UiO : Department of Technology Systems
University of Oslo

University of Oslo

FYS-STK

Project 1

Authors:
Øien B. Lasse

Supervisor:
Prof. Morten
Hjorth-Jensen

November 18, 2022

Contents

List of Figures

List of Tables

Abstract

The Report conducted an analysis and comparison of different gradient descent methods and neural networks. Two data sets were considered the Franke function for regression and Wisconsin breast cancer data for classification. From the Regression case the stochastic gradient descent with momentum and learning rate schedule faired best in cost scores. The neural network outperformed the cost scores of the best SGD utilizing the tanh activation function but did not manage to reproduce a satisfying image of the surface. For Classification, all activation functions performed well, with a range of 97-98% accuracy, whereas sigmoid had the highest average over several epochs. For classification, the self written neural net also performed better than sickit learns MLPClassifier.

Chapter 1

INTRODUCTION

Neural network is motivating, especially since it is the cornerstone of Skynet. The author of this report is highly motivated to go deeper into the neural net to gain knowledge on developing AI to counteract Skynet.

In this report, the main focus and motivation will be on neural networks, and to understand this, we need to go deep into the world-famous black-box neural net. A Neural network is, in many cases, similar to the linear regression introduced in project one. Where the idea is to find optimal parameters such that a model can represent our data. However, the neural net is the model with a cost function of your choosing, and the parameter being optimized is the weights and biases between each neuron.

Neural networks are especially good when it comes to doing automatic tasks, as well as standard tools to use in forecasts. From the author's expertise, neural networks are crucial when developing digital twins of power production plants and facilities. The hope is that this report will provide insight into the common overarching theme of neural networks that the author can use in his master thesis.

The overarching research question this report seeks to answer is as follows:

- Deemistifying neural networks and their functionality
- can Neural networks describe a topological surface and predict Malignant tumors?
- How does different activations and initializations of the neurons and pathways affect the results?

Chapter 2

THEORY

This chapter aims to introduce the theory behind the solution methods used. Although there are some scary concepts and mathematics, the author quotes the great mathematician John Von Neuman "Young man, in mathematics, you don't understand things. You just get used to them". With this quote, the author has attempted to describe these concepts and math involved through this chapter.

2.1 Regular and stochastic gradient descent

This section introduces the theory behind various gradient descent algorithms within machine learning. More in-depth on plain gradient descent and stochastic gradient descent methods.

Compared to project one, where optimal beta was calculated, the idea behind gradient descent is to have an algorithm that finds and updates the parameters. The different (GD and SGD) can be viewed in figure(2.1)

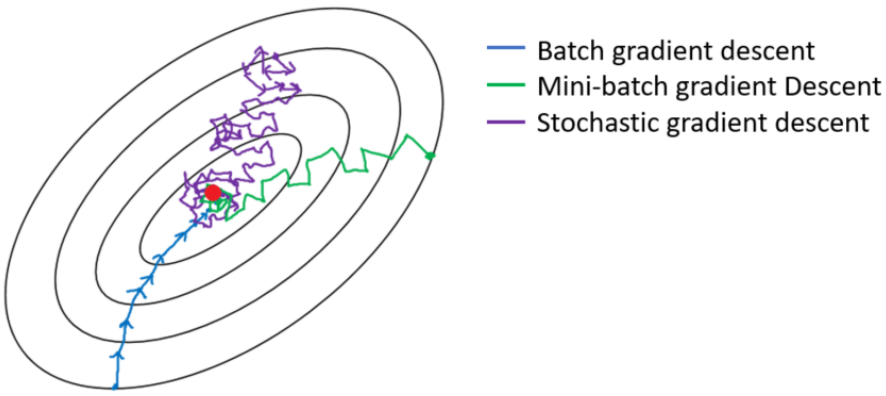

Figure 2.1: Illustration of different GD methods

2.1.1 Batch gradient descent & momentum.

The gradient descent algorithm is essentially updating the parameters beta. The assumption made that there exist a function $F(\boldsymbol{x})$ the idea is then to reach a minimum

of this function. This is achived by stepping along the negative gradient $-\nabla F(\boldsymbol{x})$ by picking a position $\boldsymbol{x}$ in said direction. The gradient descent steepest descent is calculated as follows.

$$\boldsymbol{x_{k+1}} = \boldsymbol{x_k} - \eta_k \cdot \nabla F(\boldsymbol{x_k}) \tag{2.1}$$

Where η_k is the stepp length or learning rate, and observed from the formula for small enough values of $\gamma_k > 0$ then $F(\boldsymbol{x_{k+1}}) \leq F(\boldsymbol{x_k})$. This implies that a minimum is achieved.

As introduced in project one, the cost function can be written as the sum over n data points.

$$C(\theta) = \sum_{i=0}^{n} c_i(X_i\theta_i) \tag{2.2}$$

By implementing the idea of gradient descent introduced in equation(2.1), the gradient now is the sum of each point:

$$\nabla_\theta C(\theta) = \sum_{i=0}^{n} \nabla_\theta c_i(X_i\theta_i) \tag{2.3}$$

One problem with gradient descent is what happens when the gradient approaches a ravine, aka local minima. Or when the gradient approaches the minima, it can cause slow osculations approaching it. Illustrated in figure (2.2).

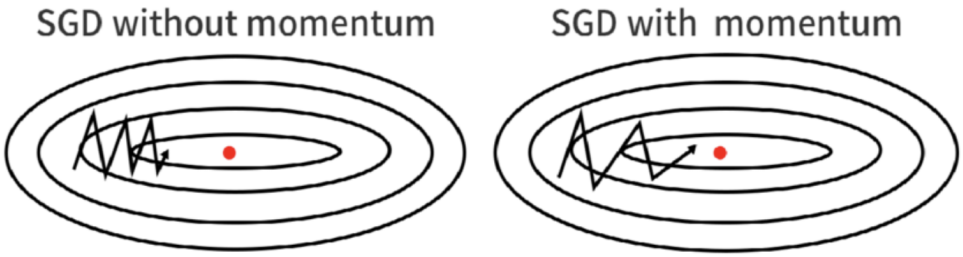

Figure 2.2: Momentum SGD & Non momentum SGD

The figure illustrates that by adding a momentum the appraoch with momentum uses fewer oscilations and approaches the minima faster. This is due to the momentum term serves as a memory term for the direction in the parameter space. The memory term is calculated as follows.

$$v_t = \gamma v_{t-1} - \eta_t \nabla_\theta E(\theta_t) \tag{2.4}$$

here γ is the momentum parameter $0 < \gamma < 1$ and η is the learning rate introduced earlier. The parameter updates are given by:

$$\theta_{t+1} = \theta_t - v_t \tag{2.5}$$

Even though 2.2 is taken for stochastic gradient descent, the momentum algorithm works well on the simple gradient descent algorithm.

The general idea behind these algorithms is

2.2 Stochastic gradient descent

In the previous section, the gradient descent was in focus. this section will go more in-depth on the stochastic gradient descent (SGD). The SGD in this paper is a combination of the mini-batch and stochastic in figure(2.1).

