UiO : Department of Physics
University of Oslo

Alicja Terelak, Giorgio Chiro, Eyyüb Güven

Project 2

FYS-STK4155 — Project Work in Applied Data Analysis and Machine Learning

Supervisor: Morten Hjorth-Jensen

2023

Abstract

This project examines the gradient descent methodologies within the realm of neural networks with OLS and Ridge Regressions focusing on the implementation of various algorithms such as standard GD, Stochastic GD, RMSprop, and ADAM. Employing these techniques on synthetic data from Franke's Function and real data from the Wisconsin Breast Cancer dataset, the study investigates their efficacy and nuances. To do this we will be using Feed Forward Neural Network (FFNN), while dissecting its architecture, activation functions. The comparison is extended to include our Logistic regression code. Results spotlight the Stochastic GD in minimizing Mean Squared Error (MSE), and the FFNN's adeptness in classifying real-world data, revealing optimal activation functions and shedding light on recognition rates for breast cancer cases.

1 Introduction

In this project we will explore the functionality and precision of various gradient descent methods, and how they can be implemented in neural networks. We will also discuss and analyze a Feed Forward Neural Network, used on synthetic data generated by the Franke's Function in addition to some real data from the Wisconsin Breast Cancer data-set. We then will try out different activator functions, to test their effectiveness on our Franke's Function data. We then will compare the results obtained by the Wisconsin Neural network with our own Logistic regression code.

2 Methods

2.1 Gradient Descent

The first part of this project consists in writing the code necessary to compute the family of algorithms known as Gradient Descent. In general, Gradient Descent is a method used to get an approximation of the *beta* optimal parameter. As hinted at by the name, this is done through the descent dictated by the gradients of a multi variable function. To calculate the Gradient Descent (GD) we need 2 components: the Cost/Loss function and a function to compute the gradient. The Cost/Loss function is a function that assigns a "cost" to a given value, and it can be used to determine how off we are from the optimal parameters. This is why we need to minimize this function as we need to find the optimal *beta* parameter, which is the one that has the minimum cost. Usually the gradient of a function indicates the steepest-ascent, which is the direction from a given point where the function rises the quickest in value, but our objective is the exact opposite, as we need to minimize the Cost Function. The simpliests form of GD is the standard GD, which simply computes the Gradient and then calculates the new theta value based on the learning rate (eta) and the gradient. The theta is the parameter which we are computing, which will become our approximation of the optimal *beta* parameter over n iterations, while eta is our learning rate, which is simply a measure of how quickly our descent will be. The main reason for why the base GD is not optimal lies in

the fact that we only take one point as a start and we compute from there until we reach a minimum, which could not be the absolute minimum of the function, but only a local minimum.A slightly more advanced GD method is Stochastic Gradient Descent (SGD), which improves on the flaws of the base GD by dividing the data points in N mini-batches and running a GD on each of them until the absolute minimum point found in the bounds of the data is found. Both base GD and SGD have a variation that includes momentum, which can speed up the learning by increasing the descent speed. Even though these previous algorithms are good at minimizing the cost function, the best GD algorithms are Root mean squared propagation (RMSprop) and ADAM. In RMSprop we also keep track of the second moment of the gradient, which allows us to speed up the convergence rate. In ADAM we keep note of both the first and second moment of the gradient, which allows us to adapt the learning rate for different parameters. Every GD method also has the possibility of using automatic differentiation, which computes the partial derivative of a given function by exploiting the chain rule applied repeatedly, and can greatly speed up the computation for the gradient.

2.2 Our first Feed Forward Neural Network

In this project we will be working with the Feed Forward Neural Network (FFNN) architecture. This architecture is composed of three major components:

1. The input layer composed of N nodes, where the data will be fed to the Neural Network

2. The hidden layers composed of N nodes each, where the inputs will be computed and modified

3. The output layer composed of one node per category, where the probabilities computed by the Neural Network will be displayed

Other important components of the Neural Network are the weights and biases: the weights represent how important the value given by a certain node is, while the biases are values added to the result of multiplication of the weights and inputs, and it is used to skew the activation function towards the negative or positive side. The activation function is a function assigned to each node except for the input layer nodes, and it's the function that determines the "value" of the node, which will then be fed to the next layer. There are many activation functions, like sigmoid or RELU, and each has a use based on the type of classification we are conducting. The general formula to calculate the value of the node is:

$$a_h = \sum_{i=0}^{N} w_{i,j}^{l} x_j + b_i^l \tag{1}$$

where l indicates the layer where we are now, N is the number of nodes in the previous layer and j indicates the current node for which we are doing the computation. Once we get to the last layer, the output layer, we have two choices: we either assign or not an activation function to the nodes. If we do, we get classification, while if we leave the outputs as-is we get regression. In this specific case we will not put any activation functions on the end-layer, as

we want to predict the values of the franke's function given by two numbers. The structure of our neural Network will be 2 input nodes, one hidden layer with 2 nodes and one output node which will contain our result. In addition we will be also testing various different types of activation functions, and we will be comparing their performances with each other. The cost function that we will be using is going to be the difference between the prediction given by the Neural network and the Target we want to reach. To measure the accuracy of the predictions we will be using the Mean Squared Error

The basic steps of a FFNN are the following:

1. Perform a feed forward, where the NN generates a prediction
2. Compare the output obtained with the Target we wish to achieve
3. Perform the back propagation step, in which the weights and biases for each node are updated, starting from the output layer and ending at the input layer
4. Repeat steps 1-3 for N iterations, with each iteration improving the accuracy of the prediction

Included in the NN will be a regularization parameter *lambda*, which is used to constrain the value of the weights. This is used to reduce the chance of overfitting and it allows us to calculate the gradients more efficiently

2.3 Classification of real data using a FFNN

In this section we will use a FFNN to classify some real world data, in this case we will use the Wisconsin Breast Cancer data-set, which contains 9 features per case with 699 cases. The activation function used in all nodes will be Sigmoid, and the cost function will be the basic one as described in the 2nd point, while the accuracy score we will be using will be the accuracy, with formula:

$$Accuracy = \frac{\sum_{i=0}^{n} I(t_i = y_i)}{n} \tag{2}$$

where I is the indicator function with Benign tumors as value 0 and Malign tumors as value 1. In addition, after finding the best eta and lambda values we will also show the Confusion Matrix to show the ratio between True and False Positives and Negatives.

2.4 Using Logistic Regression on real data

Logistic regression is essentially a simpler version of a FFNN, since it only has the input and output layer. This simplifies the overall architecture of a NN, and it is also much more efficient and quick, since there are less weights and biases to multiply and update at each step. We will then compare the results obtained from this form of regression with the results obtained with a FFNN, in function of the learning rates and lambda parameters.

