Performance Analysis of Feed Forward Neural Networks

M.S Ingstad & F.L Nilsen

(Dated: November 13, 2019)

We constructed a Feed Forward Neural Network to compare its performance to other methods. Applying it to a binary classification case in the credit card data by Yeh et. al [1], we got a maximum accuracy with k-fold cross validation using $k = 5$ of 0.808 with hidden layers of size 30 and 15 at a threshold value of 0.5. In comparison, logistic regression gave a maximum accuracy of 0.806 at a threshold value of 0.3. Plotting the ROC curves for the classifiers, we saw that the neural network performed better than logistic regression, while both models performed better than random guessing. In the regression case, we applied our model to 400 points of the Franke Function. Doing an optimal hyperparameter search to find the best possible fit, we got mean squared errors of 3.025e-4 and 3.085e-3 at amplitude relative noises of $log_{10}(\hat{\sigma}) = -2$ and $log_{10}(\hat{\sigma}) = -1$. This was significantly better than our results using the Ridge method in a previous article [2], where we got MSEs of respectively 1.884e-2 and 5.838e-2 on the same data. While performing better in both the regression and classification case, our neural network took more computational time than the other methods.

I. INTRODUCTION

When using gathered information to understand current events and predict future events, recognizing and classifying patterns is extremely important. The human brain excels at this, which has allowed us to learn from our experiences and plan our actions in a much more complex way than any other species on earth. However, as the amount of gathered information is steadily increasing in our modern world, automatizing the discovery of patterns in data is becoming more important. In the field of machine learning, various methods have been developed with this goal in mind, one such class of models with a more direct analogy to the human brain are neural networks.

Much like the neurons in the brain, a neural network consists of multiple nodes structured in a network, where each node takes some input from other nodes, and sends some output to new nodes. In a feed forward neural network, information only moves in one direction from an input layer of nodes through some hidden layers into an output layer. Although the analogy to the human brain is somewhat weakened in this case, these simpler neural networks are less computationally expensive, and can still perform well in multiple cases.

In this article, we therefore aim to evaluate its performance compared to other methods, both in the classification and regression cases, also considering how the hyperparameters of the FFNN impact its performance. First, we consider the credit card data first introduced by Yeh et. al. [1], where using various predictors the goal is to model whether a client will default or not, i.e. a binary classification case. In particular, we compare the FFNN to the method of logistic regression in this case, attempting to optimize the true positive and true negative rates of the prediction. For the regression case, we consider the Franke Function $f : \mathbb{R}^2 \to \mathbb{R}$, using polynomial coefficients of different order as our predictors. We then optimize the results in the space of hypeparameters, and compare to the results gotten in our previous article [2].

II. THEORY

Logistic regression

To perform a classification of a dataset, based on a set of predictors, one can use logistic regression. This regression method is based on the sigmoid function, given by

$$S(x) = \frac{1}{1 - e^{-x}} = \frac{e^x}{1 + e^x}. \tag{1}$$

For classification purposes, this function has the useful property that it returns values between 0 and 1. So by using the sigmoid function we can get the likelihood for an event. Here we will look at the case with two classes, and p predictors, represented as a vector $\hat{x} \in \mathbb{R}^p$. We will also introduce $p + 1$ parameters represented as $\hat{\beta} \in \mathbb{R}^{p+1}$ where we also have a bias β_0. From this we define the variable

$$t = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p. \tag{2}$$

From equation (1) and (2) we then define the probability of $y = 1$ given $\hat{x}$ and $\hat{\beta}$ as

$$p(y = 1|\hat{x}, \hat{\beta}) = S(t). \tag{3}$$

Since we only have two classes, the probability for $y = 0$ becomes $1 - p(y = 1|\hat{x}, \hat{\beta})$. Given a dataset $\mathcal{D} = \{\hat{x}_i, y_i\}$ with n data points the total likelihood for all outcomes will be given by the product of all probabilities or

$$p(\mathcal{D}|\beta) = \prod_{i=1}^{n} \left[p(y = 1|\hat{x}_i, \hat{\beta})\right]^{y_i} \left[1 - p(y = 1|\hat{x}_i, \hat{\beta})\right]^{1-y_i}, \tag{4}$$

where $p(y = 1|\hat{x_i}, \hat{\beta})$ is given by equation (3). We then define our cost function as the logarithm of this probability, multiplied with -1. By rewriting it we get a costfunction of [3]

$$C(\hat{\beta}) = -\sum_{i=1}^{n} \left(y_i t_i - \log\left(1 + e^{t_i}\right)\right). \tag{5}$$

Here t_i is the value of t one gets from equation (2) by using $\hat{x}_i$. If we now have n data points, we can define the matrix $\mathbf{X} \in \mathbb{R}^{n\times p+1}$ (where the first column is ones). We then see that we get a vector $\hat{t} \in \mathbb{R}^n$ with all values for t from equation (2). In matrix form we find this vector as $\hat{t} = \mathbf{X}\hat{\beta}$. We see that the first term in the sum in the costfunction (equation (5)) becomes $\hat{y}^T\mathbf{X}\hat{\beta}$, so the derivative with regards to $\hat{\beta}$ becomes $\mathbf{X}^T\hat{y}$ (transposed to keed the dimensions). For the second term, we see that for a single data point, the derivative becomes $\hat{x}_i \frac{e^{t_i}}{1+e^{t_i}} = \hat{x}_i S(t_i)$. In matrix form the derivative of this term is then $\mathbf{X}^T S(\hat{t})$ So the final derivative the becomes

$$\frac{\partial C(\hat{\beta})}{\partial \hat{\beta}} = -\left(\mathbf{X}^T\hat{y} - \mathbf{X}^T S(\hat{t})\right) = -\mathbf{X}^T\left(\hat{y} - S(\hat{t})\right). \tag{6}$$

Feed Forward Neural Networks

An artificial neural network is a system of so called neurons or nodes in multiple layers designed to mimic biological systems. One neuron take can take in multiple values and return one value. If we say it takes n input values x_i where $i \in [0, n]$, then the output a is calculated as

$$a = f\left(\sum_{i=1}^{n} w_i x_i + b\right) = f(z). \tag{7}$$

Here z is defined as $\sum_{i=1}^{n} w_i x_i + b_i$, w_i are adjustable weights and b is a bias term added to the expression. f is whats called the activation function, which is also adjustable. In particular, we have the parametric ReLU activation function, given by

$$f(x) = \begin{cases} x, & \text{if} \quad x > 0 \\ ax, & \text{otherwise,} \end{cases} \tag{8}$$

where a is an adjustable parameter, often set to 0.01. We will also look at the ELU function, given by

$$f(x) = \begin{cases} x, & \text{if} \quad x > 0 \\ a\left(e^x - 1\right), & \text{otherwise,} \end{cases} \tag{9}$$

and the sigmoid function given by equation (1).