The general idea behind SGD is to divide the data points n into smaller batches or uniform-sized mini-batches. The number of batches is given by data points n divided by the batch size. Each batch is represented as B_k for k number of batches. This allows us to approximate the gradient over the sum of data points in one mini-batch picked randomly for each gradient calculation.

$$\nabla_\theta C(\theta) = \sum_{i \in B_k}^{n} \nabla_\theta c_i(X_i \theta_i) \tag{2.6}$$

The good part of this algorithm is that the randomness reduces the chance of being stuck at local minima, and for sufficient batch sizes, the calculation becomes much cheaper. The iteration over one minibatch is commonly referred to as an epoch.

From project one, it was discussed overfitting and underfitting in the form of the complexity of the model. In relation, SGD can be over and under-fitted concerning the size of each minibatch and the number of epochs. Therefore these parameters should be tuned accordingly when applying SGD to a problem.

2.2.1 Learning rates & uppdates

Stochastic gradient descent is commonly used as the optimizer or solver for weights and biases in a neural network. But as the gradient descent the stochastic also have the parameter η (learning rate), which must be tuned.

To tune the learning rate, there are some methods involved. One method would be to run many tests for each learning rate and pick the one that seemed to have the best fit. This would be time-consuming, not to mention a lot of gradient calculations. A fixed learning rate, no matter how good it is, still raises the question if it is good enough to reach minima within the number of epochs.

One commonly used method is to have a timed scheduled update of the learning rate where the learning rate is decayed by a decaying hyperparameter D_k multiplied by the iteration at timestep t.

$$\eta_t = \frac{\eta_{t-1}}{1 + D_k \cdot epoch_t} \tag{2.7}$$

For small values of $D_k << 0$ the learning rate or step length will be larger at first and decay as the epoch iteration increases. Also, if the decay rate is zero, then the learning rate is at a fixed position.

Adagrad, RMS propagation, and ADAM

The previously mentioned method is relatively standard, but there are introduced other algorithms that try to solve the problem with an unpalatable learning rate.

RMS propagation, like the momentum introduced earlier the RMSprop, keeps track of the running average first momentum. But also keeps track of the second momentum. The parameter update now becomes.

$$\theta_{i+1} = \theta_i - \eta \cdot \frac{\boldsymbol{g_t}}{\sqrt{\boldsymbol{s_t}} + \epsilon} \tag{2.8}$$

where s_t is the second moment of the gradient g_t I.E $s_t = E(g_t^2)$

The next method to consider is ADAM; ADAM is a combination of the RMS-prop and the momentum method. It keeps a running average for the gradients' first and second momentum. And the update rule becomes:

$$\theta_{i+1} = \theta_i - \eta \cdot \frac{\boldsymbol{m_t}}{\sqrt{\boldsymbol{s_t}} + \epsilon} \tag{2.9}$$

Here m_t is the first-order momentum and s_t is the second order momentum. In addition, the ADAM calculates a bias correction step to account for the assumptions made for the first two momentums. More about the ADAM can be viewed in the world-famous paper [1].

2.3 Neural Networks

Neural networks are arguably today's most famous black box. Countless papers have been written using machine learning, deep learning, and neural networks to solve and figure out problems that seemed unsolvable.

The neural network got its name as it resembles neurons and pathways, much like the human brain. In project one, we showed through mathematics that one could fit data points to a sloped surface through polynomial fitting (a type of squiggle). Essentially this is also one of the aims of a neural network, fit an analytical squiggle to a lot of data points.

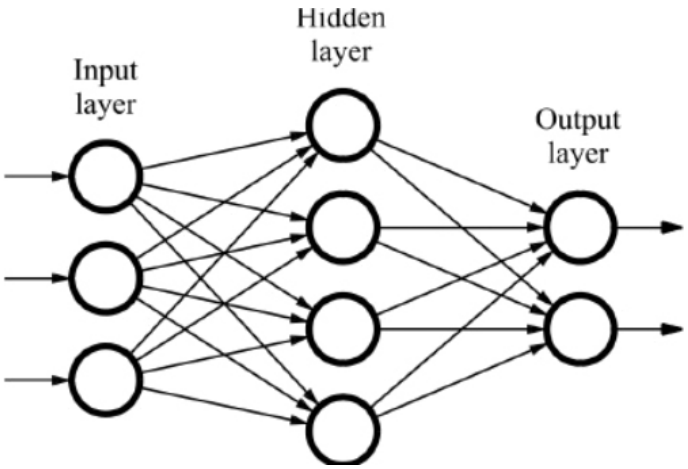

Figure 2.3: Neural Network architecture

Figure (2.3) illustrates the typical architecture of most neural networks. Each circle is a node/neuron, and the intersecting lines are the pathways. The architecture

consists of an input layer, where the data is fed to the neural network. From there, it is multiplied with specific weights and biases toward the hidden neuron. The hidden neuron contains an activation function and then outputs it to the output neurons with weights and biases. The weights and biases are calculated through what is known as the back-propagation algorithm.

The most common neural network, as illustrated in figure2.3) is a feed-forward architecture. The feed-forward architecture restricts data flow from input towards output and layer n-1 towards layer n and not backward. Giving each node an activation function, the input in node n in layer m is then the weighted sum of all the nodes in layer m-1. The nodes activation can then be calculated as:

$$a = f(\sum_{i=0}^{n} w_i \cdot x_i + b_i) \tag{2.10}$$

Where w_i are the weights and b_i are the biases, x_i represents the output from node i from the previous layer m-1.

2.3.1 Hidden layers & activation functions

Activation functions are some arbitrary continuous function that resides in the hidden neurons. For a simple illustration, the input layer provided an x-coordinate multiplied by a weight and shifted with a bias. Then the activation function takes this modified x-coordinate as its x-coordinate and outputs the y-coordinate. I.e., the value of the activation function for a given x. From project one, where the complexity of the regression was discussed in the form of polynomial degrees, the complexity of neural networks is given by the number of hidden layers and hidden neurons.

There are many activation functions, and all have advantages and disadvantages. In this paper, the activation functions that are tested are as follows:

- Sigmoid
- tanh
- RELU
- LeakyRelu (LRELU)

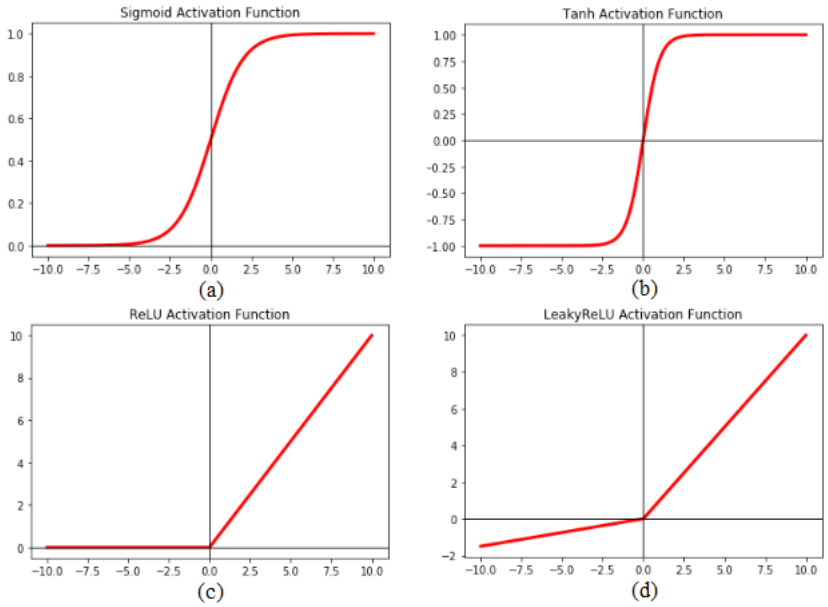

Figure 2.4: activation functions taken from [2]

