Performance of feed forward neural networks in Classification and regression problems

Gaute Holen, Paul Giraud & Fridtjof Ronge Gjengset

(Dated: November 20, 2021)

In this paper, we aim to explore the performance of a non-convolutional feed forward neural network when solving regression and classification problems in comparison to other methods. We will discuss the upsides and downsides of using a neural network for the two different types of problems, and explore how the different choices of activation functions, number of hidden layers and neurons, learning rates, regularization parameters, size of mini batches and number of epochs might impact the prediction made by the network.

I. INTRODUCTION

The main focus of this paper is to explore how a feed forward neural networks(FFNN) performs in comparison to other numerical methods in different situations. Our FFNN uses stochastic gradient decent(SGD) to optimize itself. We will therefore start by comparing the results of ordinary least squares(OLS) and Ridge regression using matrix inversion, to OLS and Ridge regression using gradient decent. We will also write our own SGD code and look at it's performance for different parameters before we implement it into our neural network.

We use the Frankefunction to generate the a 3D data-set and add some stochastic noise for our regression problem. Using the Frankefunction allows us to know the exact analytical result so we know how well both methods fit to the data-set. For our classification problem we will use the Wisconsin breast cancer data offered by the **Scikit-learn** package.

Secondly we will look at how our FFNN with SGD compares to the results from a normal OLS and ridge regression, and discuss how the hyper parameters affect our results, as well as some general discussion about our FFNN. We will also use some different activation functions for our network, as well as some different ways to initialize our weights and biases, and discuss how they affect our result.

We chose to use **Keras** as a control for our FFNN. This allows us to see if our network works as intended, as well as comparing it's performance to that of a well optimized algorithm for FFNN(re-word).

Thirdly, we will move into an analysis of a classification problem. We will adapt our network to solve classification problems instead of a regression problem. We will compare the performance of our FFNN to that of logistic regression. Again we use **Keras** as a control for our neural network, and the logistic regression functionality offered by **Scikit-Learn** to control our own logistic regression.

Lastly we will discuss the different approaches in detail, and highlight some of the advantages and disadvantages to the different approaches. As the main goal of this paper is to explore the usefulness of a FFNN, we will also highlight what kind of problems we consider FFNN to be most useful for and why.

Although we aim to discuss how different parameters may impact the prediction our network makes, we will not go into a detailed analysis of how the different choices impacts the prediction of our network. We merely wish to discuss these hyper parameters in light of the data-set we wish to process, and see how we can adjust the parameters to make our network more accurate and reliable, as well as how other choices might make our network perform worse.

All of our code is available on Github[1]. You will find that the code is organized into folders. One for the neural network, one for generating the data-set and one for the SGD. All the plots produced by these programs will be saved in the 'Figures' folder.

II. METHOD

A. Stochastic gradient decent

The general idea of SGD is that the cost function can be written as a sum over all the data-points, and that we can minimize the cost function by calculating it's gradient. The stochastic aspect of gradient decent comes from the fact that we only take the gradient of a subset of the original data-set, also called a mini-batch. So that the gradient of the minibatch can be written as:

$$\sum_{i \in B_k}^{i} \nabla_\beta c_i(\mathbf{x}_i, \beta) \tag{1}$$

Where B_k is the data contained in one minibatch, ∇_β is the gradient operator with respect to β and c_i is the

[1] Link to our GitHub page: https://github.com/fridtjrg/FYS-STK4155

cost function with respect to each data-point.

We can further improve the SGD by introducing epochs. For each epoch, we calculate the gradients for a new set of minibatches to further improve our approximation.

When training the network we calculate the gradient of the OLS and Ridge regression, then improve our approximation for the beta by weighing the gradient with the learning rate. We do this for each minibatch of the data-set for each epoch, the resulting values for β is then our SGD approximation using OLS and ridge regression. We can draft this as process as a pseudo code:

Algorithm 1 Pseudo code for calculating the SGD for Ridge and OLS regression

β is an array with the same length as number of features, of random numbers. The gradient() is simply the gradient with respect to either OLS or Ridge, depending on which you wan to calculate. η is the learning rate.

```
$\beta = random.randn(features)$
for each epoch do
    for each minibatch do
        $\nabla_\beta = gradient(X, y, \beta)$
        $\beta = \beta - \eta * \nabla_\beta$
```

Algorithm II B illustrates the core calculation of the SGD. Further, it is woth noting that we used the square loss function as our cost function for the SGD, that is:

$$\frac{1}{n}||X\beta - y||^2 \tag{2}$$

where n is the number of data-points in y, y is the data-set and X is the deign-matrix.

B. Feed forward neural network

Our neural FFNN is split into two classes: Layer and NeuralNetwork. The layer class initializes the weights, biases, and handles the activation function for a particular layer.

The NeuralNetwork handles the behavior of the network such as back propagation, feeding information through the network and calculating predictions. The NeuralNetwork class also initializes new layers using the layer class. This structure allows us to easily choose how many layers we want in our network as well as what activation functions and biases.

The output of each layer in the network is calculated using the outputs of the preceding layer, the weights connecting them and the bias and activation function for the current layer. We can write easily write this out mathematically using vector notation. If y_i is the output of an arbitrary layer i, we have that:

$$\vec{y}_i = f_i(W_i\vec{y}_{i-1} + \vec{b}_i) \tag{3}$$

Where f_i is the activation function for layer i, W_i is a matrix that describes the weights that connect layer i and $i-1$, $\vec{y}_{i-1}$ is the output for the preceding layer, and $\vec{b}_i$ is a vector containing the bias for each neuron in layer i.

