# Logistic Regression and Feed Forward Neural Network (FFNN) analysis on the MNIST and Franke's Function Dataset

Knut Magnus Aasrud
Philip Karim Niane
Ida Due-Sørensen

November 14, 2020

**Abstract**

This report considers a couple of datasets, namely the MNIST set of handwritten digits, the Wisconsin cancer dataset and a set generated from Franke's function. These have been analysed by utilizing our own feed-forward neural network, and logistic regression. For comparison, some parts use implementations from `scikit-learn` as well. The goal of this report is to study regression and classification problems, and to compare the performance of a neural network against more classic regression methods - as explored in Project 1.

The logistic regression seems reliable, with an accuracy score only slightly lower than the one produced by `scikit-learn`. The regression results of the neural network proved adequate at best, and its classification of the MNIST digits turned out unsatisfactory. Its here worth noting that we were hard pressed with time and that these results would most definitely be improved if we were able to further explore the best network structure for the task at hand, and to analyze its performance.

## 1 Introduction

Classification algorithms are an invaluable tool in our daily life with broad applications, such as medical diagnosis, image and speech recognition, fraud detection, spam detection, traffic prediction, automatic language translation and finance. In the field of machine learning, classification is considered an instance of supervised learning, or put differently, learning where a training set of correctly identified observations is available.

Regression problems is the other group of supervised learning. Both classification and regression problems have as goal to construct a model that can predict the value of the dependent attribute from the attribute variables. The difference

between the tasks is that the dependent attribute is numerical for regression and categorical for classification.

The purpose of this work is to gain an increased understanding of machine learning applications by studying both classification and regression problems. Thus, we develop a feed forward neural network (FFNN) code, and we also include logistic regression for the classification problems. The methods will be tested on the MNIST dataset. In addition we use FFNN to fit a data set generated using Franke's function. The results of the analysis on the Franke's function dataset will be compared to the results of our previous work on regression methods [1].

First, we review some important concepts needed to comprehend the discussion. Then, we briefly present how we have implemented the methods and present our results. Last, a critical evaluation of the various algorithms will be given.

# 2 Theory

## 2.1 Logistic regression

Logistic regression is a method for classifying a set of input variables $\boldsymbol{x}$ to an output or class $y_i, i = 1, 2, \ldots, K$ where $K$ is the number of classes. The review in this section is based on Hastie et al. [2, ch. 4], and the reader is referred to this book for a more detailed explanation of topic. The prediction of output classes which the input variables belongs to is based on the design matrix $\boldsymbol{X} \in \mathbb{R}^{n \times p}$ that contains $n$ samples that each carry $p$ features. We distinct between *hard* and *soft classification*, which determines the input variable to a class deterministically or the probability that a given variable belongs in a certain class. The logistic regression model is given on the form

$$\begin{aligned} \log \frac{p(G=1|X=x)}{p(C=K|X=x)} &= \beta_{10} + \beta_1^T x \\ \log \frac{p(G=2|X=x)}{p(C=K|X=x)} &= \beta_{20} + \beta_2^T x \\ &\vdots \\ \log \frac{p(G=K-1|X=x)}{p(C=K|X=x)} &= \beta_{(K-1)0} + \beta_{K-1}^T x. \end{aligned} \tag{1}$$

We consider the binary, two-class case with $y_i \in [0, 1]$. The probability that a given input variable $x_i$ belongs in class $y_i$ is given by the Sigmoid-function (also called logistic function):

$$p(x) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}}. \tag{2}$$

A set of predictors, $\boldsymbol{\beta}$, which we want to estimate with our data then gives the probabilities

$$p(y_i = 1|x_i, \boldsymbol{\beta}) = \frac{\exp\left(\boldsymbol{\beta}^T x_i\right)}{1 + \exp\left(\boldsymbol{\beta}^T x_i\right)} \tag{3}$$

$$p(y_i = 0|x_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|x_i, \boldsymbol{\beta}). \tag{4}$$

We define the set of all possible outputs in our data set $\mathcal{D}(x_i, y_i)$. Further, we assume that all samples $\mathcal{D}(x_i, y_i)$ are independent and identically distributed. Now we can approximate the total likelihood for all possible outputs of $\mathcal{D}$ by the product of the individual probabilities [2, p. 120] of a specific output $y_i$:

$$P(\mathcal{D}|\boldsymbol{\beta}) = \prod_{i=1}^{n} [p(y_i = 1|x_i\boldsymbol{\beta})]^{y_i} [1 - p(y_i = 1|x_i, \boldsymbol{\beta})]^{1-y_i}. \tag{5}$$

We want to maximize this probability by using the maximum likelihood estimator (MLE). By taking the logarithm of eq. (5), we obtain the log-likelihood in $\boldsymbol{\beta}$

$$\log P(\mathcal{D}|\boldsymbol{\beta}) = \sum_{i=1}^{n} [y_i \log p(y_i = 1|x_i\boldsymbol{\beta}) + (1 - y_i) \log(1 - p(y_i = 1|x_i, \boldsymbol{\beta}))]. \tag{6}$$