3 Results

3.1 Mean squared error on our own gradient descent code, using OLS and Ridge regression

In this first part we will compare the various variants of gradient descent algorithms, used on the OLS and Ridge regression. Where relevant we used 7 λ and η values ranging from $1e^{-5}$ to $1e^{1}$.

First, we are going to show our results for the OLS:

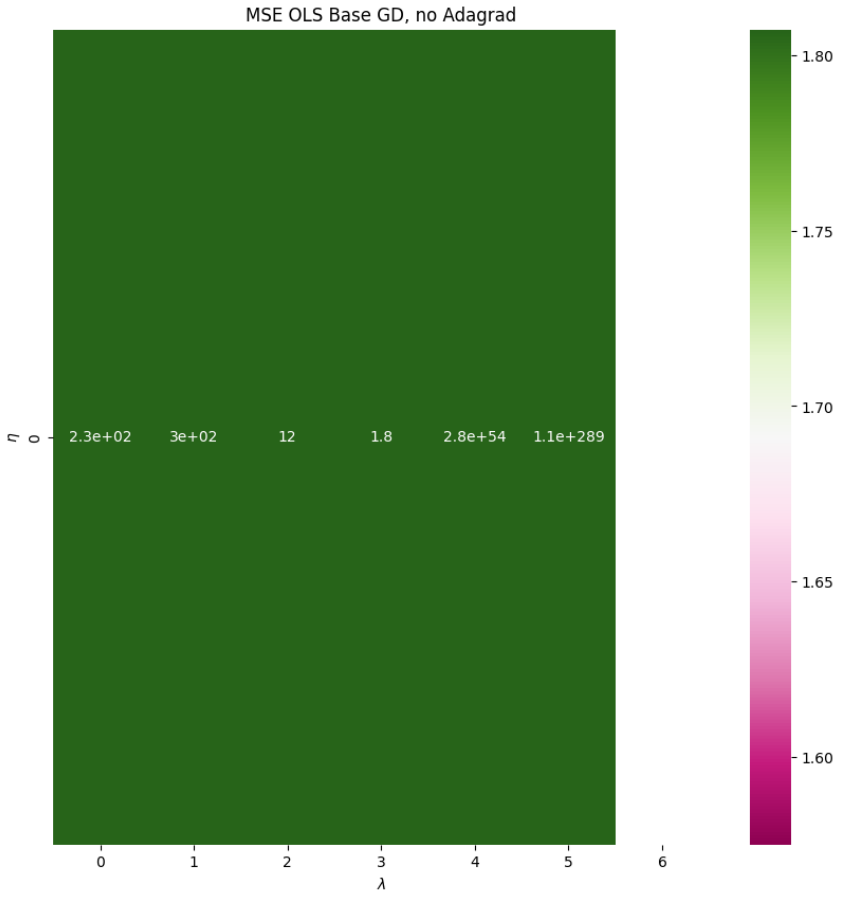

Figure 1: Base gradient descent without Adagrad.

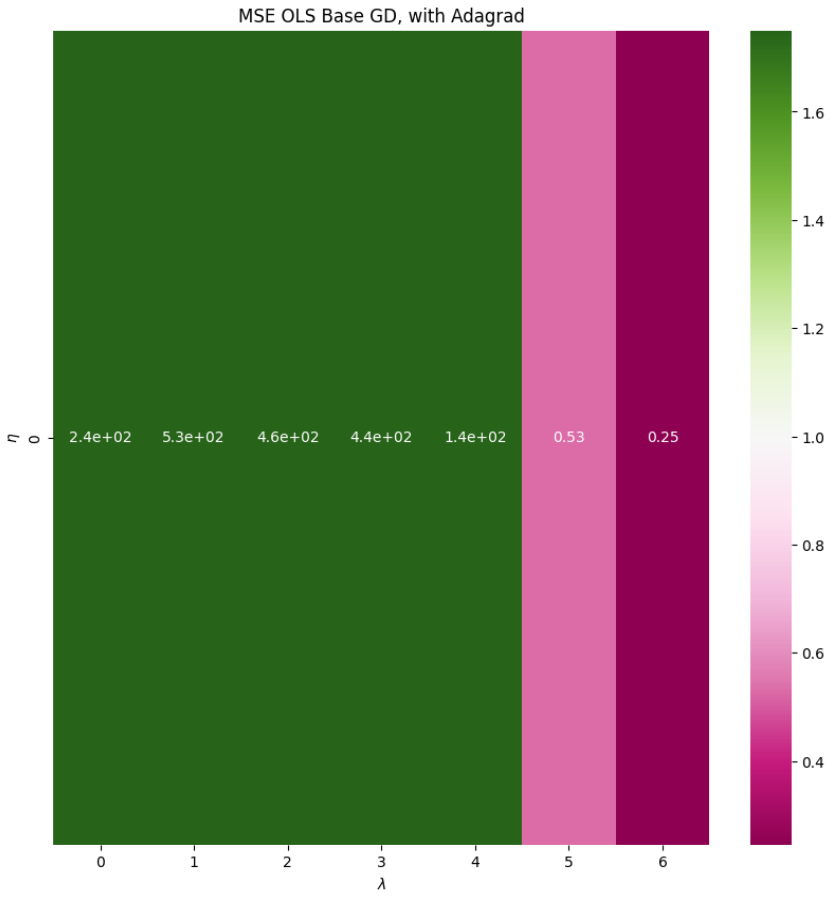

Figure 2: Base gradient descent using Adagrad.

In figures 1 and 2 we can see the differences between an implementation of a base gradient descent using Adagrad and an implementation without using it. We can conclude that between the two the better one is the Adagrad implementation, as it does not create infinite values (like those seen for $\eta = 1e^1$ in the base implementation), and it gives overall better results.

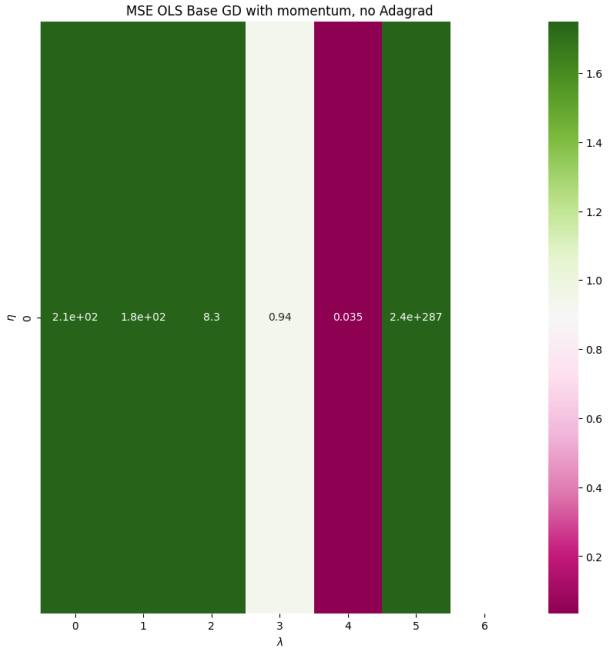

Figure 3: Base gradient implementation using momentum, using a momentum value of 0.5.