We will mainly use heat-maps for the MSE as a function of learning rate η and regularization parameter λ to find the optimal parameters. We create this heat-map using nested loop saving the MSE scores in a matrix. We also save the parameters that yielded the lowest MSE to create our 'optimal prediction'. This is outlined as a pseudo code in algorithm 2

Algorithm 2 Pseudo code for creating the heatmap and finding the optimal parameters for η and λ

```
for each eta[i] do
    for each lambda[j] do
        Build network and make prediction
        calculate MSE
        if MSE < 1e10 then
            MSEmatrix[i][j] = MSE
        else
            MSEmatrix[i][j] = inf
        if MSE < Best MSE so far then
            best eta = i
            best lambda = j
plot heatmap of MSEmatrix
print(indicess i and j gave lowest MSE')
```

C. Logistics regression with SGD

Logistic regression is a type of regression where an activation function called the sigmoid function

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{4}$$

which maps any real value z to lie between 0 and 1. It's used in classification, and the goal is to determine whether a given input X is classified as type 0 or type 1.

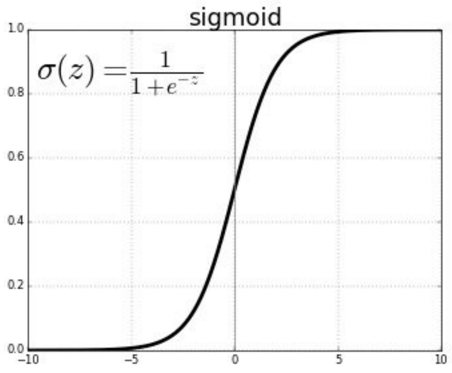


FIG. 1. The sigmoid function

The estimated probability $\hat{p}$ is given as

$$\hat{p} = \sigma\left(\boldsymbol{\beta}^T \boldsymbol{X}\right) \tag{5}$$

With simplified cost function, which aims to heavily penalise wrong estimates and barely impact correct estimates

$$C(\boldsymbol{\beta}) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1-\hat{p}) & \text{if } y = 0 \end{cases} \tag{6}$$

Using SGD, β is adjusted according to a learning rate η over a given number of epochs to minimize the loss function. The loss function for logistical regression with a l_2 regularization parameter λ is given as

$$J(\beta) = -\sum_{i=1}^{n} \left(y_i \log\left(\hat{p}_i\right) + (1-y_i)\log\left(1-\hat{p}_i\right)\right) + \lambda\|\beta\|_2^2 \tag{7}$$

III. RESULTS AND DISCUSSION

A. Data-set for linear regression

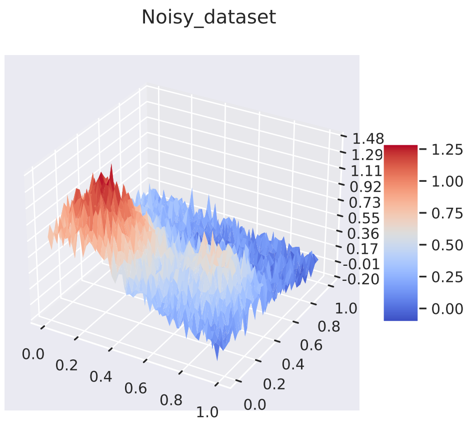


FIG. 2. This figure shows as 3D plot of the Frankefunction with some added stochastic noise.

Figure 2 shows the data-set we are going to predict throughout the regression part of this paper. The noise is distributed with the normal distribution with a mean value of $\mu = 0$ and a standard deviation of $\sigma = 0.1$. It is created with a 50x50 mesh-grid, which totals 2500 data-points.

B. Regression with matrix inversion and gradient decent

We start by looking at the the predictions when we perform OLS and ridge regression using matrix inversion

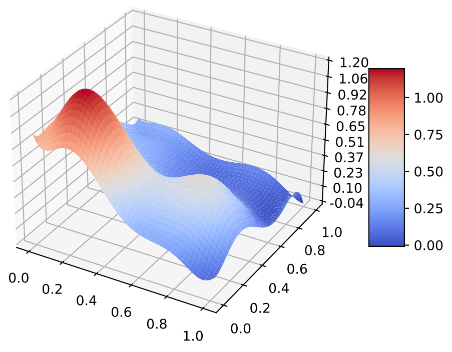


FIG. 3. This figure shows the prediction of the Frankefunction for OLS using matrix inversion.

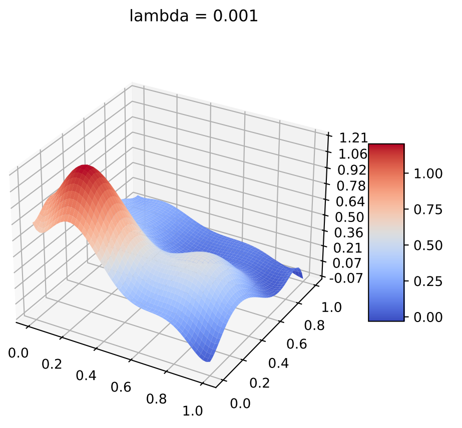


FIG. 4. This figure shows the prediction of the Frankefunction for Ridge regression using matrix inversion.

We see that both figure 3 and 4 seem to approximate the Frankefunction pretty well, and are barley distinguishable. It is worth noting that the λ value for our ridge regression is found by calculating the mean square error after a prediction for λ values between 10^{-3} and 10^{1}. The lambda that produced the lowest MSE test-score is then used for a new prediction and gives the plot you see in figure 4.

We will now look at the our predictions when we use gradient decent instead of matrix inversion.

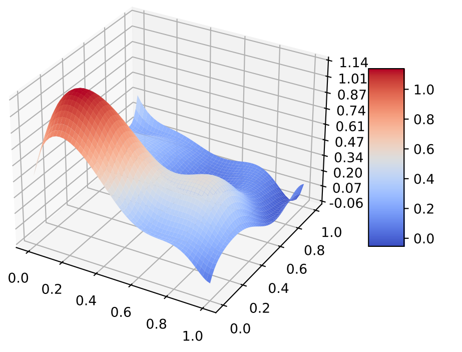


FIG. 5. This figure shows the prediction of the Frankefunction for OLS using gradient decent.