In a neural network one usually have several layers, with many neurons in each layer. If we say we have L layers we can then write the expression on matrix form to make the notation easier. For a layer l we can then calculate the vector of z values $\hat{z}^l$ as

$$\hat{z}^l = \left(\mathbf{W}^l\right)^T \hat{a}^{l-1} + \hat{\beta}^l. \tag{10}$$

Here we have defined W as the matrix of all weights for all neurons in layer l this matrix will have dimension equal the number of neurons in the previous layer $l-1$ times the number in the current layer l. We have also written a and b as vectors with all values. Note that we here use $\hat{a}^0$ as the input to the neural network. For $l \in [1, L]$ we then find $\hat{a}^l$ as $f(\hat{z}^l)$. Then we can use whats known as the feed forward algorithm to make predictions with the neural network. First start with the input $\hat{a}^0$, and then use it to find $\hat{z}^1$ and from there $\hat{a}^1$, which we can use to find $\hat{z}^2$ and so on. Continue this until you get $\hat{a}^L$ and use this as your output.

It may also be desired to apply an activation function to the output, where in particular, the softmax function is given by

$$f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}, \tag{11}$$

for a set of N z values. This activation function allows in the classification case for the interpretation of the values in different output nodes as probabilities of the respective nodes, thus giving the certainty of the neural network in a specific prediction.

For the algorithm to give good results however, one needs a good set of weights and biases. To find the best values for all weights and biases, one usually use the back propagation algorithm. In the back propagation algorithm, we start to define a quantity $\hat{\delta}^l$, which has a value for all layers. To find $\hat{\delta}^L$ we need to calculate

$$\hat{\delta}^L = f'(\hat{z}^L) \circ \frac{\partial C}{\partial(\hat{a}^L)}, \tag{12}$$

where '$\circ$' denotes the Hadamard product. After we have $\hat{\delta}^L$, we find the remaining δs with the equation

$$\hat{\delta}^l = (\mathbf{W}^{l+1})^T \hat{\delta}^{l+1} \circ f'(\hat{z}^l). \tag{13}$$

The quantity δ^l_j then has the useful property that [4]

$$\frac{\partial C}{\partial w^l_{jk}} = \delta^l_j a^{l-1}_k, \tag{14}$$

and

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j. \tag{15}$$

With equations (14) and (15) we have all we need for the back propagation algorithm, since we know the derivative of all weights and biases with regards to the cost function. So when we have all $\hat{\delta}$s from equations (12) and (13), we can find the derivatives and use a gradient descent algorithm to improve our network. For the cost funcion, this depends on whether we want to study the regression or classification case. In the regression case, the mean squared error is often used, given by

$$\frac{1}{n}\sum_{i=0}^{n-1}\left(f_i-y_i\right)^2\,, \tag{16}$$

where y_i is the data points, and f_i are the corresponding points predicted by the model.

For classification, the mean squared error can be swapped with the cross entropy given by

$$R(\mathcal{D})=-\sum_{i=1}^{N}\sum_{k=1}^{K}y_{ik}\log f_k(x_i) \tag{17}$$

for a dataset $\mathcal{D}$, with n datapoints (consisting of x and y), where y have K number of classes and $f_k(x)$ gives the probability of class k given the input x.

Stochastic Gradient descent (SGD)

When doing regression or classification with different methods, one often end up with a cost function to minimize. If there is no expression for the minimum of this cost function, one have to use numerical methods to find it. One such method is stochastic gradient descent (SGD). In SGD one first divides the dataset one uses to evaluate the cost functions into several subsets, called minibatches. One then use a form of gradient descent, but the gradient is calculated over a single minibatch. So if we say we have a minibatch B_k and if a data point $\hat{x}_i$ is in B_k we say that $i\in B_k$, we get the following expression for the approximation of the gradient

$$\nabla_\beta C(\beta)=\sum_{i\in B_k}\nabla_\beta C\left(\hat{x}_i,\beta\right). \tag{18}$$

This is then used in an algorithm similar to gradient descent, where one finds the next values for $\hat{\beta}$ by the formula

$$\hat{\beta}^{(n+1)}=\hat{\beta}^{(n)}-\eta\sum_{i\in B_k}\nabla_{\hat{\beta}}C\left(\hat{x}_i,\hat{\beta}\right), \tag{19}$$

for a minibatch B_k. If one have N minibatches, updating $\hat{\beta}$ N times, using a random minibatch each time is called an epoch. In SGD one will then typically continue until a predetermined number of epoch is reached, or until the value for $\hat{\beta}$ changes by less then a set amount (over a number of epochs). We also have the parameter η, called the learning rate which is chosen before one starts the descent algorithm.

III. METHOD

Classification

We aim to compare the performance of logistic regression and a feed forward neural network when applied to a classification problem. To do this, we investigate the credit card data first used by Yeh et. al. [1]. Along with some test functions, we also apply our methods to the Wisconsin cancer data sampled by Dr. Wolberg [5], which we expect to be more accurately solvable.

Regarding the credit card data, we must first look at it to get an idea of what we should use as our predictors. By default, the data has 23 different predictors, and the output of $Y=0$ if the client does not default, and $Y=1$ if default. The predictors are all described in the article by Yeh et. a. [1], but from kaggle/discussions, we get a more complete description of some features.

With this in mind, we first plot the correlation matrix for all the predictors, and look at the result to further determine which predictors to proceed with. Afterwards, we normalize the predictors with numerical values for amounts of money by subtracting their mean, and dividing by their standard deviation. Using these predictors, we can now apply the methods of logistic regression and the Feed forward neural network to a set of training data, and evaluate the results when compared to a set of test data.