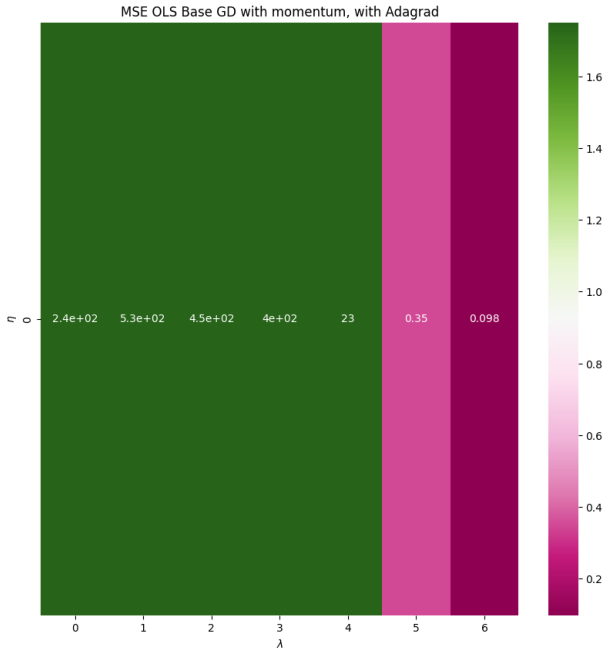

Figure 4: Base gradient implementation using momentum, using a momentum value of 0.5, with Adagrad.

In figure 3 and 4 we see the results of a base gradient implementation using momentum, using a fixed momentum value of 0.5. As before the Adagrad implementation is the better one of the two, as it returns better results overall.

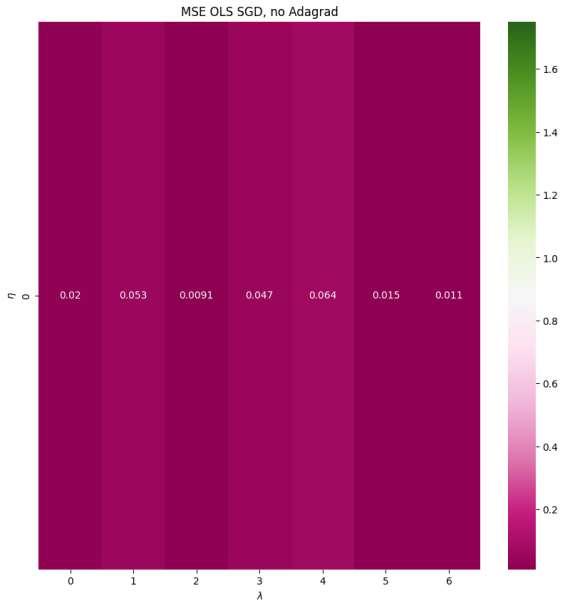

Figure 5: Stochastic gradient descent without momentum.

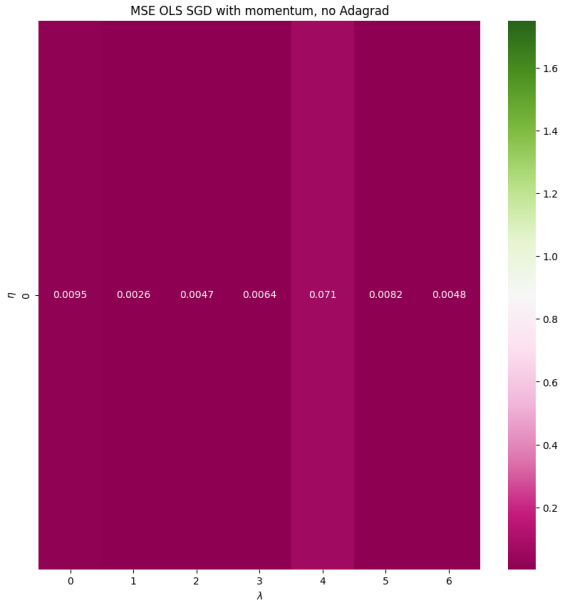

Figure 6: Stochastic gradient descent with momentum, using a momentum value of 0.5.

In figures 5 and 6 we see the results of a stochastic gradient descent, with and without momentum, using a momentum value of 0.5. As expected, the stochastic gradient descent gives better results compared to the base gradient descent implementation. Between the two implementations the momentum one is the better, as it gives much better results.

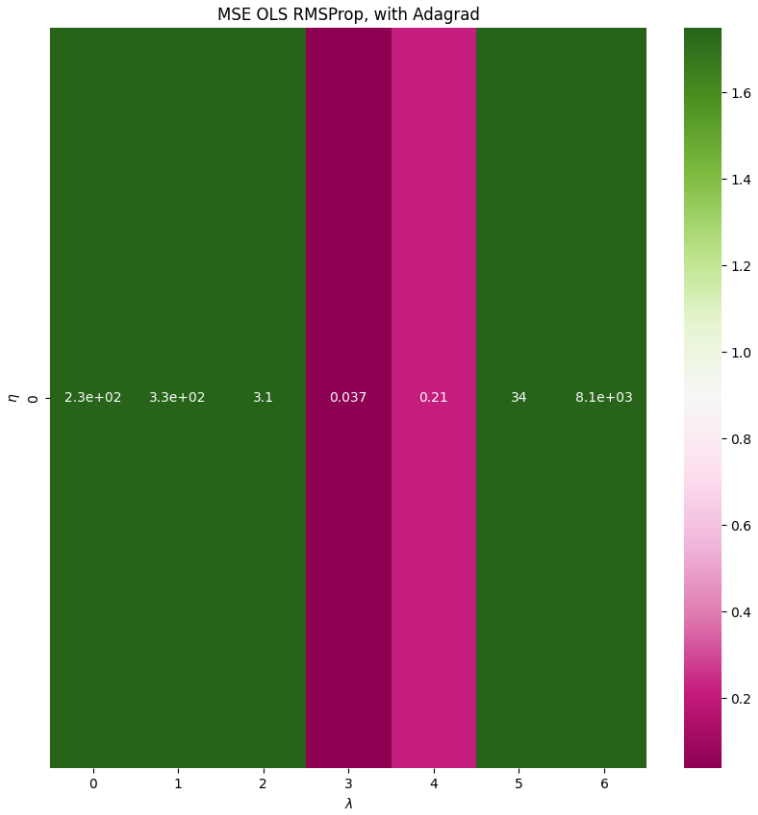

Figure 7: The results of the RMSprop algorithm using Adagrad.