The sigmoid function outputs a number between zero and one; more positive inputs lead closer to one, and more negative inputs lead closer to zero. The sigmoid function is commonly used for logistic regression.

$$\sigma_{sigmoid} = \frac{1}{1 + e^{-x}} \tag{2.11}$$

The tanh function shares some similarities with the sigmoid function, like the s-like shape of the function to a bounded input range that helps with the exploding gradient problem. An important feature that differentiates them lies in the gradient of the function observed in figure (2.5)

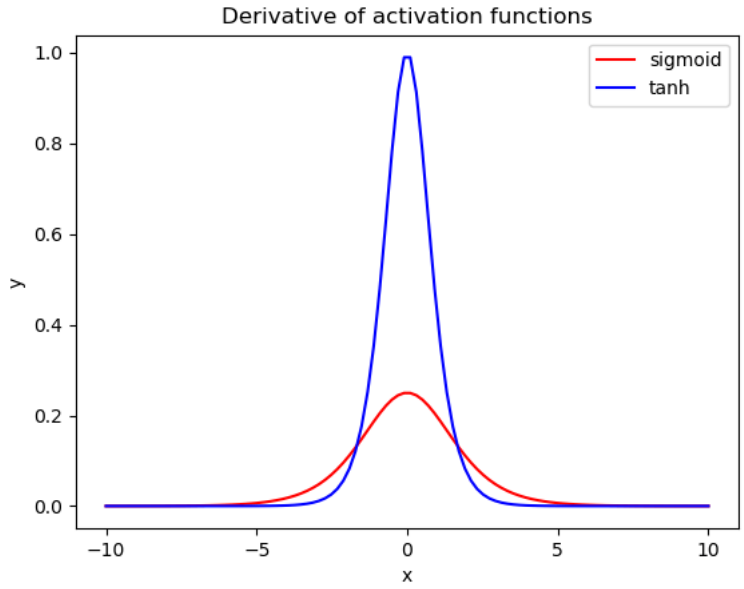

Figure 2.5: Derivation of tanh and sigmoid gotten from [3]

The figure shows that the tanh function's gradient is "pointier" and larger around zero than the sigmoid. This is interpreted as bigger updates on the weights and biases. The activation function is given as follows:

$$\mathbf{tanh}(\mathbf{x}) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.12}$$

One problem in the previously introduced activation functions is the vanishing gradient. This occurs when the weights and biases are updated through the back-propagation algorithm presented in the following subsection. Essentially, the error is so close to zero for the first layers when back-propagated, resulting in the weights and biases for the first layers being inaccurately updated and stopping the forward feeding of the network.

The following two functions deal with the vanishing gradient problem are the ReLu and LeakyReLu functions. The ReLu stands for rectifying linear units, and despite its name, it can learn nonlinear functions. Pros of this function are as mentioned dealing with the vanishing gradient and being less computationally heavy compared with the sigmoid. The ReLu is given as follows.

$$Relu(z) = max(0, z) \tag{2.13}$$

The Relu can blow up the gradient and also cause of neurons death, by setting the weights such that the function never will be activated. To fix the dying Relu problem, the leaky ReLu was constructed.

$$LReLu(z) = max(\alpha \cdot z, z) \tag{2.14}$$

As observed from the figure (2.4) and the equation above, the leaky ReLu does not set the gradient to zero for negative inputs. Instead, it allows a small contribution from the parameter alpha.

2.3.2 Weights and biases

As discussed, the different activation functions have their ups and downs. And initializing weights becomes an issue or a solution to deal with each activation function's shortcomings.

The Xavier initialization[4] for sigmoid and tanh functions is commonly used and will be used in this report as well.

$$\mathcal{N}(0, \frac{1}{n^{l-1}}) \tag{2.15}$$

The idea is to prevent the variance from increasing exponentially when passed from layer to layer, the equation sets a uniform variance I.E $= var(a^L) = var(a^{L-1}$.

Another choice is the HE initialization as it seeks to deal with the dying RELU problem. [5]

$$\mathcal{N}(0, \frac{2}{n^{l-1}}) \tag{2.16}$$

As it aims to to reduce the likelihood of the exploding gradient problem.

2.3.3 back propagation algorithm

As introduced earlier in this chapter, back-propagation is used to update the weights and biases of the neural network. To accomplish this, there must be a cost function that the back-propagation tries to minimize or evaluate the outputs of the feed-forward pass. Taking some arbitrary cost function and relating it to δ^L, the gradients for weights and biases are then calculated as:

$$\frac{dC}{dw^L_{jk}} = \delta^L_j \cdot a^{L-1}_k \tag{2.17}$$

and for the biases:

$$\frac{dC}{db^L_{jk}} = \delta^L_j \tag{2.18}$$

The δ^L full explanation will not be provided in this report. However, the final compact expression will be provided. For further details, chapter 2 in the Nielsen's book[6]will cover the complete derivation and an intuitive explanation.

$$\delta^L = \nabla_a C \odot f`(z^L) \tag{2.19}$$

Where ∇C is the gradient of the cost function describing the rate of change of C with respect to the activation, the weights and biases are then updated through gradient descent algorithms.

2.4 Logistic Regression

Logistic regression is essentially a Neural network with no hidden layers and neurons. This section will be brief as the theory is described in more detail for gradient descent methods and neural networks. As previously mentioned is the basis for the Logistic regression algorithm.

Chapter 3

METHOD

all code implementation can be viewed in the GitHub.

3.1 Data description

Since the Franke function was introduced in project one, this section will aim to introduce the Wisconsin breast cancer data set.

The dataset for Wisconsin breast cancer is a binary classification case with features that would result in benign or malignant cancer. Whereas 63% is benign and 37% is malignant, meaning if guessed all benign, the accuracy with the defined accuracy score would be 63%. The features are as follows:

- radius (mean of distances from center to points on the perimeter)
- texture (standard deviation of gray-scale values)
- perimeter
- area
- smoothness (local variation in radius lengths)
- compactness ($perimeter^2/area - 1.0$)
- concavity (severity of concave portions of the contour)
- concave points (number of concave portions of the contour)
- symmetry
- fractal dimension ("coastline approximation" - 1)

Some features have a significant covariance, and thus can be removed to reduce the problem's dimensionality.

