 0.0 | 25 | 50 | 10 | 0.12 | 0.97 | 0.98 |
| NN | MLP | Breast Cance | Yes | Sigmoid | Gaussian | 1 | 1 | 1 | 0.0 | 25 | 5 | 10 | 0.012 | 0.97 | 0.97 |
| LR | LR | Breast Cance | Yes | Sigmoid | - | - | - | 1 | 0.0 | 25 | 100 | 10 | 0.07 | 0.95 | 0.94 |
| | | | | | | | | | | | | | | | |
| | The best model, when computation's efficiency and precision metrics are compared and analyzed. | | | | | | | | | | | | | | |

Figure 87: Fitting different models and parameters, and assessing predictions for the Breast Cancer data.

Finally, the Breast Cancer outcomes reveal that the MLP model can predict cancer incidence in patients with an outstanding facility, precision, and efficiency. Actually, with only one hidden layer and as few as one neuron in this layer, the MLP with Sigmoid and Accuracy-score activation and cost functions, respectively, got the testing exactness metric of 97 percent. On the other hand, the MLP with Softmax and Accuracy-score activation and cost functions, respectively, got even better testing precision of 99 percent. Both models were rapid in their training, without any relevant difference. Regarding to the Logistic Regression method, it also proves to be very reliable and quick, although with less precision than NN models.

After all, one can conclude that the NN performed by MLP was more trustworthy for Classification than for Regression problems. However, NN and MLP are essential instruments to predict non-linear regression problems that can not be submitted to a polynomial transformation before fitting. Besides, NN models can achieve better results if applied with a larger number of hidden layers, neurons, and epochs. This study was restricted by two hidden layers and 500 epochs due to expensive computation cost, but a larger value for these parameters could have gotten more considerable metrics, which could be a good idea for future works.

# 5 Bibliography

[1] T. Hastie, R. Tibshirani, J. Friedman. *The Elements of Statistical Learning data mining, inference, and prediction.* **2nd edition**. Springer; New York, USA. 2017.

[2] Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning.* **2nd edition**. Packt; Birmingham, UK. 2017.

[3] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow.* O'Reilly Media; Sebastopol, CA. 2017.

[4] University of Oslo. "Week 40: From Stochastic Gradient Descent to Neural networks". Department of Physics, University of Oslo. https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40.html. (Oct 6, 2020)

[5] University of Oslo. "Week 41 Tensor flow and Deep Learning, Convolutional Neural Networks". Department of Physics, University of Oslo. https://compphysics.github.io/MachineLearning/doc/pub/week41/html/week41.html. (Oct 10, 2020)

[6] "Understanding the Difficulty of Training Deep Feedforward Neural Networks," X. Glorot, Y Bengio (2010).