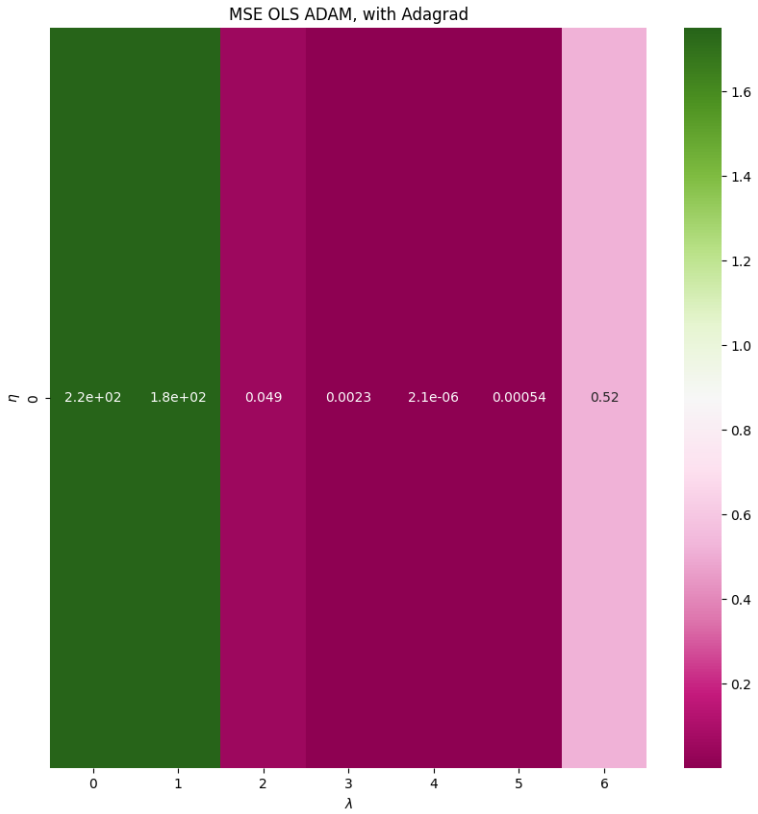

Figure 8: The results of the ADAM algorithm using Adagrad.

In figures 7 and 8 we see the results of the RMSprop and ADAM algorithms. Both of these algorithms were written using Adagrad. While the results for RMSprop are good, it's clear that ADAM has the best results out of every gradient descent method, more specifically for $\eta = 1e^{-1}$, with a mean squared error of $2.1e^{-6}$.

Let's now analyze the results obtained with Ridge regression.

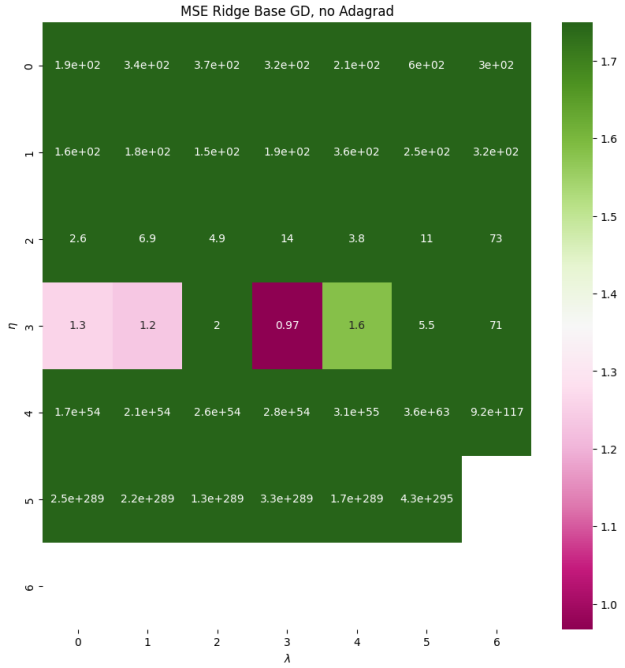

Figure 9: The Mean Squared Error for a base gradient descent without Adagrad.

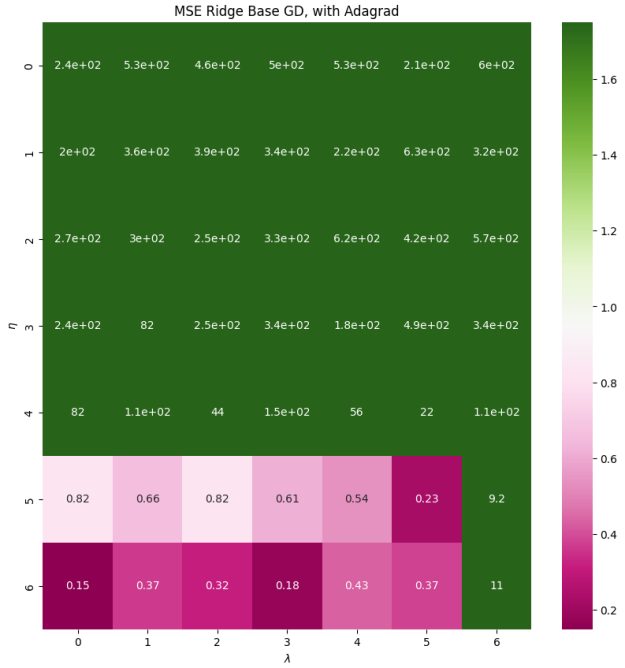

Figure 10: The Mean Squared Error for a base gradient descent with Adagrad.

In figures 9 and 10 we see the Mean Squared Error for a base gradient descent with and without Adagrad. As before we can see that the implementation with Adagrad is the best of the two, with MSE values as low as 0.15.

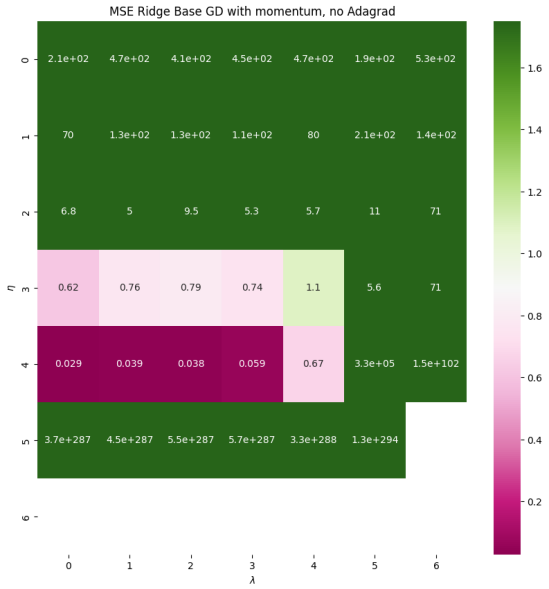

Figure 11: The Mean Squared Error of the base gradient descent using momentum with value 0.5, with Adagrad.