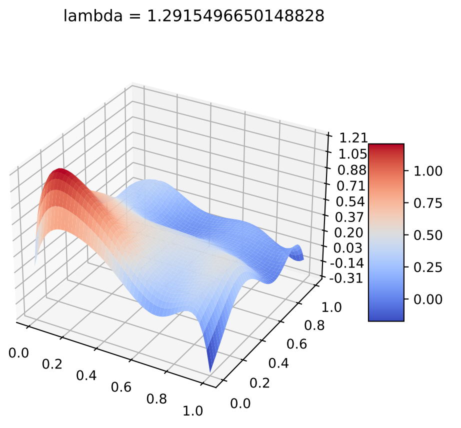


FIG. 6. This figure shows the prediction of the Frankefunction for Ridge regression using gradient decent.

The predictions using gradient decent yields results(5 and 6) that still resembles the original data-set shown in figure 2, however they do differ slightly from the predictions we got using matrix inversion. Out of the four different predictions, Ridge regression using gradient decent(figure 6) stands out the most.

Visually comparing the plot's doesn't tell us much, as they all look pretty similar, we therefore choose to look at the mean square error for each model.

		Avg. Mean Square Error	
		Training	Testing
Gradient Decent	Ridge	0.015041394	0.015065234
	OLS	0.011727902	0.013009892
Matrix inversion	Ridge	0.010339334	0.010205538
	OLS	0.010370234	0.010974138

TABLE I. This table shows the average MSE for Ridge regression and OLS, with matrix inversion and gradient decent for both the training and test-data. The average is over 5 runs.

By looking at table III B it becomes evident that the models using matrix inversion performs similarly, but the ones using GD performs slightly worse. Ridge regression preforms worst, which explains why it stands out from the other predictions visually.

C. Implementing Stochastic Gradient Decent

By introducing stochastic gradient decent, we now have more hyper parameters to work with. To get a reasonable idea of what parameters works best while not using excessive computational power, we created the following plots.

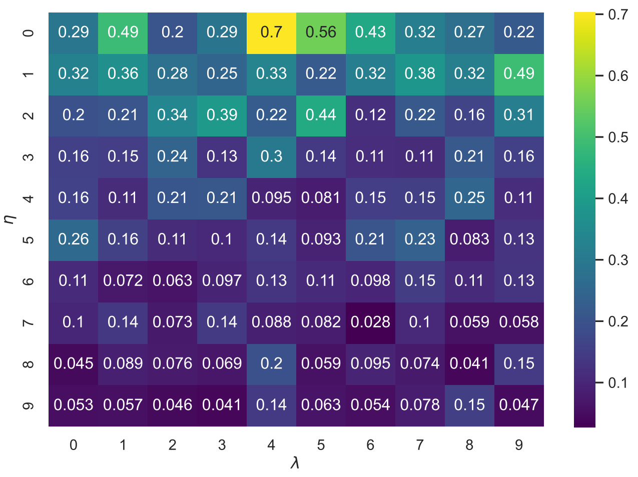


FIG. 7. This figure shows the heat-map for the test mean square error based on the learning rate η and the parameter λ, for SGD using Ridge regression.

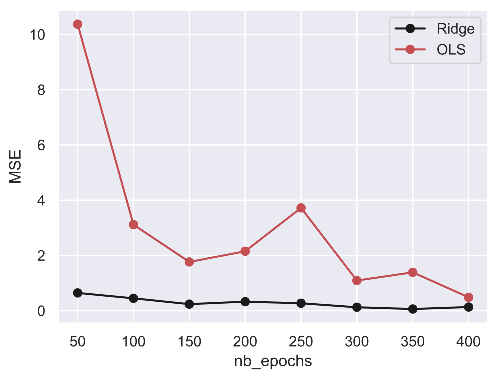


FIG. 9. This Figure shows the mean square error as a function of the number of epochs used for ridge and OLS.

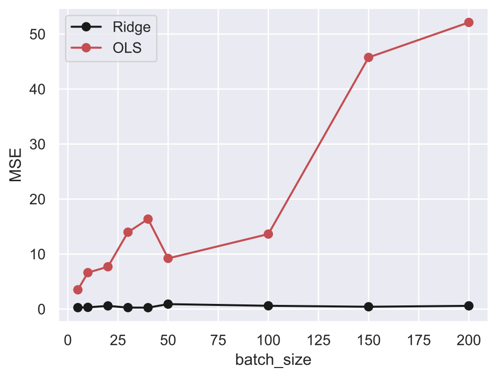


FIG. 10. This figure shows mean square error as function of the size of each minibatch for Ridge and OLS.

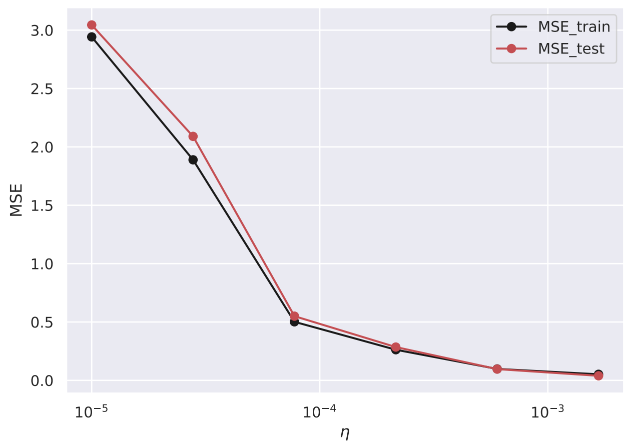


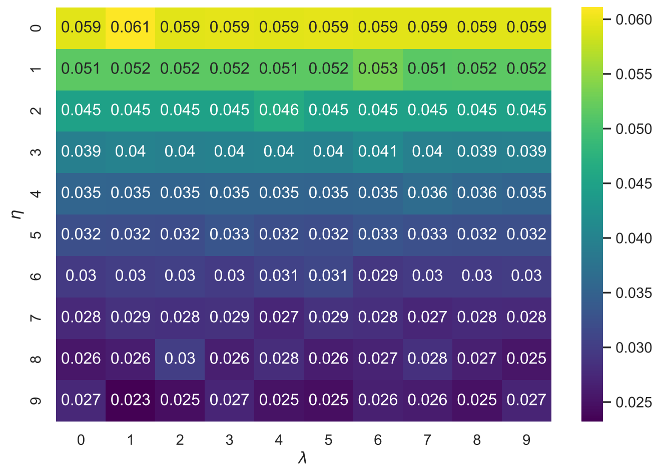
FIG. 8. This figure shows the test mean square error based on the learning rate η, for SGD using OLS.