3.2 GD and SGD

The stochastic gradient descent is used for the regression on the franke function; this section will give a brief explanation of the procedure.

The first procedure was to implement the gradient descent on the franke function, utilizing the eigenvalues of the Hessian matrix as the standard learning rate. Then by adding momentum to the gradient descent, compare both mse and convergence criteria over a range of polynomial degrees.

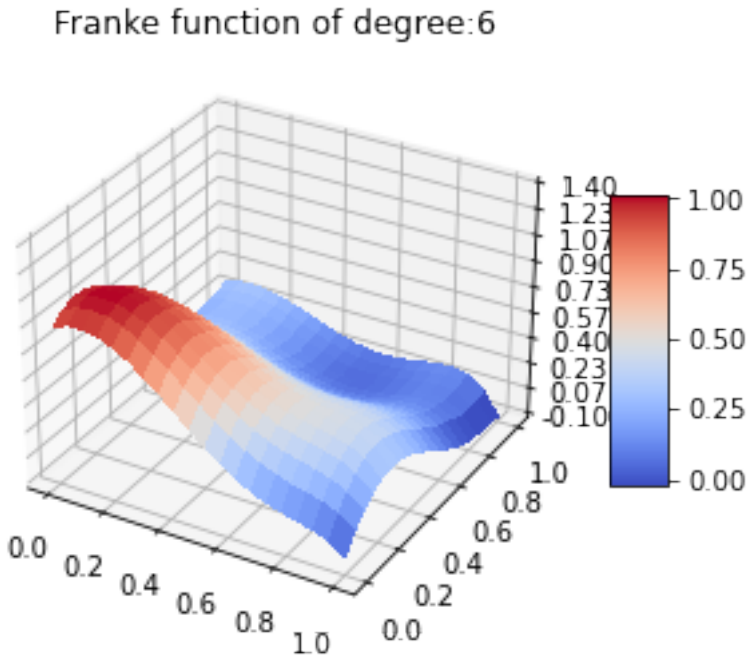

Figure 3.1: Franke function of momentum based GD

Finding that polynomial degree, 6 had the best combination of low computational time and low MSE; the next step was then to investigate the influence of γ as well as the learning rate for the momentum-based GD. It was found using the Gradient descent with the momentum that parameters $\gamma = 0.9$ and $\eta = 0.1$ gave the best MSE, but at the cost of more iterations to converge.

Concluding the investigation done on the gradient descent, it was observed for a polynomial degree of 6 that the change in gamma had a low impact on the MSE but a vastly more significant impact on the convergence criteria. I.E one change in gamma for the same learning rate gave negligible result changes in the cost function. But could yield significant differences in the convergence criteria.

The significant findings of comparing GD with and without momentum led to only implementing the momentum variation for the stochastic gradient descent methods, then comparing them with the other methods mentioned in the theory chapter.

Further investigation of the SGD was conducted in the following manner:

- Compare MSE and R2 for initial learning rate and another parameter (ridge,lambda and eta).
- Compare the MSE and R^2 score of all the methods with selected optimal parameters over a range of epochs.
- Compare each method individually with the number of epochs and mini-batch size.

All findings, including the figures and heatmaps for every method, can be viewed in the GitHub folder. The initial search was conducted for 100 number of epochs and a batch size of 20 data points. Mainly it was decided for stochastic gradient descent with a scheduler that a decay rate of 1e-5 was the best choice and a gamma value equal to 0.9. The Delta rate for RMS, Adam, and adagrad was set to 1e-5 as it yielded the best results.

Given more time and in the project and as inspiration for project 3, the author wants to investigate all the hyperparameters and their effect on cost functions.

3.3 Neural Network

To fully utilize the backpropagation algorithm introduced in chapter 2 of this report, a cost function still needed to be defined to evaluate the feed-forward pass. For this report, there was decided upon the cross entropy for binary classification with a sigmoid or tanh at the output layer. The regression part does not need activation of the output layer but the mean has to be taken if selected output neurons above one, according to Nielsen's book[6]. The cost function requirement for back propagation is that the cost function can be written as an averege.

The cross-entropy for binary classification is given as follows:

$$entropy = -(ylog(p) + (1-y)log(1-p)) \tag{3.1}$$

Where y is the binary yes-no variable I.E 0 or 1, while p is the predicted probability of the observed variable. Concerning neural networks and the equations describing the backpropagation in chapter 2, the cross entropy can be given as.

$$C = -\sum_{i=1}^{n}[t_i log(a_i^L) + (1-t_i)log(1-a_i^L)] \tag{3.2}$$

Where now, the probability is given by the activation of a neuron i inside layer L, while t_i represents the target variable—taking the derivative for the activation function yields.

$$\frac{dC}{da^L} = \frac{a^L - t_i}{a^L(1-a^L)} \tag{3.3}$$

As stated earlier in this section, the output layers need an activation function when doing classification. This report will focus on tanh and sigmoid as output functions for classification.

$$\begin{aligned} Sigmoid = f'(z) &= f(z) \cdot (1 - f(z)) \\ tanh = f'(z) &= 1 - f(z)^2 \end{aligned} \tag{3.4}$$

The derivative of the activation function is then put together with the final equation for δ^L:

$$\delta^L = f'(z) \odot \frac{dC}{da^L} = (a^L - t) \tag{3.5}$$

Previously from project one, there was a small discussion on over-fitting. Regarding neural networks, this can come from hidden layers and networks To counteract over-fitting, we take the same observations from project one, where one could increase the number of data points or add the L2 regularization. The cross-entropy with L2 Regularization then becomes.

$$C = -\sum_{i=1}^{n} [t_i log(a_i^L) + (1 - t_i) log(1 - a_i^L)] + \frac{\lambda}{2n} \sum_{i=1}^{n} w^2 \tag{3.6}$$

The parameters in focus are the lambda and learning rate parameters, as well as number of hidden layers and neurons.

As a test run and verification of functionality, a simple function $y = x^2$ was applied with 200 evenly spaced x values to ensure that the neural network fits the shape and peaks of the curves given.