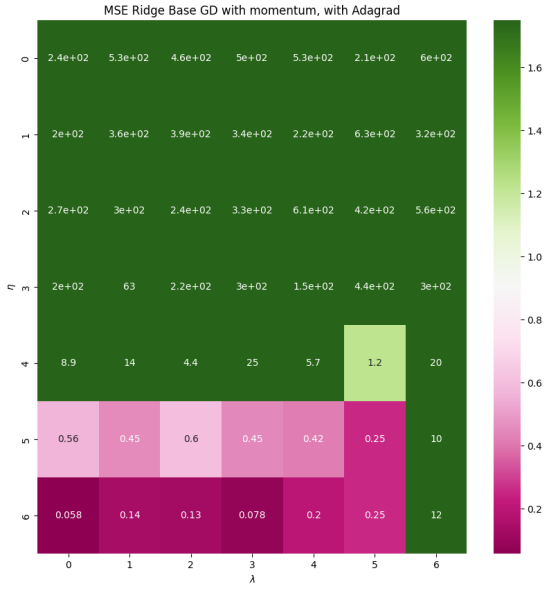

Figure 12: The Mean Squared Error of the base gradient descent using momentum with value 0.5, with Adagrad.

Figures 11 and 12 show the MSE of the base gradient descent using momentum with value 0.5, with and without Adagrad. As before, of the two the better one is the Adagrad implementation, with MSE values as low as 0.058.

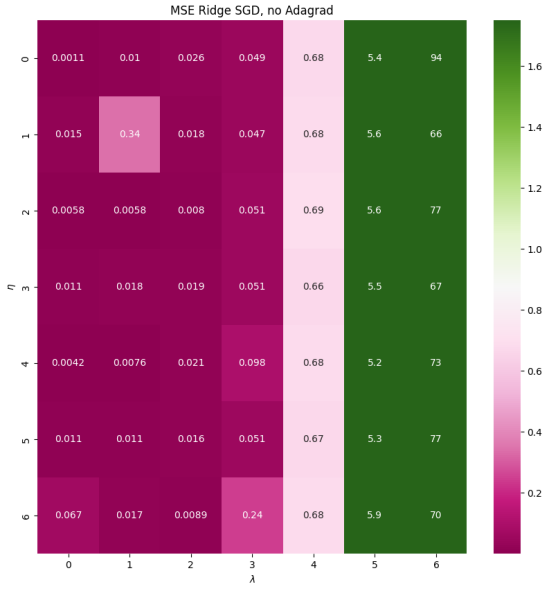

Figure 13: The results of a stochastic gradient descent, without momentum.

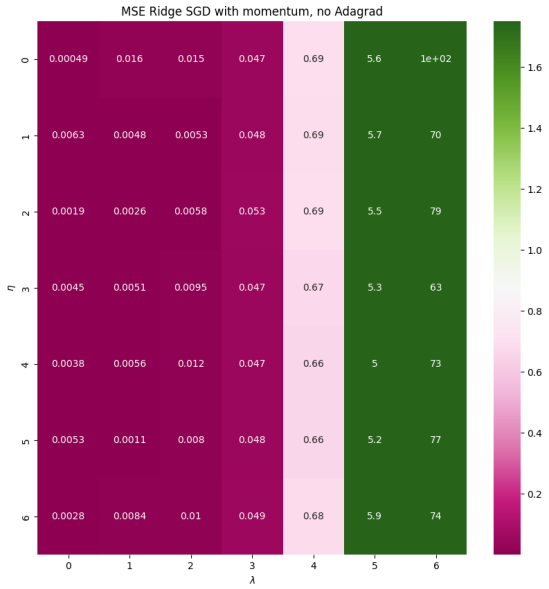

Figure 14: The results of a stochastic gradient descent, with momentum, using a momentum value of 0.5

In figures 13 and 14 we see the results of a stochastic gradient descent, with

and without momentum, using a momentum value of 0.5. As before, between the base and momentum implementations the better one is the momentum one.

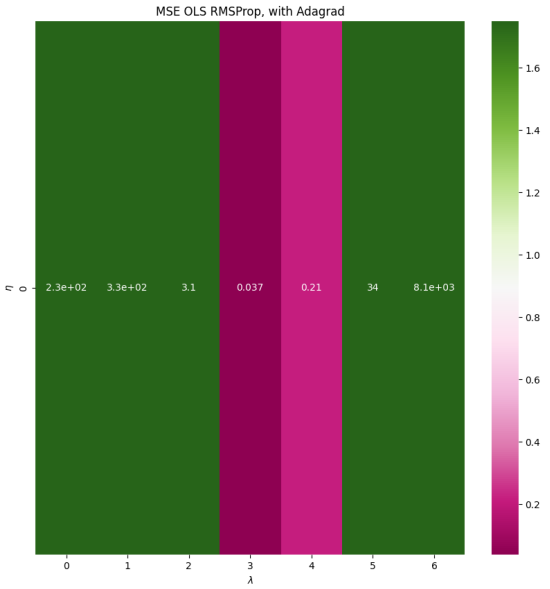

Figure 15: The results of a stochastic gradient descent, with momentum, using a momentum value of 0.5

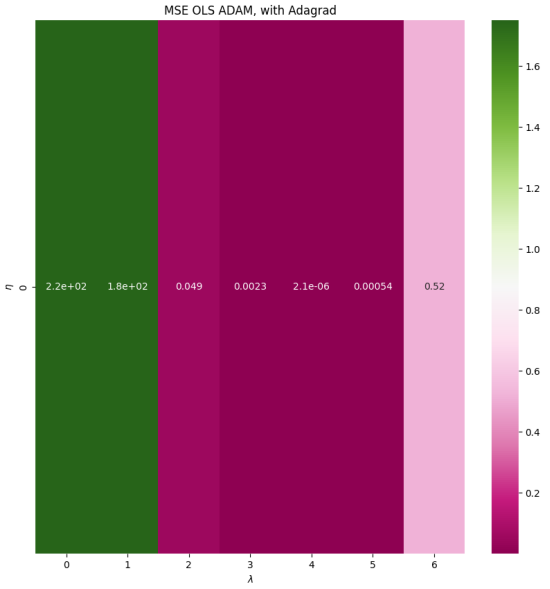

Figure 16: The results of a stochastic gradient descent, with momentum, using a momentum value of 0.5

Figures 15 and 16 show the results of the gradient descent using RMSprop

and ADAM. As seen when using OLS, while RMSprop is a good algorithm, the winner is clearly ADAM, with MSE values as low as $2.3e^{-5}$ for $\eta = 1e^{-1}$ and $\lambda = 1e^{-1}$.