We further plotted how the MSE reduces depending on the batch size and the number of epochs.

Figure 8 and 7 gives us an idea of what learning rate and regularization para-maters to use for the SGD depending on what regression method we use. Meanwhile figure 9 and 10 tells us how many epochs are worth using to improve the MSE, and the size of each minibatch.

From these figures we can draw the following conclusions: Looking at figures 9 and 10 it is reasonable to choose a huge number of epochs and small size for mini-batch. However, decreasing the size of the mini-batches and increasing the number of epochs increases the number of operations and therefore the computation time. The advantage of SGD is to have lower computation times for large databases. We must not lose this interest by increasing too much the number of epochs and decreasing too much the size of the minibatch because there would be no more interest in comparison with the classic GD method.

To control our results we look at the SGD offered by the **Scikit-learn** package.



	0	1	2	3	4	5	6	7	8	9
0	0.059	0.061	0.059	0.059	0.059	0.059	0.059	0.059	0.059	0.059
1	0.051	0.052	0.052	0.052	0.051	0.052	0.053	0.051	0.052	0.052
2	0.045	0.045	0.045	0.045	0.046	0.045	0.045	0.045	0.045	0.045
3	0.039	0.04	0.04	0.04	0.04	0.04	0.041	0.04	0.039	0.039
4	0.035	0.035	0.035	0.035	0.035	0.035	0.035	0.036	0.036	0.035
5	0.032	0.032	0.032	0.033	0.032	0.032	0.033	0.033	0.032	0.032
6	0.03	0.03	0.03	0.03	0.031	0.031	0.029	0.03	0.03	0.03
7	0.028	0.029	0.028	0.029	0.027	0.029	0.028	0.027	0.028	0.028
8	0.026	0.026	0.03	0.026	0.028	0.026	0.027	0.028	0.027	0.025
9	0.027	0.023	0.025	0.027	0.025	0.025	0.026	0.026	0.025	0.027

FIG. 11. This figure shows the heat-map for the test mean square error based on the learning rate η and the parameter λ, for **Scikit's** SGD using Ridge regression.

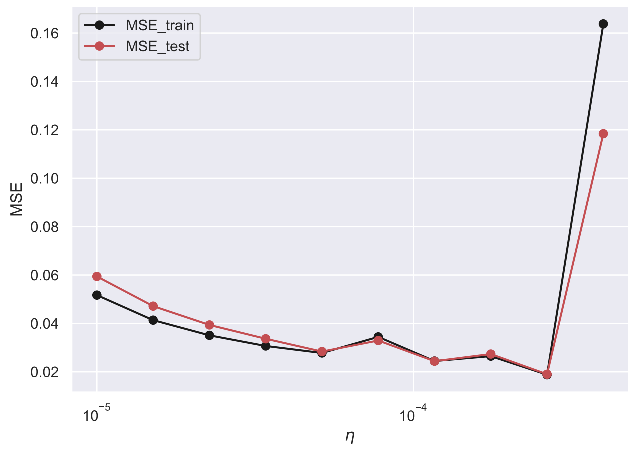


FIG. 12. This figure shows the test mean square error based on the learning rate η, for **Scikit's** SGD using OLS.

Figure III C and III C from **Scikit-learn** both give results that resemble what we see in figure 7 and 8 from our own SGD. The SGD from **Scikit** does however achieve a lower MSE score for both Ridge and OLS. However, the results from **Scikit** does verify that our model SGD works as expected.

D. Neural Network Regression[2]

For our neural network, we start by looking at the plot for the MSE as a function of the number of epochs and the MSE as a function of the size of a mini-batch.

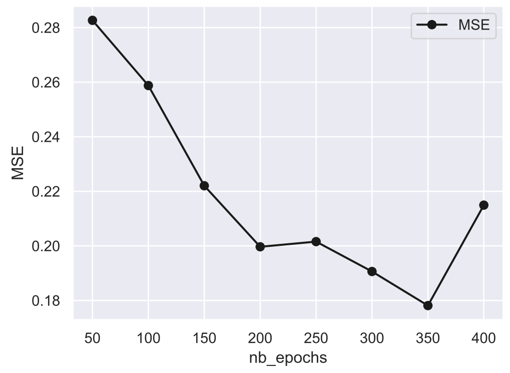


FIG. 13. This figure shows the MSE of neural network prediction as a function of the number of epochs.

Figure 13 shows that we achieve the best accuracy with $\sim$ 350 epochs. If we were to increase the number of epochs we can see that the MSE increases due to overfitting to the training data. It is worth noting that the higher number of epochs significantly increases the runtime for our program.

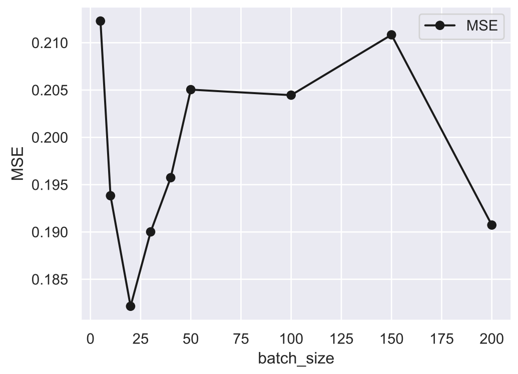


FIG. 14. This figure shows the MSE of neural network prediction as a function of the size of each mini-batch.