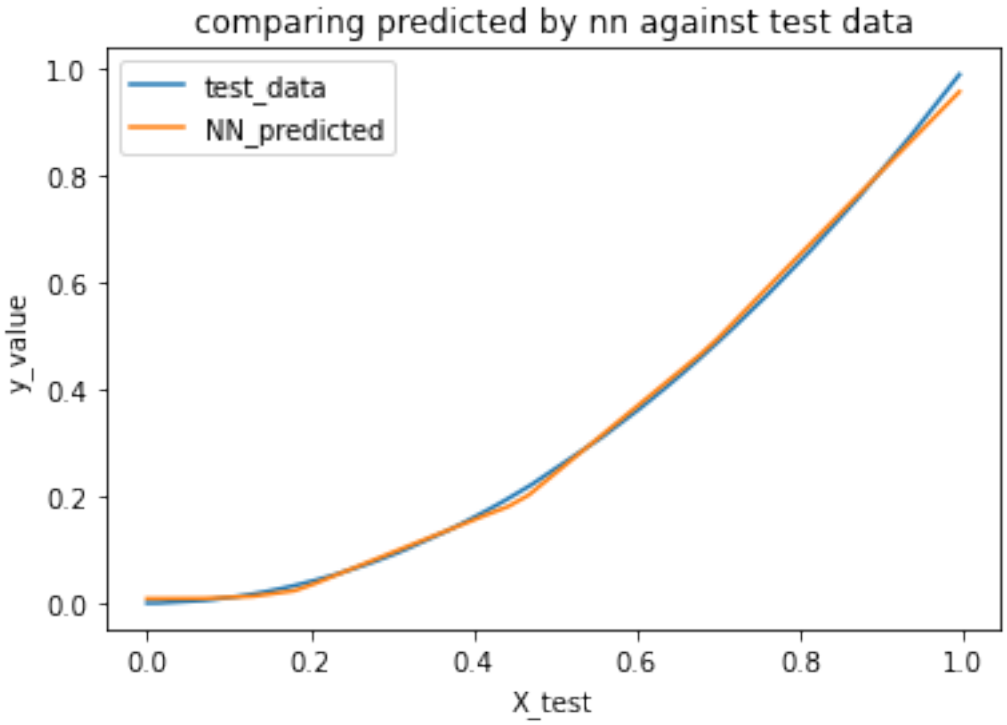

Figure 3.2: test run to confirm neural network. functionality

Further on the Frankes function and the classification the following method is applied:

- Tune Learning rate and lambda, for all activation function with weight initialization.
- Tune Hidden nodes and hidden layers
- Compare Different activation functions and weight initialization
- compare convergence over the number of epochs
- For classification case, also compare output functions

For the Frankes Function with a batch size of 10, the maximum allowed learning rate becomes 0.001 without encountering overflow errors, but to ensure the code runs for a more extensive range. It has been implemented that the output returns an array of zeros with the same size when exploding/vanishing gradients occur. Thus allowing for flexibility in the analysis.

Chapter 4

RESULT/Discussion

This cahpter will provide the results and discussion of them, as well as a final comparison for the classification and regression

4.1 Regression

This section will provide the final results and highlight the best results from each case. The tuning of the parameters can be viewed in the GitHub link, as well as more figures of different heatmaps for SGD as well as a comparison of the self-built neural net compared to sickit learn.

4.1.1 Ridge SGD

Investigating how lambda and eta affected the different cost scores on the frankes function. Resulted in optimal values are given in the table below

Table 4.1: Hyper parameters for ridge

Algorithm	η	λ
ADAM	0.1	0.0025
Adagrad	0.1	0.0025
RMS	0.1	0.0025
Momentum	0.1	1e-5

The Learning rate across the board seems to be optimal, with a value equal to 0.1. Similar findings were found using sickit-learn SGD regressor with l2 penalty. With these values, the following results were observed.

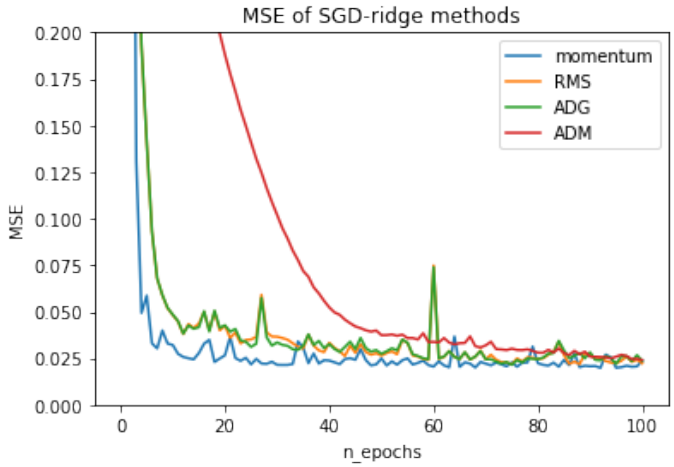

Figure 4.1: MSE for different methods

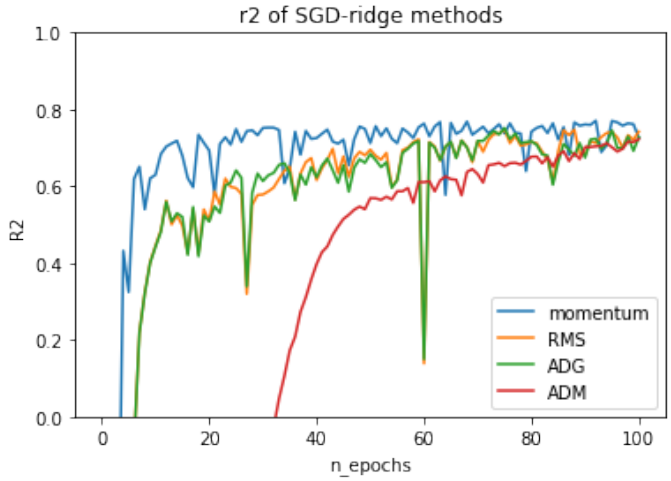

Figure 4.2: R^2 of different methods

Observed from the figures above, it is clear that ADAM converges slower than the other methods; this may have to do with the parameters ρ_1&ρ_2 or the small parameter δ. Although the convergence rate is slower than the other methods, the curve is smoother.

Noteworthy, the best MSE and R2 Score was found with momentum using the learning rate scheduler. This value is as follows.

- MSE: 0.019
- R^2 : 0.78

Size of mini-batches and number of epochs

As mentioned in the theory chapter number of epochs and the size of mini-batches are hyperparameters that are related to overfitting. contrary to the previous section will keep the optimal lambda and eta and grid search for the number of mini-batches and epochs.

Similar trends were observed in this analysis; the momentum with a learning rate scheduler converged faster and yielded the best results. The ADAM had slow convergence but yielded similar results. Both Adagrad and RMS-prop had somewhat spiky behavior. In conclusion, the momentum seems to best describe the Franke function with a learning rate scheduler, given the fixed parameters for decay, delta, and gamma in the code.

Since the momentum with a learning schedule describes the Franke function best, comparing it to project one, it outperforms not only in test scores but also in smoothness and form. See GitHub for proof. The influence of the size of mini-batches and will be given for momentum, but the rest can be observed in GitHub.