When analysing the results, we have several measures to get an idea of how good the model is. The most simple is to use the model to predict a number of outcomes, and then calculate the proportion of correct predictions. This is called the accuracy of the model. This measure however, has some problems when the dataset has a large proportion of one type of observation, as you can get a high accuracy by always predicting that class. As this is the case for the credit card data, we will also look at the confusion matrix. The elements in the confusion matrix for a binary classifier are the number of true negatives (TN), false positives (FP), false negatives (FN) and true positives (TP). To actually find the predictions we need to implement a threshold, such that if the output for the positive outcome is larger than the threshold we classify the data point as a positive outcome or 1 (default for the credit card data), and if not we classify it as a negative outcome or 0.

When actually calculating these measures, we will use k-fold cross validation. Here the dataset is split into K subsets, and then one trains the model on $K-1$ subsets, while the last subset is used as the test data for the model. One then does this K times, with each set acting as the test data once. As the final estimate for the accuracy measure of the model one use the mean of all results. When using k-fold validation to find the values for TN, FP, FN and TP we will not compute the mean of each repetition, as each subset may not be of the same size. Instead we will divide the values by the number of data points in the test set so that we get normalized values. To evaluate how good a method works, we will then plot the ROC curve for the classification. To find this curve, we split our data into 80% training data to train the classifier and 20% test data which we use to find the ROC curve. To find the ROC curve we vary the threshold for classification, and calcualte false positive rate (FP/(FP+TP)) and true positive rate (TP/(TP+FN)) for each value for the threshold. To find the ROC curve we used the `metrics.roc_curve`-function from scikit learn [6]. We then took the average curve over 10 random test-train splits, and used this as the final curve. Since `metrics.roc_curve` returns a varying number of points dependent on the dataset, we then had to use interpolation on the true positive rate to find the values at some common values for the false positive rate.

For classification, the models we will evaluate are logistic regression and a neural network. For the activation function for neural network, we tested the ReLU function given by equation (8), the ELU given by equation (9) and the sigmoid given by equation (1). This was apart from the last layer, where we use the softmax function (equation (11)). As the cost function we use the cross entropy, given by equation (17).

To train the model, we will use stochastic gradient descent, with a minibatch size of 128 in the classification case. For logistic regression we will use equation (6) for the gradient, and for the neural network we will follow the backpropagation algorithm, but only use 128 datapoints at a time to find $\hat{\delta}$.

Regression

For the regression case, we consider the real multivariate franke function $f : \mathbb{R}^2 \to \mathbb{R}$, which we explored in a previous article [2] using methods of linear regression. We will thus compare the results we get here to the results gotten before, and evaluate whether the neural network performs better.

Before applying the neural network to the data, we note that the output is no longer a binary case, but instead singular, as the Franke Function only yields a single value for y. We can therefore no longer use softmax (11) for the final activation function. As this is now regression, we want output in terms of numerical values, and thus use ReLU as before (8), but with the parameter a set to 1, effectively yielding the numerical value found in the output neuron. We must also change the cost function of the network, as the Cross-Entropy method is no longer suited. In this case, the mean squared error (16) which we used in [2] seems like the logical choice, having a direct interpretation as a deviation from correct values, and also being exactly differentiable. As for the input predictors that determine the shape of the input layer of the network, these are given by polynomial coefficients of the independent variables x_1 and x_2 in the Franke Function, similar to how we did it in [2].

With the neural network set up, we mainly evaluate the performance on 400 data points of the Franke-function, with varying random noise added to further study potential effects of overfitting and directly compare to the performance of the linear regression methods in [2].

In order to best evaluate the performance of the neural network, we must first find a minimum amount of epochs needed for the network to converge near its optimal fit.

Using this amount of epochs, we also need to determine multiple hyperparameters which impact the performance of the Neural Network, and any possible dependence on the amount of noise we add to the data. This noise is gaussian, where $\hat{\sigma}$ is the standard deviation of the noise relative to the range of the data in terms of the dependent variable y. For the hyperparameters, we aim to find optimal values for the learning rate η, the size of the batches used in the SGD method, the amount of hidden layers as well as their node counts, the value of the parameter a in ReLU for the hidden layers, and finally the order of the input polynomial coefficients determining the complexity in the input layer.

To compare all these adjustable parameters, one method could be to do a grid-based search on different values for all possible combinations. As this would be computationally expensive, and also not easily plottable in less than 5 dimensions, we instead choose to only look for the optimal values. To do this, we first initialize a value for the 5 different parameters from the desired search space. Then we in turn find the optimal value for 1 parameter, fixing the remaining 4, and resetting that parameter to its optimal value, before moving on to the next. This process is then repeated a given amount of times, and the parameter combination that gave the best result is stored. The time saved by this process can be written as

$$\frac{N \sum_i s_i}{\prod_i s_i}, \tag{20}$$

where the s_i are the amount of values used in the different search parameters and N is the amount of loops used in our proposed parameter search.

To evaluate the errors for comparison during the parameter search, we use the R^2 score and apply the k-fold cross validation method with k=2,3,4,5. This means that for k=2,3,4,5 we split the data into k parts, and choose one at a time for training, and the rest for test data. We then use 500 epochs of training for the FFNN, and evaluate the R^2 score with respect to the test data. We also calculate the MSE with the k-fold cross validation for the found optimal parameters in order to compare with our results from [2].

Having found optimal parameters, we further investigate the hyperparameters one at a time, by plotting the bias, variance and MSE of the model as a functdion of values of the given parameter. In this case, we use bootstrap with k=20 resamples to get a better estimate for the errors.

IV. RESULTS

.

Classification

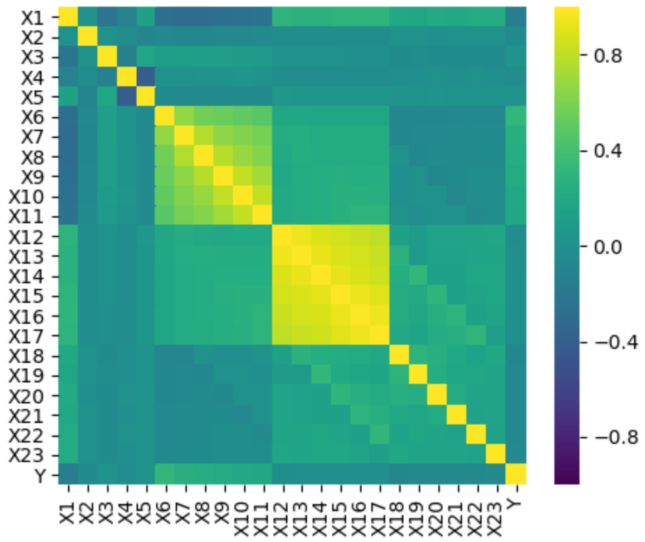

Figure 1. Contains all the predictors in the credit card data set as they are by default.