From figure 14 we can see that a batch size of $\sim$ 25 seems to give the lowest MSE. The graph shows that the MSE increases and decreases after this. However

[2] For the neural networks we encountered some major problems. We outline these problems in appendix VI A. We strongly encourage to read this before the results, as the problems somewhat invalidates our results, and will also be a part of our discussion.

it is not worth exploring a batch size $\sim$ 200 or higher because this will cause one batch to contain a large fraction or the entire data-set, effectively defeating the purpose of using stochastic gradient decent to improve our prediction.

We also wish to look at the heatmap for the learning rate η and the regularization constant λ. We plot these heatmap using different activation functions, so that we can compare what them and see what performed best. As described in the appendix VI A, we incountered some trouble with overflow. The grey squares you see in all the heatmaps are MSE values larger that $1e10$:

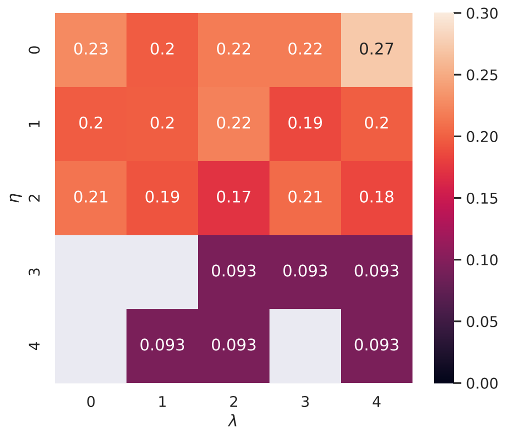


FIG. 15. This figure shows a heatmap of the MSE as a function of the learning rate η and the regularization parameter λ for our neural network using the Sigmoid activation function. Note that only the index for the learning rate and regularization parameter is shown, not their actual value, which can be found in listing 1

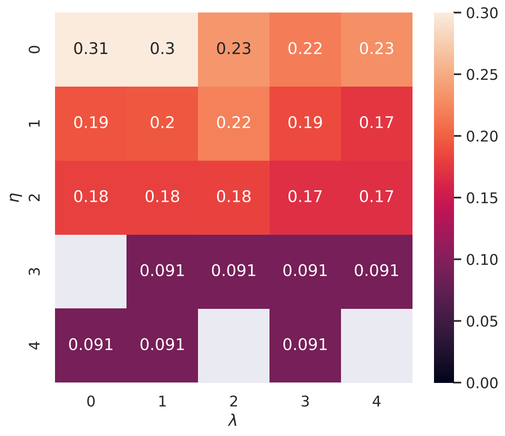


FIG. 16. This figure shows a heatmap of the MSE as a function of the learning rate η and the regularization parameter λ for our neural network using the ReLu activation function.

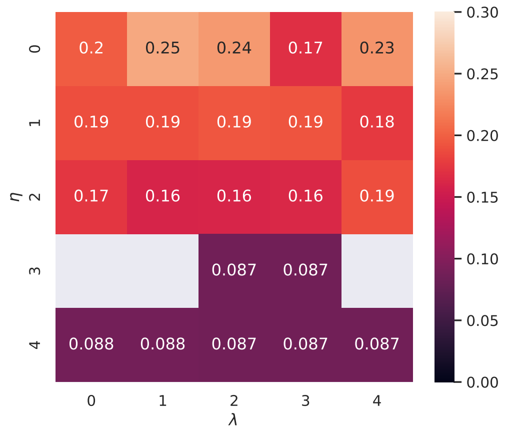


FIG. 17. This figure shows a heatmap of the MSE as a function of the learning rate η and the regularization parameter λ for our neural network using the Sigmoid-ReLu activation function.

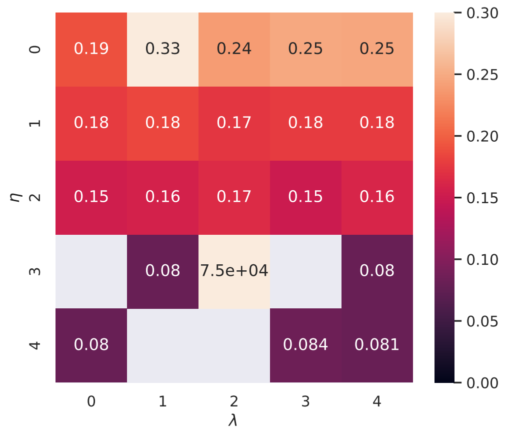


FIG. 18. This figure shows a heatmap of the MSE as a function of the learning rate η and the regularization parameter λ for our neural network using the Tanh activation function.

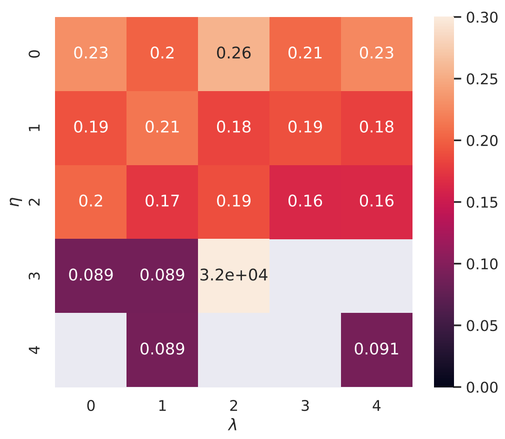


FIG. 19. This figure shows a heatmap of the MSE as a function of the learning rate η and the regularization parameter λ for our neural network using the Sigmoid-Tanh activation function.

However, we see from figures 16, 17, 18 and 19, that none of the different activation functions provides any drastic improvement over the Sigmoid activation function seen in figure 15. In light of the problems we faced outlined in appendix VI A, the Sigmoid function seemed the most stable, so we will stick to using the Sigmoid activation function.