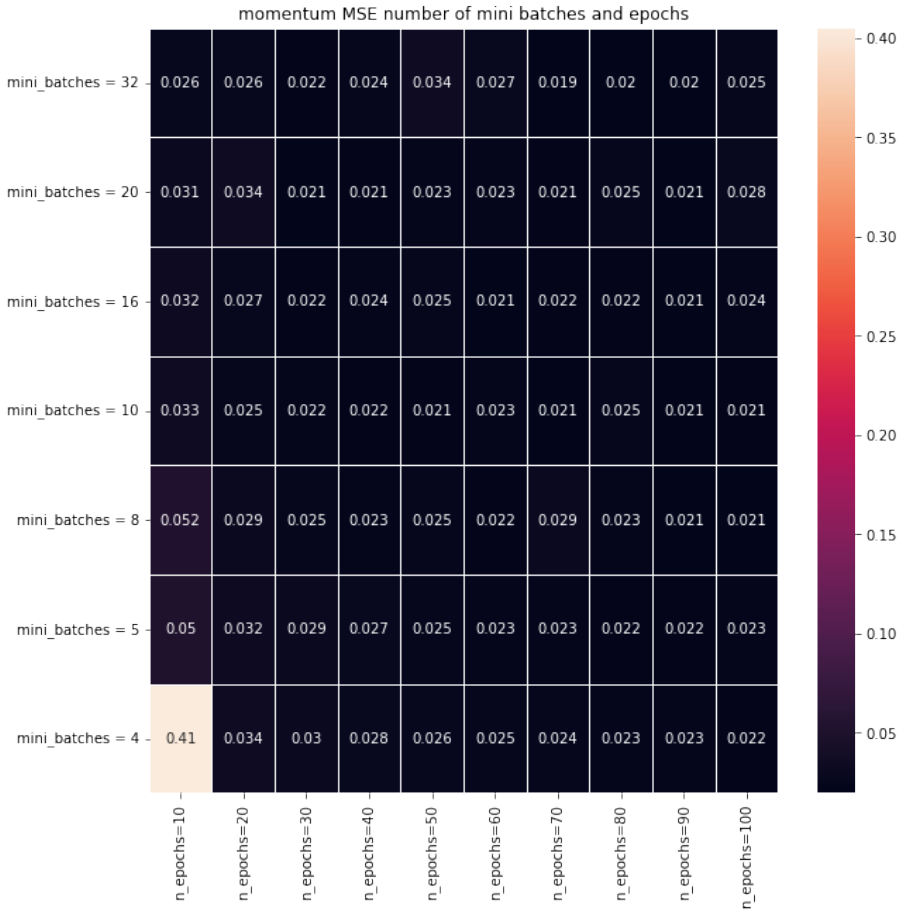

Figure 4.3: Mini batches vs. number of epochs

It is clear from the figure that a samaller MSE could be achieved with 70 epochs and a batch size of 10 i.e. 32 mini-batches.

4.1.2 OLS SGD

The findings in this section are evident with the parameters highlighted in the method chapter. Similar behavior was observed with the OLS as with Ridge. From the figures it is observed a slower convergence rate for Adam; the lack of the L2 regularisation that is provided in ridge also makes for a more unstable progression.

In light of these results, it is clear that describing the Franke function stochastic gradient descent with momentum is preferred. However, Adam is a highly credited method for adaptive learning rate. Its performance was not favorable in this research. This has been observed through several papers stated in [7]. Stating what was observed here, even though training time with adam is superior, it does not converge to the optimal solution for many problems as SGD with momentum does.

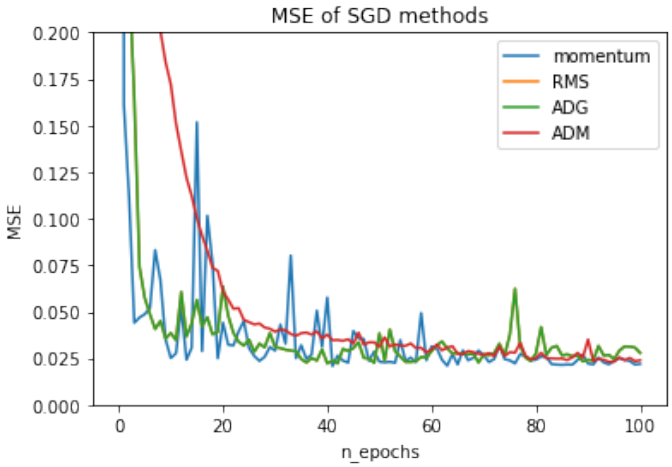

Figure 4.4: OLS MSE Comparison of methods

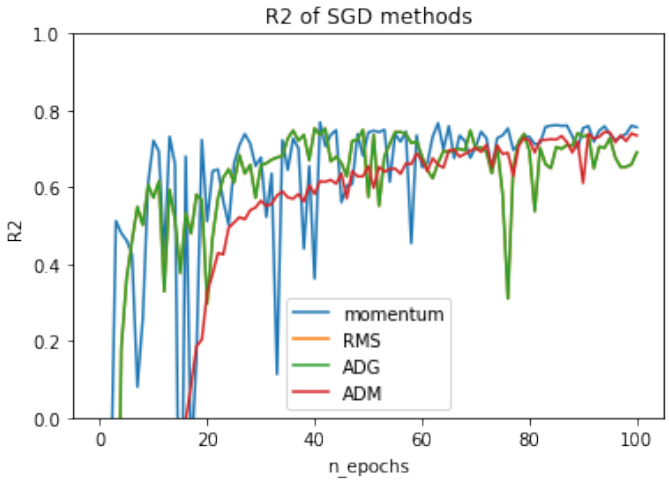

Figure 4.5: OLS R2 Comparison of methods

The best performance found of OLS is with momentum SGD and is as follows:

- MSE: 0.022
- R^2 : 0.75

4.1.3 Neural network results

The Neural network performed well; the author was surprised with the tuning done on the neural network. The best result was yielded from the tanh activation function for the regression case.

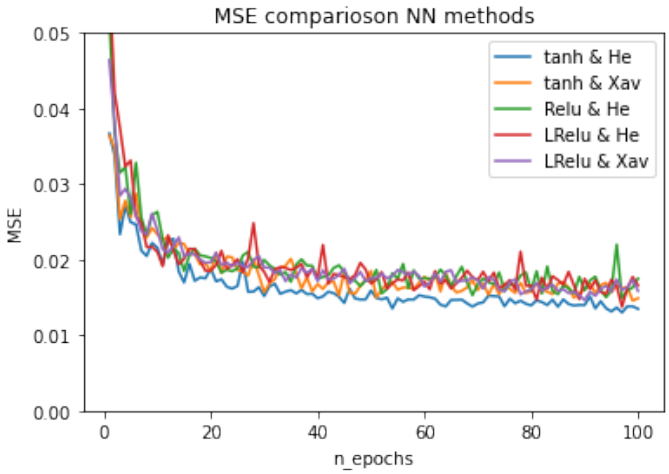

Figure 4.6: own nn Regresion MSE

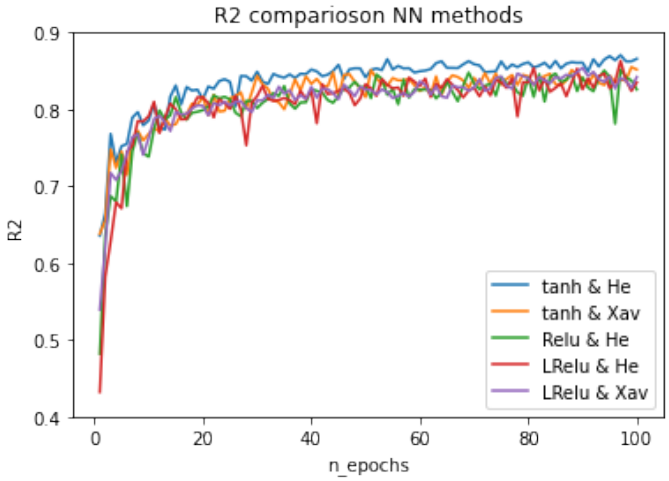

Figure 4.7: own nn regresion R2

Observed from the figures tanh, with the "He" weight initialization, performs the best after tuning at 40 neurons and 4 layers it gave scores of $R^2 = 0.87$ and $MSE = 0.012$, which is better than both SGD and GD. And a vast improvement when compared to project 1. Although the significant drawback is that the neural net has not successfully reconstructed the shape of the Frankes function when fed raw data.