For the credit card data, we see from Figure 1 that there are some features which appear to be strongly correlated. In particular there is a strong correlation between the features X6-X11 and X12-X17 which describe respectively history of past payment and the amount of bill statement in previous months. We also note that in particular, X5 and X6 appear to have a negative correlation, which is consistent with the two being respectively marital status and age.

Although the correlation with the default status Y is not as strong as some of the others in this correlation matrix, we can still see from figure 1. that X6-X11 in particular seem to correlate strongly, along with X1 which has a strong negative correlation. After looking at the numerical values we found that X6-X11 correlated most with Y, followed by X1, then X17-X23, then X2 and X3.

These were thus the features we chose to base our model on, but first we also modified the predictors somewhat into new features. In particular, we chose a weighted average for X6-X11 along with X17-X23, and also used values for X1, X2, and X3, giving a total of 5 predictors. For the second predictor X2 corresponding to gender, we modified it from being 1 = male and 2 = female, to -1 = male and 1 = female.

With these new predictors, we get the correlation plot shown in Figure 2.

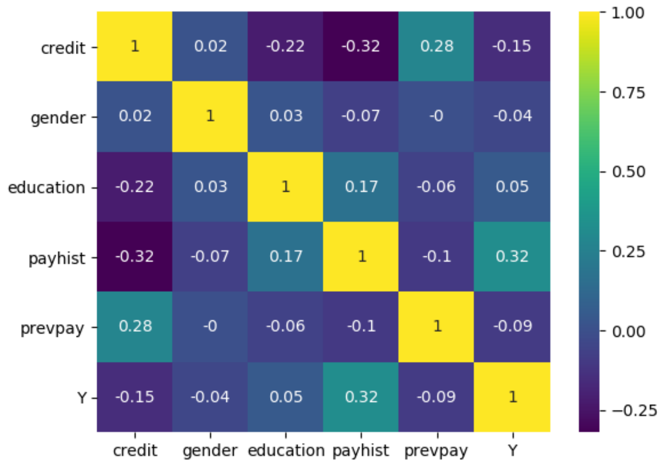

Figure 2. Correlation plot showing the 5 modified predictors of the data set which we use.

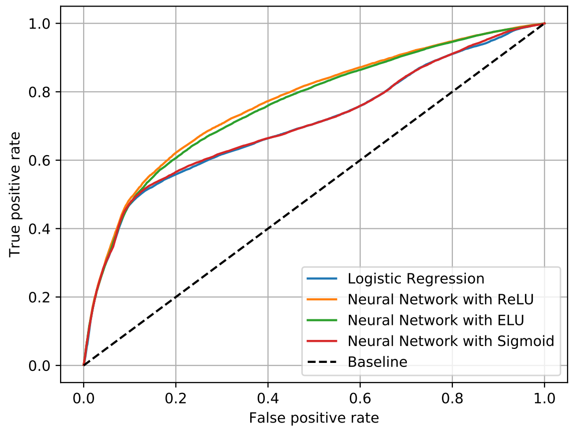

Figure 3. ROC curve for the credit data, with neural networks using the ReLU, ELU and sigmoid activation function, and logistic regression. Logistic regression is in blue, below the neural network using the sigmoid function.

When using k-fold cross validation with $K = 5$ on the entire dataset with a thresholds 0.1, 0.2, ..., 0.9 for logistic regression, we got the results given in Table I. For the neural network we got the results in Table II. The neural network that was used for these results had two hidden layers, of size 30 and 15, both with the ReLU activation function with $a = 0.01$. Each method was trained using 1000 epochs, with a learning rate of 0.1 and a batch size of 128. The ROC curves are plotted in Figure 3, with the same specifications for the methods. There we have also included the sigmoid and ELU activation functions for the neural network. These curves are the average of 10 random test train splits. We also tested the methods on the cancer data, and got the ROC curves in Figure 4. In both these figures we have added a baseline (dotted black line) which represents the theoretical performance of a random classifier, where the rate of positive events correctly classified as positive, and the negative events wrongly classified as positive are the same.

Threshold	**TN**	**FP**	**FN**	**TP**	**Accuracy**
0.10	856.4	3719.6	107.6	1212.0	0.351
0.20	2466.6	2109.4	399.0	920.6	0.575
0.30	4141.4	434.6	708.0	611.6	0.806
0.40	4299.0	277.0	872.8	446.8	0.805
0.50	4434.0	142.0	1019.2	300.4	0.803
0.60	4536.4	39.6	1218.4	101.2	0.787
0.70	4558.4	17.6	1279.4	40.2	0.780
0.80	4567.8	8.2	1303.8	15.8	0.777
0.90	4572.8	3.2	1314.6	5.0	0.776

Table I. Results for logistic regression with 1000 epochs, trained with a batch size of 128.

Threshold	**TN**	**FP**	**FN**	**TP**	**Accuracy**
0.10	1539.6	3036.4	146.4	1173.2	0.460
0.20	3411.0	1165.0	441.8	877.8	0.727
0.30	3989.0	587.0	625.4	694.2	0.794
0.40	4152.6	423.4	712.6	607.0	0.807
0.50	4236.0	340.0	793.0	526.6	0.808
0.60	4440.6	135.4	1031.8	287.8	0.802
0.70	4527.4	48.6	1188.8	130.8	0.790
0.80	4566.4	9.6	1295.4	24.2	0.779
0.90	4575.2	0.8	1317.6	2.0	0.776

Table II. Results for the neural network with two hidden layers of size 30 and 15, with 1000 epochs trained with a batch size of 128.

