Performance of feed forward neural networks in Classification and regression problems

Gaute Holen, Paul Giraud & Fridtjof Ronge Gjengset
(Dated: November 14, 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 use the SGD offered by **Scikit-Learn** to control check our own SGD.

We use the Frankefunction(reference?) 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.

Secondly we will look at how out FFNN with SGD compares to the results from a normal OLS and ridge regression, and discuss how the hyperparmaters 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**(referance?) 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 onto an analysis of a classification problem. By using the same code but different parameters(e.g. amount of layers, neurons, activation functions) we will adapt our network to solve classification problems instead. 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 a how other choices might 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. (something more... the results shown in this paper was generated by the code in the respective folders ish....)

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 cost function with respect to each data-point.(correct?)

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

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

When training the network we then calculate the gradient of the OLS and Ridge regression, then improve out 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.

```
β = random.randn(features)
for each epoch do
    for each minibatch do
        ∇_β = gradient(X, y, β)
        β = β − η ∗ ∇_β
```

Algorithm II A 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.

III. RESULTS AND DISCUSSION

A. Data-set for linear regression

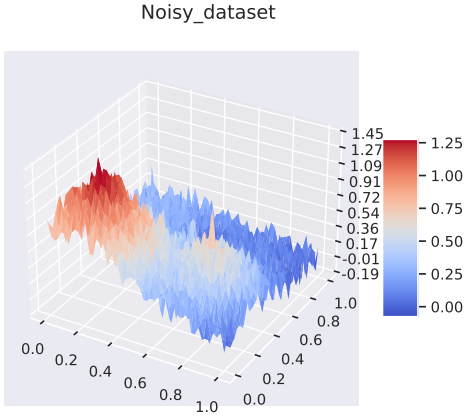

FIG. 1. This figure shows as 3D plot of the Frankefunction with some added stochastic noise.

Figure 1 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

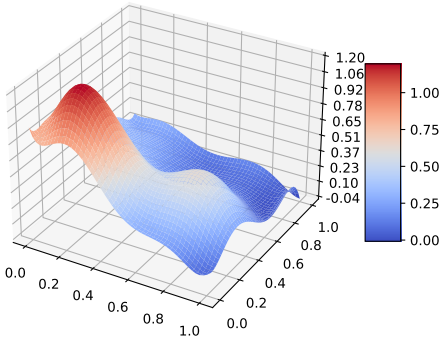

FIG. 2. This figure shows the prediction of the Frankefunction for OLS using matrix inversion.

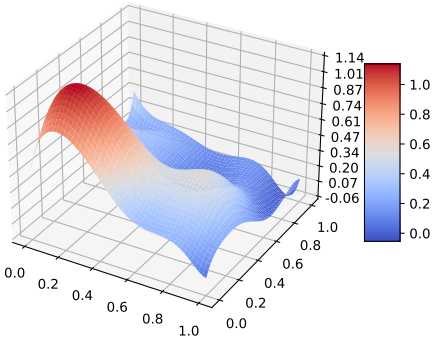

FIG. 4. This figure shows the prediction of the Frankefunction for OLS using gradient decent.

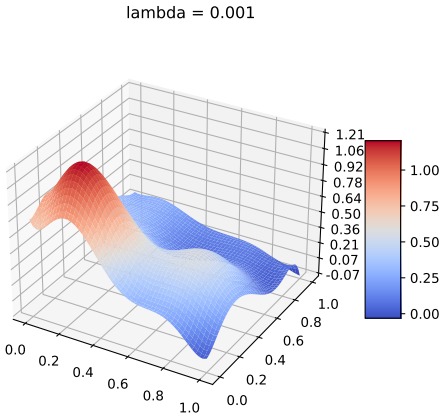

FIG. 3. This figure shows the prediction of the Frankefunction for Ridge regression using matrix inversion.