3.2 Using a feed forward neural network to do predictions on synthetic Franke's function data

In this section we will show and discuss the results obtained from a FFNN used on Franke's function data. To calculate the MSE in the FFNN we used 7 λ and η values ranging from $1e^{-5}$ to $1e^{1}$. All of the results are static, as we used a seed to remove the randomness every time the FFNN is restarted.

Special thanks to Eric Reber and Gregor Kajda for developing the classes for the Cost, Activator and Gradient Descent functions.

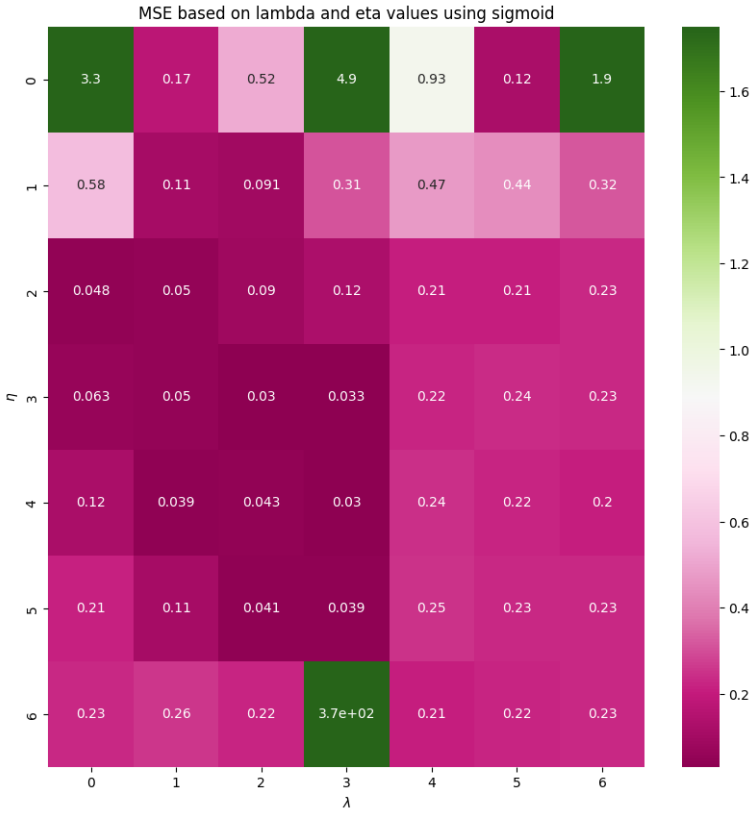

Figure 17: The MSE using the previously stated λ and η values using sigmoid as an activation function.

Figure 17 shows the MSE using the previously stated λ and η values using sigmoid as an activation function. We can see that we get pretty good error values overall, and the best is for $\lambda = 1e^{-2}$ and $\eta = 1$ with $error = 0.03$.

We also tried other activation functions, and we will now showcase the results that we obtained from each of them, and we then will give a final result on what activation function is best.

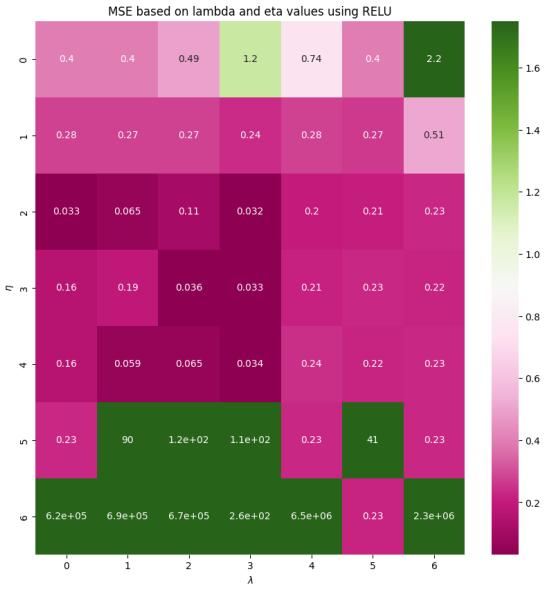

Figure 18: The MSE using the the RELU activation function.

When using the RELU activation function we get the lowest MSE value for $\lambda = 1e^{-2}$ and $\eta = 1e^{-3}$ with $error = 0.032$.

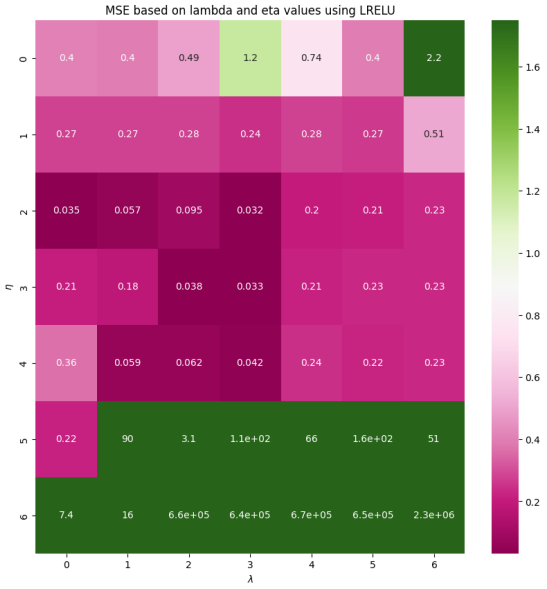

Figure 19: The MSE using the the LRELU activation function.

When using the LRELU activation function we get the lowest MSE value for $\lambda = 1e^{-2}$ and $\eta = 1e^{-3}$ with $error = 0.032$.

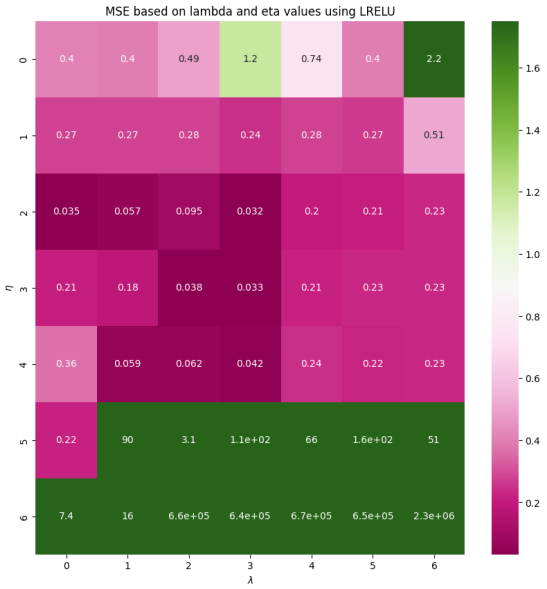

Figure 20: The MSE using the Softmax activation function.