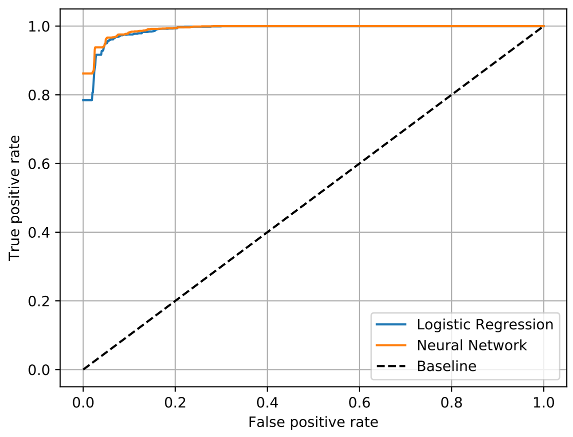

Figure 4. ROC curve for the cancer data

Table IV. Optimal fits of 400 points of the Franke function using Ridge regression from [2]

$\log_{10}\hat{\sigma}$	# data points	MSE	p_{opt}	$\log_{10}\lambda_{\mathrm{opt}}$
-3.0	400	1.954e-02	10	-10.0
-2.0	400	1.884e-02	7	-11.0
-1.0	400	5.838e-02	6	-6.0

Regression

Table III. Search space for optimal parameter search. 400 data points of data used

Order	η	**hlayers**	**a**	**Batches**
1	0.05	[]	0.005	1
2	0.1	[30]	0.01	5
3	0.15	[60]	0.015	10
4	0.2	[30,15]	0.02	15
5	0.3	[60,30,15]	0.03	20
6	0.4	[32,16,8]	-	-
7	-	[16,8,4]	-	-

For the regression case, we first plotted the MSE as a function of epochs of training in figure 5 for different values of the amplitude relative noise $\hat{\sigma}$. The plot shows that at lower noise, convergence is not reached after 10000 epochs, while at higher values of noise, convergence is reached earlier, and the MSE starts increasing at higher epochs. We chose 500 epochs and used the search space shown in Table III to find the optimal parameters for the FFNN for 400 points of the Franke Function. First, we considered 2d polynomials containing terms of the dependent variables x_1 and x_2 from 1^{st} to 7^{th} order. For the learning rate η, we chose values between 0.05 and 0.4 based on some initial tests. For the hidden layers, we experimented with different amount of hidden layers, as well as different amount of nodes in each layer. For the paramer a in ReLU we simply chose values around the often used value of $a = 0.1$. For the amount of batches, we chose values between 1 and 20. Since the training data is 80% of the 400 data points, this thus ranges in batch sizes between 360 and 18.

Since we used the k-fold cross-validation method using splits of k = 2,3,4,5, we had to fit 14 times for each value in each set of parameters, with the whole process also being repeated for 11 different values of the relative noise $\hat{\sigma}$. This caused the optimal parameter search to take multiple hours. As we used $N = 3$ loops over the search, we can calculate the fraction of time it took with respect to the grid-based search, which becomes

$$\frac{3 \cdot (7+6+7+5+5)}{7 \cdot 6 \cdot 7 \cdot 5 \cdot 5} \approx 1/82.$$

Thus, a grid based search in the same search space would

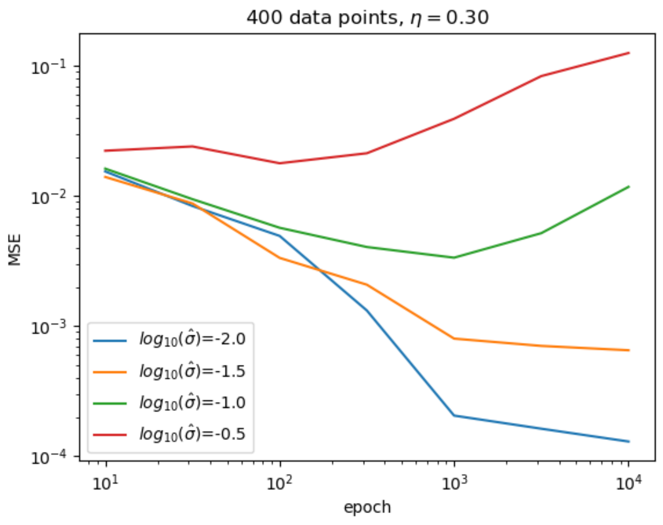

Figure 5. Mean squared error as a function of epochs of training for different values of noise. We modeled on 400 data points, with $\eta = 0.3$, a 4th order polynomial and hidden layers [60,30,15]. 15 batches of SGD was used and the parameter $a = 0.01$. k-fold cross-validation with k=2,3,4,5 was used to find the values of the MSE.

have taken approximately 82 times as long, extending the search time to weeks.

The results from the optimal parameter search is shown in table V. From this, we firstly see that the amount of optimal hidden layers remains the same for all the different values of noise, which is the highest complexity of hidden layers in our search space. The polynomial order, . the learning rate η, and the amount of bathces used in the SGD all tend to decrease as the noise increases. The parameter a belonging to the ReLU activation function (8) fluctuates with no such pattern. As for the error shown by the MSE, we see that this does not change much for the lowest values of the noise, but eventually starts decreasing as the noise increases. Comparing to table IV, we see that $\hat{\sigma} = 10^{-3}$, $\hat{\sigma} = 10^{-2}$ and $\hat{\sigma} = 10^{-1}$ are all lower in the fit made by the FFNN than with the Ridge method.

Further analysis of the hyperparameters with respect to bias, variance and MSE can be seen in figures 9, 6, 8 and 7, where we analyzed on 100 data points of the Franke function with $log_{10}(\hat{\sigma}) = -1$. For the parameters that do not vary in each of these figures, we kept the hidden layers at [60,30,15], the number of batches at 15, the value of η at 0.2, the epochs of training at 500, and the parameter a in ReLU at 0.005. This was chosen from table V. The values for MSE, bias and variance are calculated after 15 resamples of the bootstrap method with an 80-20 train-test split. Note that the reduced data points increases the found MSE compared to the results from table V.

From figure 6, we see that both the bias and variance

Table V. Results from the optimal parameter search, using the search space in Table III for different values of the noise $\hat{\sigma}$ with 400 data points. Finally the mean squared error evaluated on 500 epochs of training using the k-fold error with k=2,3,4,5 is listed.

$\log_{10}\hat{\sigma}$	**Order**	η	**hlayers**	**a**	**Batches**	MSE
-3.0	6	0.4	[60,30,15]	0.01	20	1.846e-4
-2.75	7	0.4	[60,30,15]	0.02	20	2.045e-4
-2.5	7	0.4	[60,30,15]	0.03	20	2.588e-4
-2.25	7	0.4	[60,30,15]	0.03	20	2.130e-4
-2.0	6	0.3	[60,30,15]	0.015	20	3.025e-4
-1.75	4	0.3	[60,30,15]	0.03	15	5.366e-4
-1.5	4	0.3	[60,30,15]	0.005	15	7.991e-4
-1.25	5	0.2	[60,30,15]	0.03	15	1.992e-3
-1.0	3	0.2	[60,30,15]	0.005	20	3.085e-3
-0.75	2	0.3	[60,30,15]	0.02	5	6.496e-3
-0.5	1	0.1	[60,30,15]	0.015	5	1.516e-2

mostly decrease until abourt 4 hidden layers before increasing again. The variance gains more importance for the MSE from 1 hidden layer and more. We also note that the value for the MSE has a minimum at about about 3-4 hidden layers, which is consistent with the optimal hidden layer complexity found in table V.