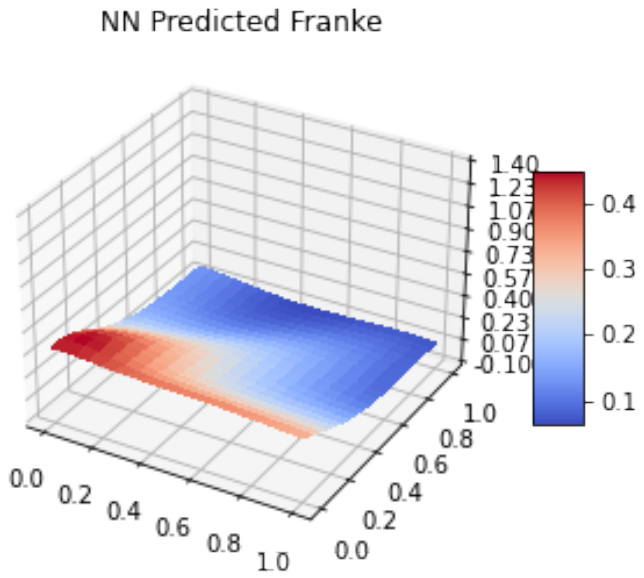

Figure 4.8: Franke function predicted by NN

The Neural network does find the correct coloring, as well as some curvature, but when comparing to what the SGD algorithms produced as an output, it is far off.

Compared with scikit-learn, the best function is Relu, although to be discussed some of the parameters in scikit may give a better interpretation of the surface, such as the number of neurons, the number of layers the cost function scikit uses, etc. These figures are available as well as the code from GitHub.

4.2 Classification

4.2.1 Neural nettwork

As introduced in the method chapter, the classification is on the Wisconsin breast cancer dataset. Running it through the neural network comparing sigmoid and tanh as an output function. Same as for the regression case, the tuning of the parameters can be viewed in GitHub, and different activation functions for the hidden neurons and weight initializations will be compared.

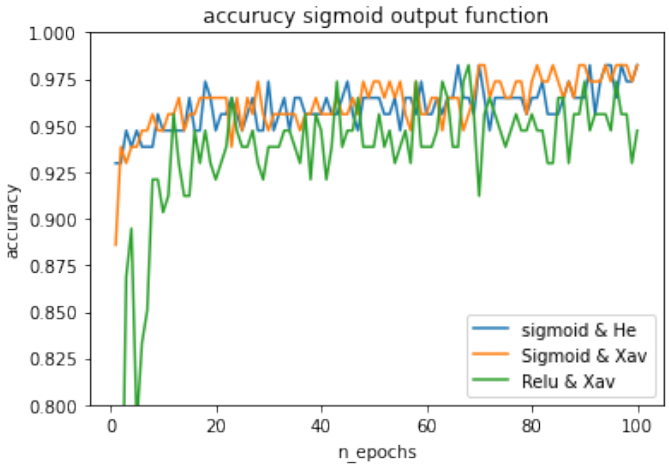

Figure 4.9: accuracy score with sigmoid in the outputlayer

The results with sigmoid in the output layer caused most activation functions to reach 98.24% accuracy after tuning. However, the need for 3-4 layers was needed for the score. Figure(4.9) shows some of the functions and weights that reached the score, plotting them all would result inn a messy plot.

However, 99% was the highest recorded with sigmoid in the output layer, although this happened at 30 epochs with the RELU with "HE" initiation.

Looking at the tanh function in the output layer, same observations were made

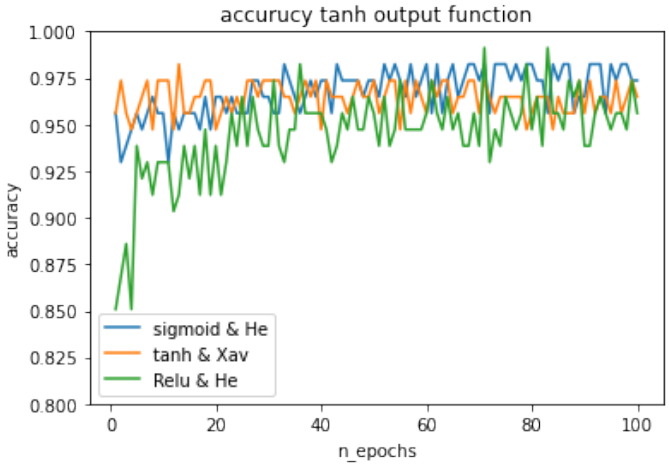

Figure 4.10: accuracy score with tanh in the outputlayer

The Leaky-Relu with He initiation managed to reach 98.24%. accuracy with tanh in the output layer with 1 layer and 20 neurons. But other good choices were also observed as seen in the plot. Overall the sigmoid activation function performed well overall and had the highest mean over 100 epochs.

4.2.2 Logistic Regression

For the logistic regression case, a separate function was created. Although the idea was to have a neural network so flexible that when setting hidden layers and neurons to zero would result in a logistic regression.

The logistic regression was tuned the same way as the neural network procedure with lambda and eta, although adding the decaying schedule for the learning rate was adopted.