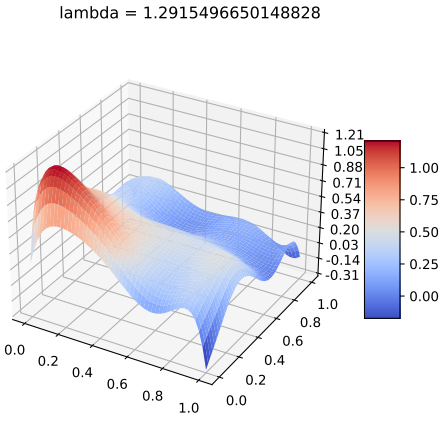

FIG. 5. This figure shows the prediction of the Frankefunction for Ridge regression using gradient decent.

We see that both figure 2 and 3 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}(Check to see if still correct). The lambda that produced the lowest MSE test-score is then used for a new prediction and gives the plot you see in figure 3.

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

The predictions using gradient decent yields results(4 and 5) that still resembles the original data-set shown in figure 1, 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 5) stands out the most. It is worth noting that the λ parameter chosen for ridge regression with matrix inversion is very different than that for gradient decent. This leads us to believe that the λ value is the cause of figure 5 standing out.

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.

		Mean Square Error	
		Training	Testing
Gradient Decent	Ridge	0.0390797	0.03930481
	OLS	0.01263608	0.01183486
Matrix inversion	Ridge	0.01070737	0.01067826
	OLS	0.01054026	0.01100813

By looking at table III B it becomes evident that all models perform pretty similarly, with the exception of gradient decent ridge regression. Note that due to the stochastic noise in the data-set, these numbers are not 'definite' but is just and example after running the program once.(maybe find average instead, is the GD ridge correct though and why is the test MSE lower?)

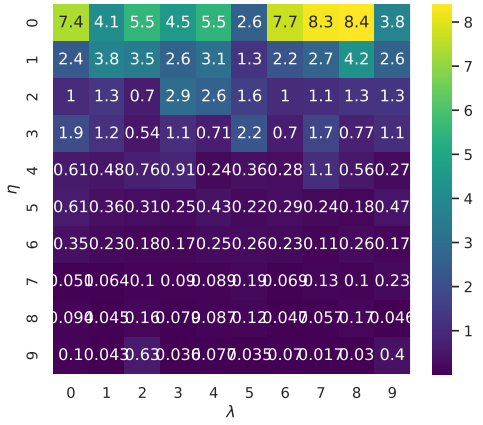

FIG. 6. This figure shows the heat-map for the test mean square error based on the learning rate η and the regularization parameter λ, for SGD using OLS.

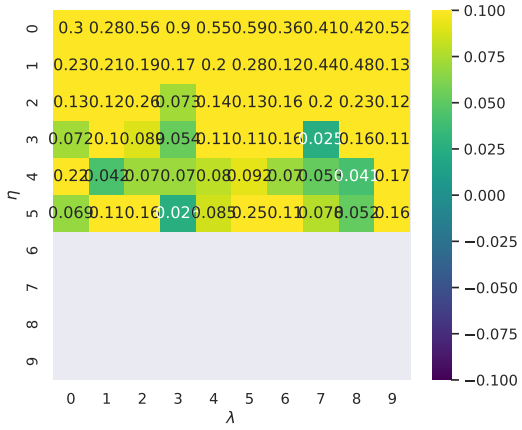

FIG. 7. This figure shows the heat-map for the test mean square error based on the learning rate η and the regularization parameter λ, for SGD using Ridge regression.

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.

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

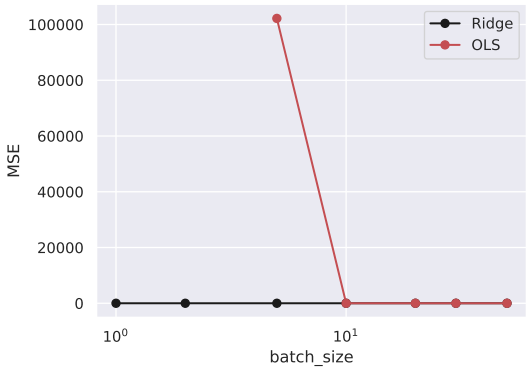

FIG. 8. This Figure shows the mean square error as a function of the number of epochs used.