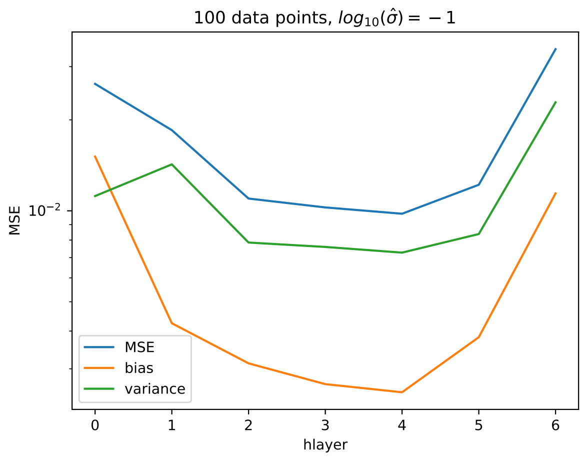

Figure 6. Bias, variance and mean squared error plotted as a function of the amount of hidden layers in the FFNN. The MSE, variance and bias are respectively plotted in blue, green and orange. The nodes in the layers were respectively 64, 32, 16, 8, 4,2.

In figure 7, we now plot the bias, variance and MSE as a function of the epochs of training for the FFNN. For the relative difference between the bias and variance, we see that the bias dominates for a lower amount of epochs, while for a higher amount of epochs the variance seems to dominate. This results in the MSE having minimum values around 300-700 epochs.

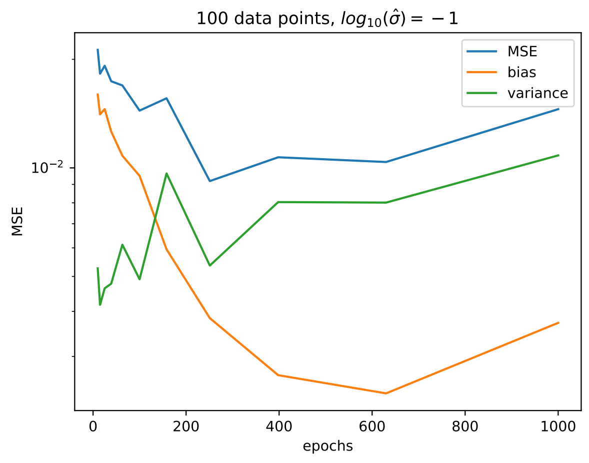

Figure 7. Bias, variance and mean squared error plotted as a function of the epochs used to fit the data. The MSE, variance and bias are respectively plotted in blue, green and orange.

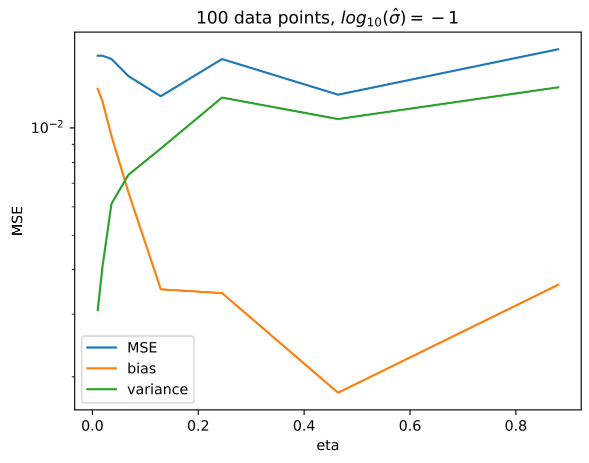

Figure 8. Bias, variance and mean squared error plotted as a function of the learning rate η. The MSE, variance and bias are respectively plotted in blue, green and orange.

In figure 8, we plotted the MSE, variance and bias as a function of the learning rate η. Here we see that the bias starts out as the dominant factor, but quickly becomes less important for increased values of η. The MSE, however, seems to remain mostly constant for all the values.

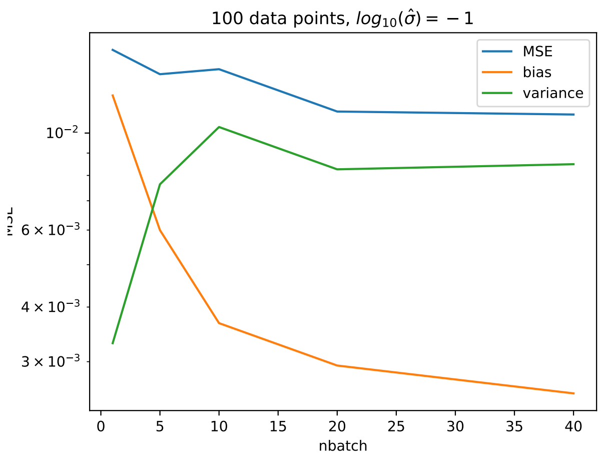

Figure 9. Bias, variance and mean squared error plotted as a function of the amount of batches. The MSE, variance and bias are respectively plotted in blue, green and orange.

In figure 9, we compared the MSE, bias and variance to the amount of batches used in SGD. We also used 1 batch, which corresponds to the standard gradient descent method. We see that the bias dominates at first, but the variance starts dominating at around 5 batches, which corresponds to 10 data points per batch. The MSE also seems to reach its lowest point at 20 batches.

V. DISCUSSION

Classification

When modifiying the predictors of the credit card data, we first considered the features which correlated most strongly with the default value Y. However, it is also important to note that including multiple features that correlate strongly to each other may not give as much information as a feature which does not correlate with already selected ones. So if we want to choose 5 predictors, we didn't choose all of X6-X11 simply because these correlated most with Y, because they also correlated strongly with each other. Exactly because of their correlation, we chose to condense these predicotrs into one through a weighted average, which we chose somewhat arbitrarily. A possible improvement tho the fit we made could thus be to use this as a hyperparameter for the case, and perhaps get better predictions. We could have also only chosen X6, as X6-X11 describes past payment records for previous months, and X6 is the most recent one. In the figure, this can also be seen by this predictor having the highest correlation with Y. As for the gender predictor X2, we chose to shift this from 1 and 2, to -1 and 1, as we thought that this would better separate the two cases and perhaps increase the correlation of this predictor. This is also why we chose to normalize the numerical values found in X1 and X18-X23 about their mean.