After we have calculated the test MSE for the different learning rates and regularization parameters as shown in figure 15, our program chooses the two paramaters that gave the lowest MSE, and constructs the NN using these values. We can see this from the output of our program shown in listing 1

```
==========================================================
Our final model is built with the following
    hyperparamaters:
Lambda index =  1
Lambda =  5.623413251903491e-05
Eta index =  4
Eta =  0.01
Epochs =  100
Batch size =  5
----------------------------------------------------------
The Eta and Lambda values we tested for are as
    follows:
Lambda =  [1.00000000e-05 5.62341325e-05
    3.16227766e-04 1.77827941e-03
 1.00000000e-02]
Eta =  [1.00000000e-05 5.62341325e-05 3.16227766
    e-04 1.77827941e-03
 1.00000000e-02]
----------------------------------------------------------
Mean square error of prediction:
Train MSE =  0.17242630426430847
Test MSE =  0.17590749529133012
==========================================================
```

Listing 1. Information from terminal which displays what hyper parameters were used for our neural network, which learning rates and regularization parameters we tested for and the MSE for train and test for the network.

The output shown in listing 1 shows all the hyper parameters that were used to build the network. It also shows the MSE for the training and test data. The program also provides the prediction of the network which is shown in figure 20. Notice that we used 100 epochs and a batch size of 5 instead of 350 and 25 respectively. This is due to the fact that er were unable to plot the prediction for these values as the solution contained *nan* values(probably as a result of the overflow problems from appendix VI A).

Note that in listing 1, the test MSE score is not the same as the one we saw in figure 15 for the same indices. This is to be expected as the NN is re-constructed using the optimal parameters for λ and η, and we will therefore get a slightly different MSE.

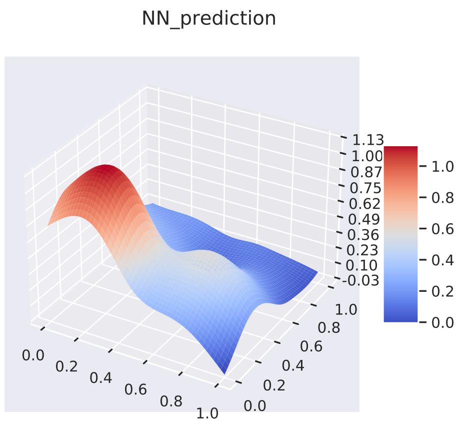


FIG. 20. This figure shows the prediction made by our neural network

The prediction shown in figure 20 seems to approximate the noisy data pretty well, but seems to miss some details which are present in the plots for both the OLS and Ridge regression shown in figure 3, 4 and 5. Looking at table III B and comparing it to the output shown in listing 1, we can see that the MSE from our neural network is also higher than that of Ridge and OLS, both with GD and matrix inversion.

As a control we compare the MSE we got for our prediction to the ones offered by **Keras**, which are shown in 21.

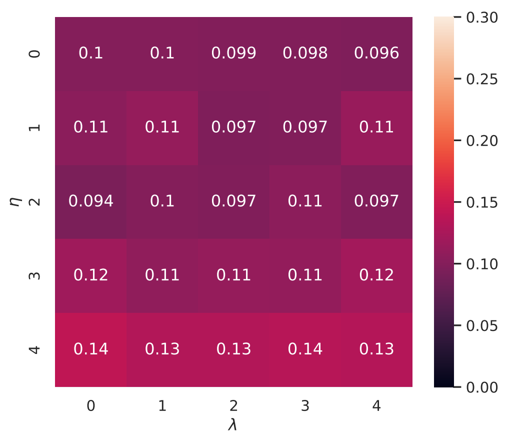


FIG. 21. This figure shows a heatmap of the MSE from **Keras** as a function of the learning rate η and the regularization parameter λ for our neural network using the Sigmoid activation function

In figure 21, we see that **Keras** gives a prediction with a somewhat smaller best MSE. However, it still does not come close to an MSE lower than that of the other methods shown in table III B.

Finding the right hyper parameters for the network is a computationally costly process, more than that of regression using OLS or Ridge with matrix inversion or gradient decent.

As the optimal hyper parameters for the network is dependent on the data-set, we have to recalculate these values every time we want to predict a different type of regression problem. We therefore need to take the entire process of finding the hyper parameters into account when considering if the use of FFNN is worthwhile for regression problems.

As finding the right hyper parameters is more computationally costly than the entire regression process for the other methods, and the NN yields a worse prediction, that is, a higher MSE than the other methods, we question the suitability for FFNN in regression problems.

However, as we faced some major problems[3] when computing these results, they are not entirely reliable. From the data we managed to produce, our MSE seems to work correctly most of the time, but is very unstable. This is reinforced by the fact that **Keras** produces an MSE of approximately same size as our NN, without encountering problems with overflow. We therefore believe that if we had managed to solve the overflow problems our NN's MSE might improve a bit, but not enough to justify it's use in favour of OLS or Ridge using matrix inversion.

[3] These are outlined in appendix VI A

E. Neural Network Classification

Moving onto a classification problem, we'll look at the Wisconsin breast cancer data from the **Scikit-learn** package. We keep using the same neural network code as we did for regression, but we opt to use the Relu activation function instead of the Sigmoid function for the hidden layers.

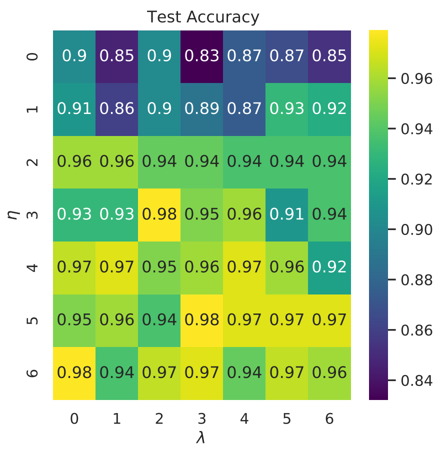


FIG. 22. This figure shows a heat-map of the accuracy score of our neural network, as a function of the learning rate η and regularization parameter λ. The accuracy score is calculated according to the test-data and prediction.