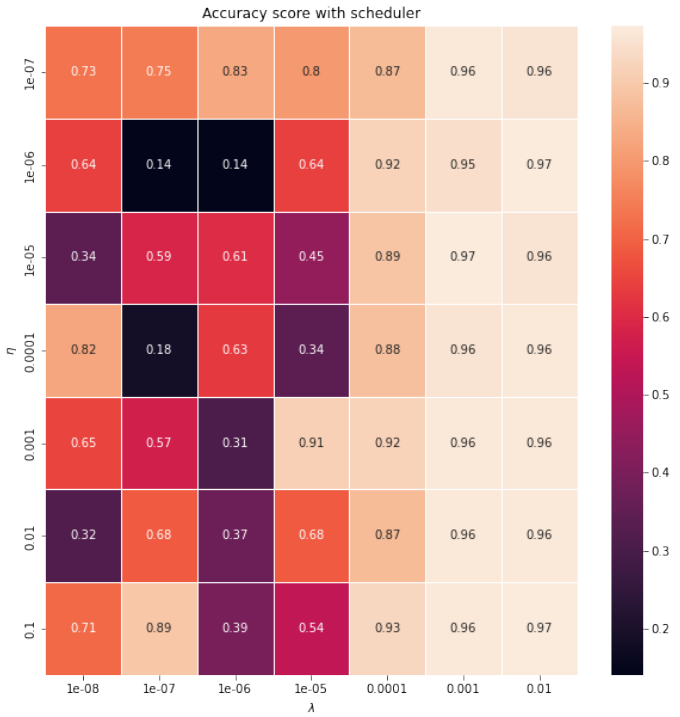

Figure 4.11: lambda eta with schedule

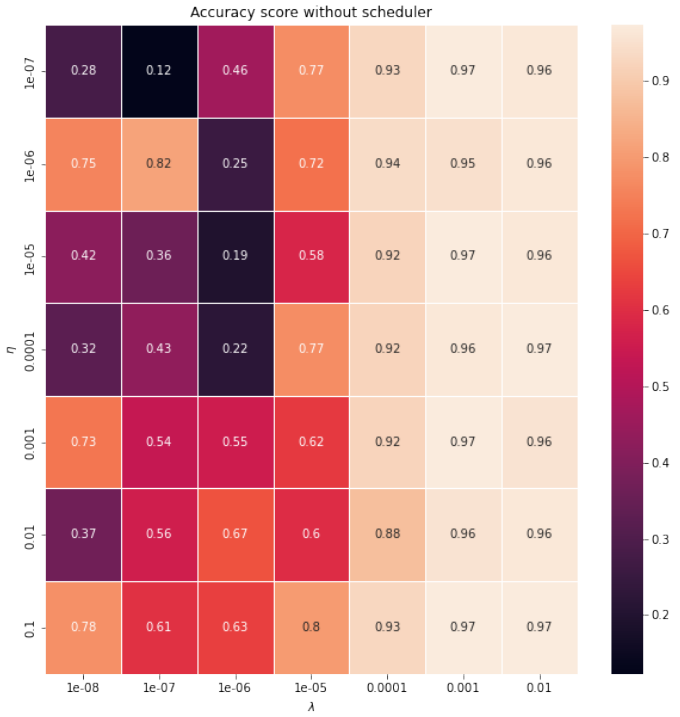

Figure 4.12: lambda eta without schedule

As observed from the test data, there is negligible accuracy with and without a

scheduler. And lambda seems to be the dominating factor. Both scheduled and no scheduled learning rates reached the best accuracy with $\lambda = 0.01\&0.001$, which was found to be 97.36%. This is close but not better than the neural network run of the classification.

Chapter 5

Discussion of the project

This section will give a broader discussion of the main topics and the comparison. As well as an explanation.

5.1 Breast cancer data

The data contains, as described, many features; some have a colinearity, such as a diameter and a radius. For a genuinely valid project, the data should have had reduced dimensionality either through PCA or other methods, but for the sake of time, this was dropped.

5.2 Stochastic gradient descent

The stochastic gradient descent was only computed with adaptive learning rates; this is because the author was intrigued by this method and by observing the number of iterations for momentum vs. non-momentum in GD lead to the notion that a plain minibatch stochastic gradient descent would be outperformed regardless. Although the momentum with scheduled learning rate was the best method after tuning, there would be interesting to observe over 200-300 epochs. Due to time, further investigation was not conducted as the author was left to his device from the middle of the project.

Describing the Franke function compared to project one, it seemed that the best scores came from the OLS. But for stochastic gradient descent, adding a regularization term helped with the sporadic behavior and led to better results. Also, when comparing figures from project one vs this project, the resulting plotting of the Franke Function was observed to be much smoother and more precise for the gradient descent methods compared to a polynomial fitting. Although minor changes, the Ridge did come out on top, these values could be minor differences and/or loss of numerical precision that contributed.

5.3 Neural Net Franke

In general, for the Neural network, the main idea was to include more optimizers for the training, such as adam and momentum. Sadly the time consumed did not

allow this to happen, and the author unwillingly decided to implement the most straightforward way with help from the lecture notes.

Even though the neural net gave quite good results for the cost scores, it was observed a dissatisfying plot of the Franke function. Why this occurred is unknown to the author, but due to time constraints, a decision was made to keep it as it was. The reason for this is that neural net managed to reproduce figures such as $y = sin(2\pi x)$ and $y = x^2$ with reasonable accuracy, and it worked as evidence for the neural net doing what it was intended to do.

In terms of cost scores, the time spent on training the network for this topological surface, and the results provided, one could argue that utilizing a gradient descent method might be better. However, the key from the neural network is that there is no need to create the design matrix of the wanted polynomial degree. Thus for a faster-automated process after updating the hyperparameter, the neural network can reliably provide good scores.

5.4 Neural net Classification

For the Classification case, the author is aware of his discussion in project one about cross-validation. And for this case, cross-validation should have been implemented to ensure a solid conclusion on the best activation function and reasonable accuracy. Again due to time constraints (2 days to hand in) when the analysis was conducted, the run time with cross-validation took too long, and it was sacrificed for the sake of giving results.

The results yield good accuracy for most activation functions with the proper tuning. Picking out one over the other based on the accuracy is not valid, but looking at the tuning, the LRELU with tanh in the output layer reached the highest score with the lowest combinations of layers and neurons. In addition, the LRELU is less computationally heavy than the sigmoid, so such a function is preferable in most cases.

Chapter 6

CONCLUSION

The project has been carried out coded and written by the author.

For describing the Franke function, the SGD with momentum and L2 regularization was superior to the gradient descent methods; the neural network, after tuning, resulted in better cost scores but could not reproduce the surface satisfactorily. Although the best performance is clear from the results that the tanh activation function

For the classification, a neural network with a sigmoid activation function was observed overall best performance for various epochs. Although other functions, such as leaky Relu with tanh in the output layer, have the lowest combination of layers and neurons. Also, as discussed, if faster convergence and lower computational time could be possible for a similar result, the trade-off should be made.

The conclusion is that a neural network, when tuned properly, can predict malignant tumors with high accuracy. Whereas for regression, it reaches an excellent cost score but fails to provide a topological figure of the surface. So, evaluating if it is a good model needs to be looked at further.

The result section on both regression and classification illustrates that activation functions, as well as weight initialization, have an impact on the goodness of the model. This is not always easy to see beforehand but needs to be tested for each case. This report concludes that the activation and weight initialization needed to be different for the optimal score in classification and regression. As well for similar scores, the activation function and weight initialization resulted in different layers and neurons. However, it has yet to be the main focus of the discussion of training the neural network.

From personal interest, the author has learned much, demystifying the neural network and seeing that even with simple and shallow nets, it can compete on simple data sets with sickit learn. The author has also learned that the need for neural net vs. other techniques should be considered for each task.

Bibliography

[1] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[2] Ibrahem Kandel and Mauro Castelli. "Transfer learning with convolutional neural networks for diabetic retinopathy image classification. A review". In: *Applied Sciences* 10.6 (2020), p. 2021.

[3] Panagiotis Antoniadis. *Activation Functions: Sigmoid vs Tanh.* URL: `https://www.baeldung.com/cs/sigmoid-vs-tanh-functions`. (accessed: 31.10.2022).

[4] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[5] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[6] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015.

[7] Vitaly Bushaev. *Adam — latest trends in deep learning optimization.* URL: `https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c`. (accessed: 08.11.2022).