When using the Softmax activation function we get the lowest MSE value for $\lambda = 1e^{-5}$ and $\eta = 1e^{-1}$ with $error = 0.027$.

Figure 21: The MSE using the Identity activation function.

When using the Identity activation function we get the lowest MSE value for $\lambda = 1e^{-5}$ and $\eta = 1e^{-3}$ with $error = 0.034$.

Taking into account the results obtained from the previous functions, we can conclude that for this classification case the best activation function is Softmax, while the worst is Identity, even if the difference in the minimum MSE compared with the sigmoid activation function is really low $(+0.004, -0.003)$.

3.3 Using a FFNN to classify Breast cancer cases

In this section we will discuss the results obtained when we try to classify Breast cancer cases. The dataset used was taken from the Wisconsin Breast cancer database, and contains cases recorded from 1989 to 1991 [**misc_breast_cancer_wisconsin_(original)_15**].

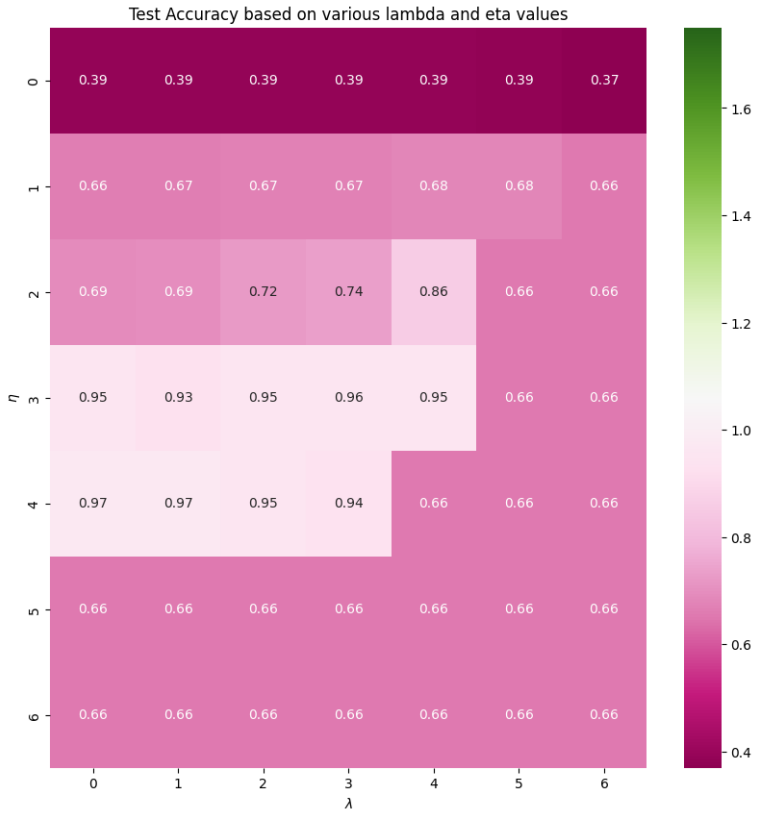

Figure 22: The plot of the accuracy score in function of the λ and η values, using a FFNN to classify Breast cancer cases.

Figure 22 shows a plot of the accuracy score in function of the λ and η values. We can see that there are 2 optimal values, where the accuracy is 0.97. Since the accuracy score is a generalization of the amount of true/false negatives and true/false positives we will show the Confusion matrix (a plot that shows the amount of cases for each true/false negatives/positives) for each of the 2 sets of values.

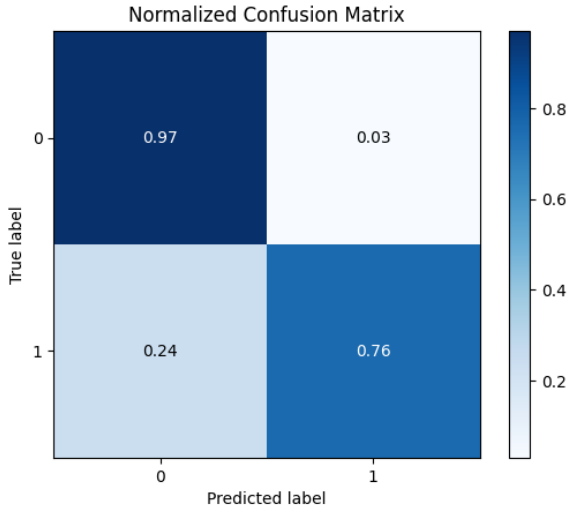

Figure 23: The normalized confusion matrix. The plot of the accuracy score in function of the $\lambda = 1e^{-5}$ and $eta = 1e^{-1}$ values, using a FFNN to classify Breast cancer cases.

We can see that for $\lambda = 1e^{-5}$ and $\eta = 1e^{-1}$ we get almost perfect recognition for the benign cases, while having bad recognition for the malignant cases, with a lot of false malignant classifications.

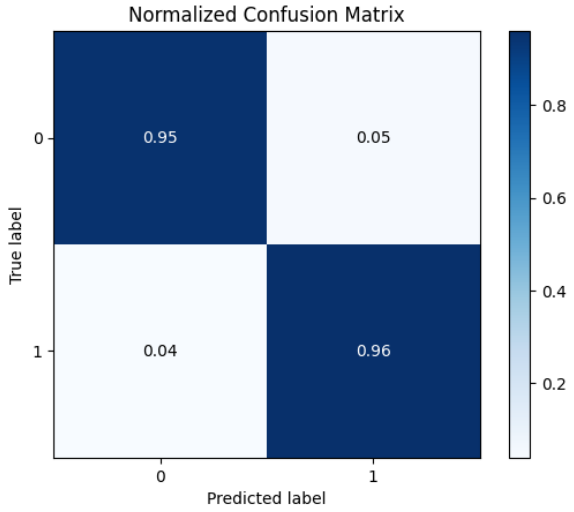

Figure 24: The normalized confusion matrix. The plot of the accuracy score in function of the $\lambda = 1e^{-4}$ and $eta = 1e^{-1}$ values, using a FFNN to classify Breast cancer cases.

Contrary to the last plot, for $\lambda = 1e^{-4}$ and $\eta = 1e^{-1}$ we get much better results. While the benign case recognition is slightly worse, the malign case recognition is greatly improved.

This shows that even if the accuracy score is high it doesn't always mean that the recognition rate is.

3.4 Using logistic regression to classify Breast cancer cases

In this last section of the results we will analyze the accuracy scores obtained by running a logistic regression on the Breast cancer dataset, in function of the η and λ values.