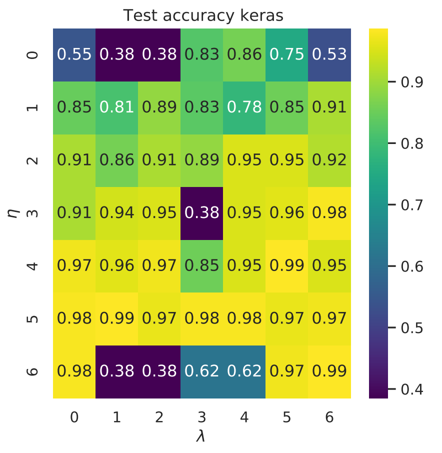


FIG. 23. This figure shows a heat-map of the accuracy score of a neural network created using **Keras**, as a function of the learning rate η and regularization parameter λ. The accuracy score is calculated according to the test-data and prediction.

Looking at figure 22, we can see that our network predicts the data-set fairly accurate, with the best accuracy topping out at 0.98. Comparing this to the prediction done by Keras seen in figure 23, we see a slightly improved accuracy score of 0.99.

With an accuracy 0.98 for our network and 0.99 for the network created using Keras, we can say they perform fairly similarly. Although the accuracy of our network is slightly lower, it would still amount to a use-full tool for classifying benign and malignant breast cancer tumors.

F. Logistic Regression with SDG

In Figure 24 and 25 the optimal parameters for batch size and eopchs is determined to be 50 and 300 respectively, which are the values that will be used for both our own implementation of SDG logistical regression and **Scikit-leanr's** SDGClassifier class. Here, again, the Wisconsin breast cancer dataset is used.

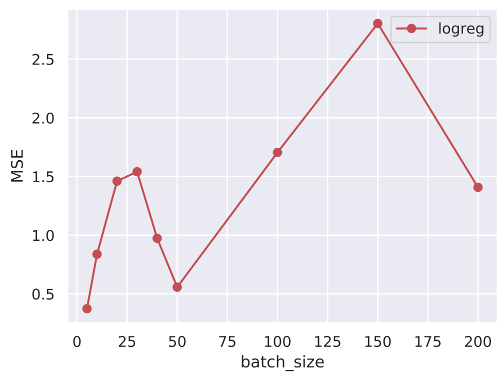


FIG. 24. This figure shows the mean loss of logistic regression prediction as a function of the size of each mini-batch

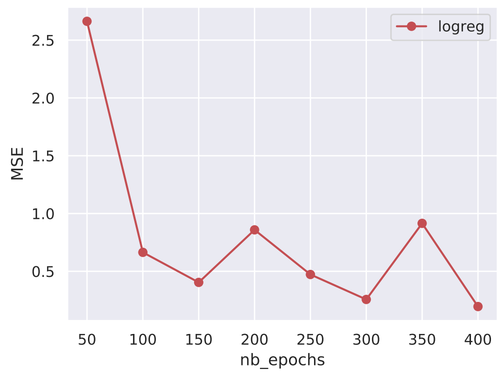


FIG. 25. This figure shows the mean loss of logistic regression prediction as a function of the number of epochs

In Figure 26 and 27 the accuracy of the model is plotted as a function of learning rate η and regularization parameter λ.

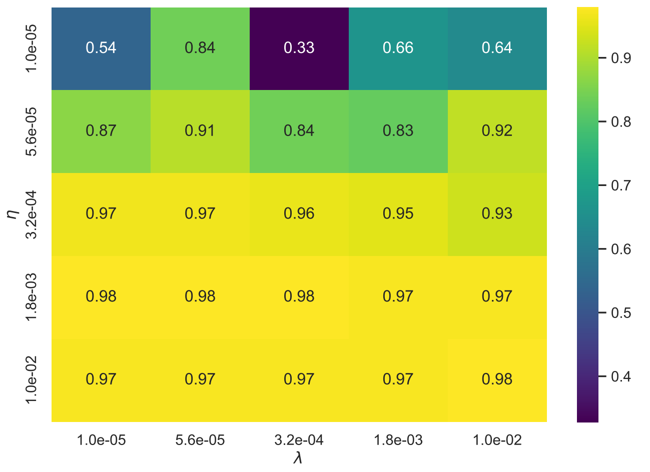


FIG. 26. This figure shows a heat-map of the accuracy score of a logistic regression using SDG as a function of the learning rate η and regularization parameter λ. The accuracy score is calculated according to the test-data and prediction.

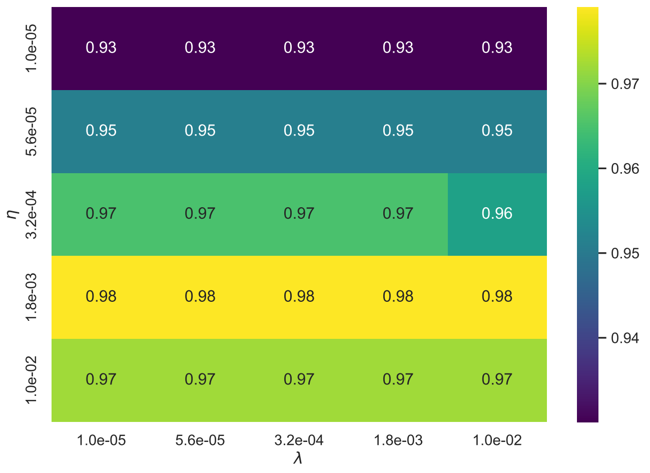


FIG. 27. This figure shows a heat-map of the accuracy score of a logistic regression using SDG with the SDGClassifier in **SKLearn**, as a function of the learning rate η and regularization parameter λ. The accuracy score is calculated according to the test-data and prediction.

From figure 26 and 27, we see that our own logistic regression performs on par with the logistic regression offered by **Scikit-learn**, which leads us to believe that our logistic regression is working as intended.

G. Analysis and evaluation

It quickly becomes apparent that the use of Neural Networks to solve a regression problem is not the optimal approach. As the appropriate learning rate η and regularization parameter λ must be found by reconstructing the network, training it with a certain amount of epochs and mini-batches, causes the process of arriving to an appropriate solution to be very slow.

Even after finding the optimal parameters, we found that the MSE of the prediction is actually worse then that of other regression methods such as OLS or Ridge, as can be seen from table III B and figure 15. As the optimal η and λ values are dependent on the problem, these can therefore not be reused either.