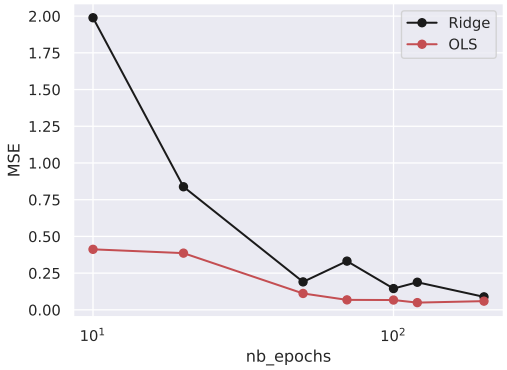

FIG. 9. This figure shows mean square error as function of the size of each minibatch.

Figure III C and 6 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 8 and 9 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 figure 8 it is reasonable to choose around 10^2 epochs as the this is where the MSE seems to converge. Adding more epochs might improve the MSE a bit, but the computational cost is not worth it.

D. Neural Network Regression

For our neural network, we again plotted a heatmap for learning rate and regularization parameter, as well as plot for the MSE as a function of the number of epochs and teh MES as a function of the size of a mini-batch.

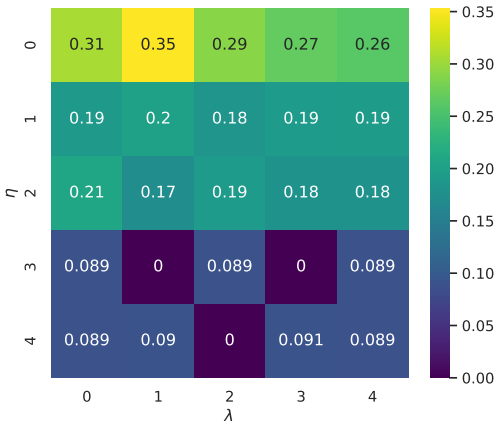

FIG. 10. This figure shows a heatmap of the MSE as a function of the learning rate η and the regularization parameter λ for our neural network. Note that only the index for the learning rate and regularization parameter is shown, not their actual value, which can be found in listing 1

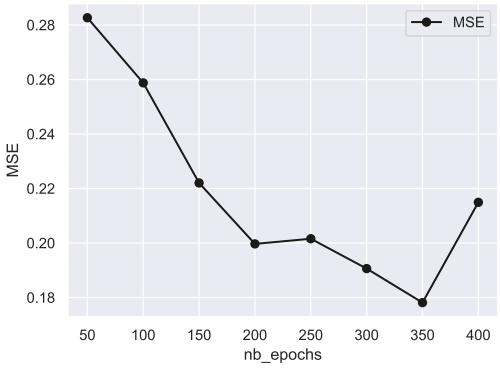

FIG. 11. This figure shows the MSE of neural network prediction as a function of the number of epochs.

Figure 12 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.

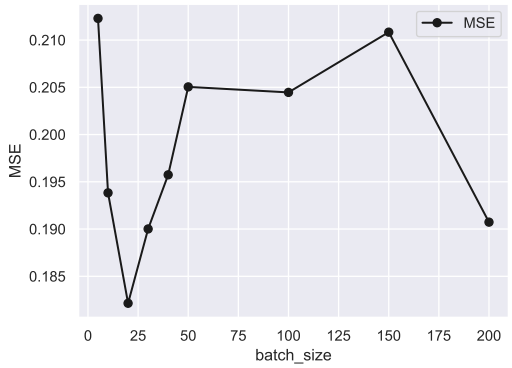

FIG. 12. This figure shows the MSE of neural network prediction as a function of the size of each mini-batch.