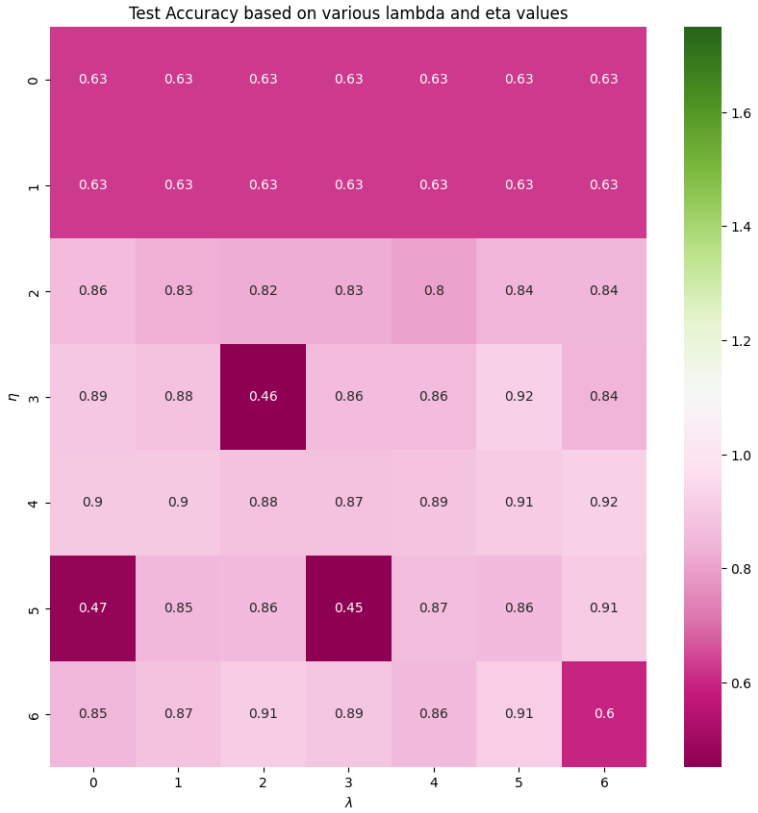

Figure 25: The plot of the accuracy score in function of the λ and η values, using a logistic regression to classify Breast cancer cases.

We can see that the maximum accuracy score value using logistic regression is 0.92, for $\eta = 1e^{-2}, \lambda = 1$ and $\eta = 1e^{-1}, \lambda = 1e^{1}$.

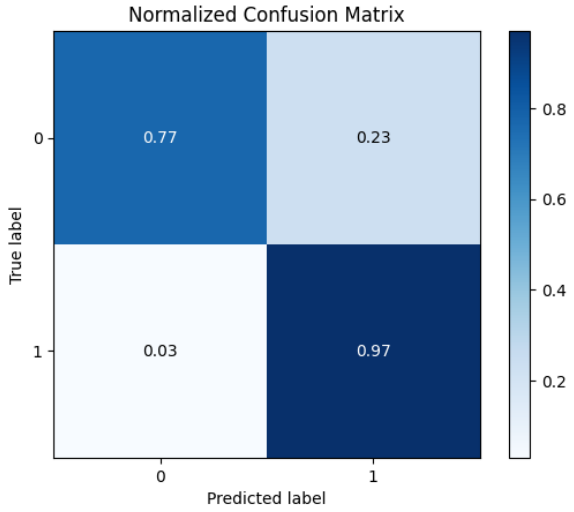

Figure 26: The normalized confusion matrix. The plot of the accuracy score in function of the $\lambda = 1$ and $eta = 1e^{-2}$ values, using a logistic regression to classify Breast cancer cases.

We can see that for $\eta = 1e^{-2}$ and $\lambda = 1$ we get good malignant case recognition, while getting a worse benign case recognition.

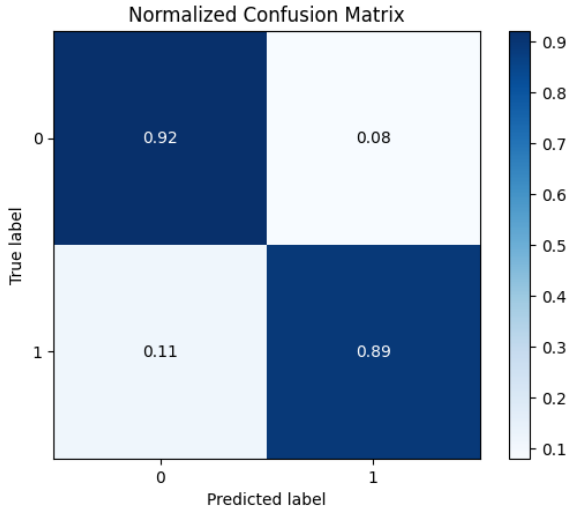

Figure 27: The normalized confusion matrix. The plot of the accuracy score in function of the $\lambda = 1e^{1}$ and $eta = 1e^{-1}$ values, using a logistic regression to classify Breast cancer cases.

In this case, with $\eta = 1e^{-1}$ and $\lambda = 1e^{1}$, we have much better benign case recognition, while having a worse malignant case recognition.

Overall the second case, with $\eta = 1e^{-1}$ and $\lambda = 1e^{1}$, has the better recognition, and is the best.

4 Conclusions

In conclusion, this project unravels a nuanced tapestry of gradient descent methods, showcasing ADAM's supremacy in minimizing MSE on both OLS and Ridge regression models. The FFNN emerges as a potent tool, particularly in the prediction of synthetic Franke's Function data. Softmax activation proves paramount, underscoring its effectiveness in contrast to other functions. Transitioning to real-world data, the FFNN's classification of breast cancer cases demonstrates nuanced recognition rates, further illuminated through Confusion Matrices. A parallel exploration of Logistic Regression unveils its comparable accuracy scores. This holistic examination accentuates the adaptability and efficacy of gradient descent methods and FFNNs, underscoring their utility in diverse contexts from synthetic function approximation to real-world disease classification.

5 Appendix

1. To get the code see: the code on Github repository
2. To get the source graphs images see: Project2/Images on Github repository

6 References

1. Wolberg, WIlliam. (1992). Breast Cancer Wisconsin (Original). UCI Machine Learning Repository. https://doi.org/10.24432/C5HP4Z.
2. Geron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems (2nd ed.). O'Reilly.
3. Bishop, C. M., Nasrabadi, N. M. (2006). Pattern recognition and machine learning (Vol. 4, No. 4, p. 738). New York: springer.
4. Murphy, K. P. (2013). Machine learning : a probabilistic perspective. Cambridge, Mass. [u.a.]: MIT Press. ISBN: 9780262018029 0262018020
5. Davison, A. C., Hinkley, D. V. (1997). Bootstrap methods and their application (No. 1). Cambridge university press.
6. Hastie, T., Tibshirani, R., Friedman, J. H., Friedman, J. H. (2009). The elements of statistical learning: data mining, inference, and prediction (Vol. 2, pp. 1-758). New York: springer.