From the plot of all the predictors in figure 2, we see that they seem to correlate relatively well with the dependent variable Y. However, it still appears that the values for gender and education in particular may not matter as much, and could be excluded for a more simple analysis. Regarding the loss of information from including predictors which correlate strongly inbetween, we can also see that the amount of credit seems to correlate somewhat with all our other predictors, except for the gender, so in that sense, the gender remains a separate, and thus possibly useful predictor. As for the correlated features, a possible improvement to our selected data could for instance be to perform a principal component analysis on the 5 predictors to reduce it to a lower amount, perhaps not losing too much information because of the correlation with the credit predictor. Still, a pca will not take into account the correlation with the data we are trying to model, so the loss of information may still be noticeable, and 5 predictors is not a particularly high number with regards to numerical complexity of the situation.

From Figure 3 and Tables I and II in the classification case, we see that the Neural network generally seems to perform a bit better than logistic regression on the credit card dataset. With a threshold of 0.5, we see that around 70% of the predicted positive values are true positives and 81% of all negative predictions are true negatives. When one takes into consideration that about 80% of the data is negative (non-default) we see that the value for positive predictions are about three and a half time as high as that we would expect from random guessing, but when looking at the negative predictions we get around from what we would expect from random guessing. We also see that we get an accuracy of 0.805 at a threshold of 0.3, which mean we get an error rate of 0.195. This is somewhat higher than that found by Yeh et. al. of 0.18 on their validation data [1]. For neural networks we have that 61% of all positive predictions are true positives, and 0.84% of negative predictions are true negatives. So here we again see that the amount of positive predictions that are true positives is around 3 times higher that what it would have been from random guessing, while for the negative values it only a few percentage points. With the neural network we get an accuracy of 0.808, which gives an error rate of 0.192, which is close to the value of Yeh et. al. of 0.17 [1], but again somewhat higher. So we see that we get comparable, but slightly worse values for the accuracy than Yeh et. al. This could be because of the random way the test and training data is selected. Another possible explanation is that we use a simple SGD method to train our methods, while some more advanced method like adam optimization may make the results better, which is something we could try in further studies. For the neural network we could also have tried different numbers of hidden layers which we did not focus too much on here, but could be looked further at in future work.

From Tables I and II, we also see that our two methods preform quite differently at a given threshold. At 0.5 for instance, logistic regression actually gives fewer false positives and more true negatives. However it then also give more false negatives and fewer true positives. We see similar results for other thresholds. This indicates that for a given threshold, logistic regression preforms better on negative results, while the neural network preforms better on positive results. Howver comparing the two methods for a given threshold is not necessarily useful, as one can adjust the threshold until it matches the results of the other on the part where it originally preformed worse. This is essentially what is done in the ROC curve in Figure 3, which should indicate that if we try something like the method mentioned, the neural network should preform clearly better (except for low false positive rates).

In Figure 3 we have plotted the ROC curves for the classifiers. Here anything above the black line is better than random guessing, since this means there are more true positives per actual positive event, then there are false positive per actual negative event. A perfect classifier is then at the top left corner of the plot with no false positives and all positives correctly classified, which then would mean that all events are correctly classifies. We see that both classifiers preforms better than random guessing, regardless of which threshold one picks. We also see that the neural networks with a ReLU and ELU activation function in general out-preforms logistic regression. This agrees with what we saw when looking at the specific results in Tables I and II, and also what was found by Yeh et. al when looking at the area ratio [1]. While we do not plot the same type of plot with cumulative number of target data against total data, we have the ROC, which should give similar plots. So by comparing visually, we see that our ROC plots for logistic regression and neural network look roughly similar to those found by Yeh et. al. From our plot of the ROC curve, we also see that if a low false positive rate is the important characteristic of the classifier (values below 0.1), logistic regression preforms about as well as our neural network. However otherwise the neural network is somewhat closer to an ideal classifier. However we also see that with a sigmoid activation function, the neural network preform very similar to logistic regression. This is probably because with the sigmoid function, each preceptron functions similarly to logistic regression, and so the final result is also very similar. From Figure 3 we also see that ELU and ReLU preform very similar, with ReLU seemingly having a slight edge in performance. This is not necessarily a real effect however, since the two curves are very close, so this result may be random. Given that we have not have the standard deviation for the curves, we can not truly say that any of these results are not a result of randomness. However, since we have two results for two very similar methods that end up as close as they are in Figure3, it seems fairly unlikely that the difference between logistic regresison/neural network with sigmoid and neural network with ReLU/ELU are random fluctuations. However to confirm this, one could calculate the standard deviation in future work.

One thing to notice is when comparing the ROC plot for the credit card data with that of the cancer data, we notice that the models for the cancer data seem to generally work much better. This then seems to indicate that the cancer data is much more suited for classification. This result is also a strong indicator that our programs for logistic regression and neural networks work as they should, as a result like this is unlikely to happen accidentally.

Regression

Regarding the regression case, we found that the neural network in fact performed significantly better than the Ridge method given appropriate epochs of training. As we see from figure 5, the amount of training seems to depend on the value of the noise, where high noise may lead to too much training, most likely due to overfitting. Since we consistently used 500 epochs in our optimal parameter search, this might mean that the data corresponding to the lower noises may have been overfitted. With this in mind, it seems logical that the order of the polynomial decreases with higher noise, as to compensate for this overfitting that occurs by reducing the complexity of the input.

In order to better understand the change in the other parameters, we will consider their effect on the bias and variance of the model.

As we discussed in our previous article [2], one would expect that the bias decreases with the complexity of a model, while the variance increases. The decrease in bias corresponds to more aspects of the data being represented by a model, while the increase in variance shows a difference between predictions with different training data samples, which is a typical sign of overfitting to the training data. In the case of our Feed Forward Neural Network, we first see this when plotting against the amount of hidden layers in figure 6. More hidden layers would correspond to increased complexity in the model, and it is therefore as expected that the variance increases with more hidden layers, while the bias decreases. Although the variance matters more than the variance with higher hidden layers, we also see an increase in bias with higher complexity, which is not as expected. Additionally, we can see that the variance also starts out relatively high when only one hidden layer is present. The bias is still high, so this is probably not a sign of overfitting, but we rather believe that this is a sign of the randomness in the initialization of the model. Since all weights are intialized in randomly in a uniform distribution, one would expect that with fewer nodes, this random difference may have increased importance.