From figure 12 we can see that a batch size of ~ 25 seems to give the lowest MSE. The graph shows that the MSE increases again after this, but then decreases again. However it is not worth exploring a batch size at the size of ~ 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.

By looking at figure 11 and 12, we therefore chose to use 350 epochs, with a batch-size of 25.

After we have calculated the test MSE for the different learning rates and regularization parameters as shown in figure 10, our program chooses the two paramaters that gave the lowest MSE(Currently not the case), and constructs the NN using these values.

```
==================================
Our final model is built with the
following hyperparmaters:
Lambda =  0.00031622776601683794
Eta =  0.00031622776601683794
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
3.16227766e-04 1.77827941e-03
 1.00000000e-02]
----------------------------------
Mean square error of prediction:
Train MSE =  0.1735519574645502
Test MSE =  0.16726309424451224
==================================
```

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 **??** 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 13.

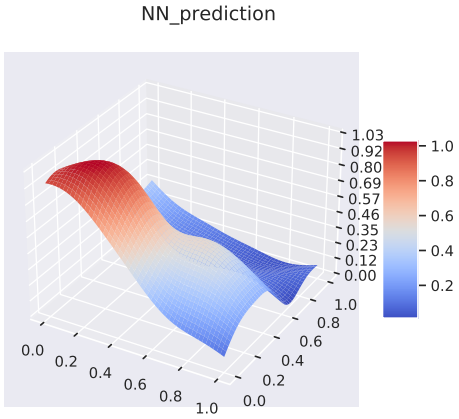

FIG. 13. This figure shows the prediction made by our neural network

The prediction shown in figure 13 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 2, 3 and 4. 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, with the exemption of Ridge with GD(for now).

As finding the right hyper parameters for the network is a computationally costly process, performing regression using OLS or Ridge with matrix inversion or gradient decent, is a less costly process, but yields a better approximation than our neural network.

E. Neural Network Classification

Moving onto a classification problem, we'll look at the Wisconsin breast cancer data from the **Scikit-learn** package(reference). 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.

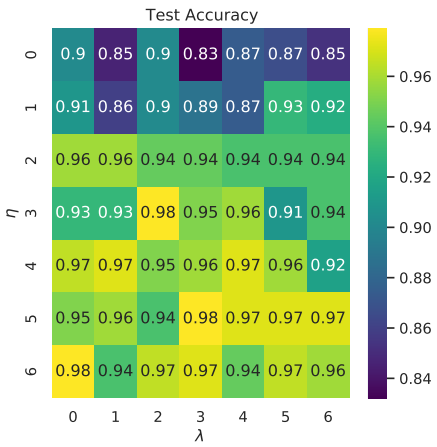

FIG. 14. This figure shows a heat-map of the accuracy score of our neural network, as a function of the learning rate η and regularization parameter λ. The accuracy score is calculated according to the test-data and prediction.

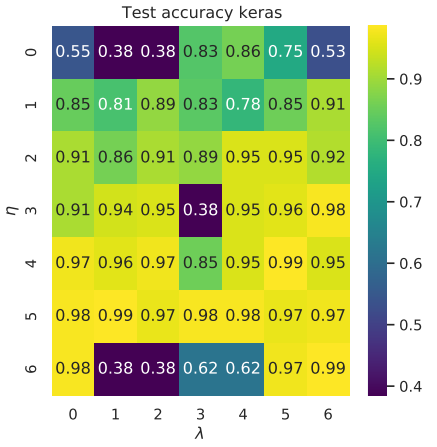

FIG. 15. This figure shows a heat-map of the accuracy score of a neural network created using **Keras**, as a function of the learning rate η and regularization parameter λ. The accuracy score is calculated according to the test-data and prediction.

Looking at figure 14, 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 15, 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 out network is slightly lower, it would still amount to a use-full tool for classifying benign and malignant breast cancer tumors.

IV. CONCLUSION