As both OLS and Ridge regression is much faster and produces a more accurate result, it's safe to say that a neural network solution for such a regression problem is not an efficient approach.

We didn't see any evidence that chaining the activation function, or using other hyper parameters would improve the the MSE in the regression case, as is evident from both our experimentation with the structure of the network as well as the plots for different activation functions(Figure 16, 17,18 and 19). Our belief is strengthened by the fact that the network created using **Keras** was not close to having a MSE low enough to compare to the ones we saw for other methods shown in table III B.

It is however worth noting that we were not able to run the program with the optimal number of epochs or mini batch size as we found in figure 13 and 13. However, we don't believe running for the optimal parameters would reduce the MSE enough to make the NN more accurate than it's linear regression counterpart. This is because 14 and 13 only shows the MSE being halved when we approach the optimal parameters, but we would need an MSE divided by 10, to rival the other methods.

As we encountered some problems in our NN solution for regression, our results might not be presented as a definitive evidence that a neural network solution is not worthwhile when solving this regression problem, however all our results points to this being the case.

In the classification case however, we find that our network performs very well, classifying 98% of the test-data correctly as shown in figure 22. The classification done by **Keras** show a slight improvement over our own code, classifying 99% of cases correctly as seen in figure 23.

Comparing our networks performance in Figure 22 and 23 to the logistic regression for classification in Figure 26 and 27 we see that logistic regression performs slightly better across all the hyper parameters. The SK-Learn SDG Classifier implementation for logistic regression is by far the best, with our implementation of SGD and logistics regression still performing mostly better than the neural network. In particular, what separates the two methods is that although some combinations for η and λ produce good accuracy for the neural network, the logistics regression have overall better accuracy and less variation for for different η and λ. Perhaps the neural network could be tuned better to find the absolute optimal values to rival the logistical regression, but in this case it seems like a waste of effort.

Some literature reviews [1] state that there is no definite better method for classification, and that it largely depends on the ability to adapt the model to fit the specific data and case in question.

IV. CONCLUSION

From our analysis of the regression problem we find that the use of a feed forward neural network can not rival the efficiency of OLS or ridge regressions. We say *efficiency* because it might be possible to tailor the network and increase the number of epochs to improve the MSE to rival that of Ridge and OLS, however, it will be a slow and laborious process. We argue that the process of finding the optimal hyper parameters and calculating for the appropriate number of epochs and batches, to *possibly* find a prediction better than that of OLS or Ridge is simply not worth it. At least not in this case.

There is a similar sentiment from our analysis when it comes to classification when comparing logistics regression and neural networks. Although the neural networks are much better at classification than regression, the logistic regression outperforms the neural network. Additionally, the logistic regression needs less fine tuning on the hyperparameters η and λ to provide good results. Perhaps, given enough time and effort into tailoring the neural network for this specific classification task, the neural network might perform better, but the effort and resources does not seem to be worth it compared to the relatively simple and good results from the logistic regression.

V. REFERENCES

[1] Stephan Dreiseitl, Lucila Ohno-Machado,"Logistic regression and artificial neural network classification models: a methodology review," Journal of Biomedical Informatics, 2002

[2] Micheal A. Nielsen, ”Neural Networks and Deep Learning”, Determination Press, 2015
[3] Friedman, Jerome H. The elements of statistical learning: Data mining, inference, and prediction. springer open, 2017.
[4] Mehta et al, arXiv 1803.08823, A high-bias, low-variance introduction to Machine Learning for physicists, ArXiv:1803.08823.

VI. APPENDIX

A. Major problems

We encountered two major problems very late in the process on working on the project. We tried many ways to fix both, but eventually we ran out of time to fix it, and chose to include the results as well as outline the problems we faced:

Firstly, we get an overflow message when building the neural network. This is probably due to the way we initialize the weights in the network. The weights are simply initialized according to the normal distribution with the *numpy.random.randn*() function. The sum of all the weights can then be very large, and cause overflow issues when feeding forward through the activation function(line 82 in NeuralNetworkRegression.py) . We tried to fix this using the Xavier method for initializing weights, as well as the HE method. However, we did not manage to solve the problem. As this is a problem with the core algorithm for our NN, it effects all the results we present where we use our own neural network. Even though you may encounter overflow issues when running the code, it can still produce a decent prediction and a decent MSE, which is why we are still presenting the plots. Bare in mind that if you choose to run the code yourself you might encounter issues with bad predictions for the different activation functions, as wells a *nan* values in the prediction causing the program not being able to plot at all.

Secondly, we encountered a problem when using **Keras**. When use used SGD, **Keras** would give a very weird prediction which did not look like the frankefunction. When we changed to using the Adam optimizer, we got more or less the same heatmap for MSE, but also a decent prediction. However, we wanted to use the SGD optimizer because the Adam optimizer does not use the same parameters (like λ). Even though the **Keras** prediction plot was not reasonable, the heatmap was. That is why we decided to present the heatmap even though the prediction did not work as intended.

We suspect that the first problem described might be a result of how we plot the the prediction with the optimal values. Our code calculates the optimal values for λ and η by building the network and calculating the MSE. The λ and η values that produced the lowest MSE is then used to produce the optimal prediction. As our NN code is unstable due to the overflow issues, this might cause the optimal values that were chosen to be result of which case of initializing we were 'lucky' and which where we not. This means the λ and η values actually has no correlation to the MSE, but are just a result of when we are 'lucky'. Reconstructing the NN with the optimal values might then be a case with bad weights, which will cause a bad prediction.

We also suspect that the **Keras** prediction also might be a result of how we calculate the optimal prediction, with the optimal values instead of saving the network state that gave the lowest MSE. However, this does not explain why we were able to solve the problem with the Adam optimizer using the same method for plotting the prediction.

We know that these problems somewhat invalidates our results, but as the MSE heatmaps are somewhat reasonable(as we saw when comparing to the Adam optimizer), we chose to use them so we could have a useful discussion. We will however take these problems into consideration when discussing our results.