Of course, we could have used a seed for our random initialization, which could have given less calculated initial variance. However, this would poorly represent a real situation, since the innitial weights are in fact random, and it is important to keep in mind that with a bad initialization, one might not reach the optimal minimum as one could with a better random initialization. As we see from the plot, however, this effect may become smaller with more hidden layers. Again, adding too many hidden layers may cause overfitting, so this is a trade-off one should keep in mind when structuring a Neural Network.

Comparing to the amount of epochs in Figure 7, we also see some signs of overfitting after a sufficient amount of epochs, with an ideal value of about 500. It seems that increasing the epochs of training in some ways increases the complexity of the model. This makes sense if you consider that the not fully trained model still has some nodes with somewhat random weights. These nodes will thus not be entirely useful, and can be seen as inactive. With an amount of inactive nodes, it is clear that the complexity is less than with all nodes active, and thus the bias-variance tradeoff with regards to complexity can also be seen in this figure. Regarding the inactivity of nodes, there are in fact methods such as the leaky ReLU which randomly deactivates nodes, attempting to avoid overfitting. So utilizing this, we may have gotten better results in this case.

We note, however, that 100 data points and $\hat{\sigma} = 10^{-2}$ was necessary to clearly show this effect, so it may only be noticeable if there is a situation where a lack of data is apparent.

Regarding the effects of the learning rate η, we did not see any clear signs of a noticeable effect on the MSE. However, we again noticed a reduce in bias, and increase in variance for increased values, which might suggest an increase in complexity. Again using the alternative understanding of complexity as activated and still inactive random nodes, it indeed seems logical that higher learning rates lead to faster convergence to its final state where all nodes are active, and complexity is increased. In comparison, lower learning rates will leave some nodes in their "inactive" random states, decreasing the complexity.

If this interpretation is true, one would expect that with an increased amount of epochs, the variance levels would be more similar from the start as well, as lower learning rates η would also have time to converge. In fact, we would expect that higher learning rates may even overshoot the minimum, and not converge completely, while lower learning rates although slower, would eventually reach a minimum, be it local or global.

Because higher learning rates converge faster, but may not converge entirely, or may even diverge, many methods use an adaptive learning rate, which would be another possible improvement of our model of a feed forward neural network.

It is also important that the model converges to a global and not local minima, which the stocasthic gradient descent method is meant to ensure. In figure 9 we further investigated the effects of the SGD, and saw that although it performed better than the normal GD, it also seemed like it was possible for the batch size to become to small. In particular, we saw that although the bias mostly decreased for larger batch sizes, there was a slight increase in variance in the highest amount of bacthes, which again might indicate overfitting. Since the usage of only one batch can lead to some loss of detail, and only a local minima being reached, this indeed seems to correspond to a low complexity and high bias. Further, a small batch size might mean that effects of the relatively high noise is captured, and would logically result in the increased variance which may be what is visible. However, as the effect is rather small, we cannot conclude that this effect is not random.

Still, we seemed to have reached what essentially boils down to a trade-off between bias and variance with increased complexity with regards to the most important hyperparameters of the neural network.

In the case of the franke function with 400 data points, we seem to have managed to evaluate this trade-off well enough, to get better results than we did with methods of linear regression. However, the training of the FFNN takes longer time than these simpler methods, which must be kept in mind for cases with larger amount of data. The parameter search is indeed proportional to the training time, so if this is too long, using a neural network might not be feasible, as it needs to reach convergence for the performance to be better than methods of linear regression. Additionally, a property of deep neural networks with multiple hidden layers is that the model is a black box in the sense that one cannot really extract meaning from the way the numbers go from input to output except for the pure mathematical properties. In this case, we no longer model the Franke Function with an n'th order polynomial, where we find the desired coefficients, and we can no longer recreate the data only using such coefficients as we could for linear regression. Thus, although we allow for our model to be more complex, we lose some of its direct interpretation.

VI. CONCLUSION

We evaluated the performance of feed forward neural networks in both classification and regression cases. In the classification case, the NN seemed to perform slightly better than logistic regression on the credit card data by Yeh et. al. [1]. One exception for this was with the sigmoid function as the activation function, which preformed about as well as logistic regression. Looking at the ROC-curves however, we saw saw that the performance still lacked especially with a high amount of false negatives compared to true positive, i.e. a misclassification of the positive result corresponding to default of the credit card. For a given threshold, logistic regression actually performed better on negative data points, but if we compared logistic regression with a neural network with a more tuned threshold, the neural network should at worst be as good as logistic regression. As for the regression case, we applied our FFNN to the Franke Function, and did a parameter search to determine the optimal hyperparameters for the NN. Only varying one paramter at a time, we saved considerable computation time compared to a grid based search. Further plotting the dependence of the bias and variance on the different hyperparameters, we were able to see that varying the hyperparmeters mostly boils down to a trade-off between high bias at low complexity, and high variance at high complexity. This was only visible at few data points and low noise, however, so it may not be as relevant in a situation with plenty of data. Using 400 data points, the result we got from our optimal model performed significantly than the Ridge method when the noise was not too large. A variable learning rate and drop out nodes could improve the model even further, possibly preventing overfitting in the high noise case.

[1] I.-C. Yeh and C.-H. Lien, "The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients,"

[2] M. S. Ingstad and F. L. Nilsen, "Comparison of methods of linear regression," oct 2019.

[3] Hjort-Jensen M., "Data Analysis and Machine Learning: Logistic Regression," *Lecture notes*, 2019.

[4] Hjort-Jensen M., "Data Analysis and Machine Learning: Neural networks, from the simple perceptron to deep learning," *Lecture notes*, 2019.

[5] W. Wolberg and O. Mangasarian, "Multisurface method of pattern separation for medical diagnosis applied to breast cytology,," in *Proceedings of the National Academy of Sciences*, pp. 9193–9196, Dec 1990.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

VII. ADDITIONAL RESULTS

Can be found in the github repository https://github.com/Magnus-SI/FYS-STK3155