By reordering the logarithms and taking the negative of eq. (6), we obtain the *cross entropy*

$$\mathcal{C}(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left[ y_i \boldsymbol{\beta}^T x_i - \log\left(1 + \exp\left(\boldsymbol{\beta}^T x_i\right)\right)\right]. \tag{7}$$

The cross entropy is used as our cost function for logistic regression. We minimize the cross entropy, which is the same as maximizing the log-likelihood, and obtain

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\sum_{i=1}^{n} x_i (y_i - p(y_i = 1|x_i, \boldsymbol{\beta})) = 0. \tag{8}$$

The second derivative of this quantity is

$$\frac{\partial^2 \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta}^T} = \sum_{i=1}^{n} x_i x_i^T p(y_i = 1|x_i, \boldsymbol{\beta})(1 - p(y_i = 1|x_i, \boldsymbol{\beta})). \tag{9}$$

These expressions can be written more compactly by defining the diagonal matrix $\boldsymbol{W}$ with elements $p(y_i = 1|x_i, \boldsymbol{\beta})(1 - p(y_i = 1|x_i, \boldsymbol{\beta}))$, $\boldsymbol{y}$ as the vector with our $y_i$s values and $\boldsymbol{p}$ as the vector of fitted probabilities. We can then express the first and second derivatives in matrix form

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{p}) \tag{10}$$

$$\frac{\partial^2 \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta}^T} = \boldsymbol{X}^T \boldsymbol{W} \boldsymbol{X}, \tag{11}$$

also known as the Jacobian and Hessian matrices, respectively. We will use the stochastic gradient descent (SGD) (section 2.2) to find the optimal parameter $\boldsymbol{\beta}$.

## 2.2 Stochastic Gradient Descent

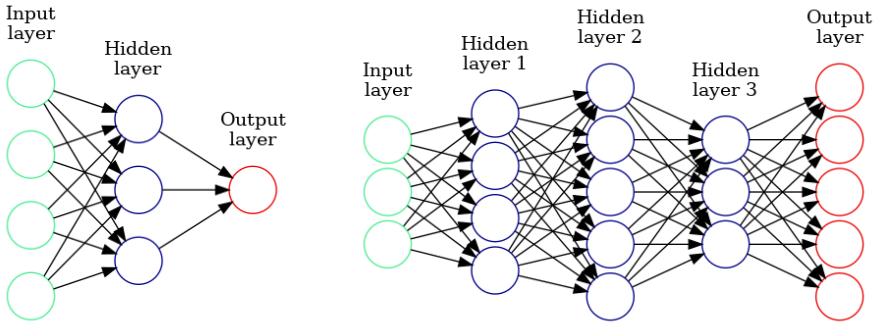
*Gradient descent* describes the process of finding a local minimum of a function (the cost function, in our case) by following the negative value of the gradient at each point, stepwise. *Stochastic gradient descent* or SDG is a way of increasing the numerical efficiency of this process, by doing this process stochastically.

This involves randomly dividing the training data into a given number of *mini batches*. For each mini batch, the gradient is found by averaging the gradient value each mini batch sample has. Then the weights and biases are updated (take a step down the "slope") and the process is repeated for the rest of the mini batches. The updating done at each mini batch is expressed mathematically as

$$w \to w' = w - \frac{\eta}{m} \sum_{i}^{m} \nabla C_{i,w}$$

$$b \to b' = b - \frac{\eta}{m} \sum_{i}^{m} \nabla C_{i,b},$$

where $m$ is the number of data points in the mini batches and $\nabla C_i$ is the gradient of the cost function at each individual data point. After exhausting all the training data, we have finished a so-called *epoch*, of which we can perform as many as necessary.




(a) (b)

Figure 1: Schematic diagram of a neural network with: (a) four inputs, one hidden layer of three neurons and one output layer with one neuron. (b) three inputs, three hidden layers (four, five and three neurons) and one output layer with five neurons. Note that we have only fully connected layers.

## 2.3 Neural networks

Neural networks is a machine learning technique vaguely inspired by our understanding of how networks of neurons function in the brain. There are several types of neural networks. In this work, we study a multi-layer perception (MLP). The MLP is characterized by one or more *hidden layers* between the input and output layers. Each hidden layer consists of several *neurons* (also called *nodes*). The neurons in each layer is not connected to each other, but all nodes of a layer is connected to every neuron in the previous layer. We say that we have a *fully connected layer*. Figure 1a shows an example of a neural network used for binary classification, while fig. 1b display an example of a neural network used for a multi-class classification problem. In both sub-figures of fig. 1, we see that the inputs are fed through the network and processed, resulting in an output.

### 2.3.1 Forward feeding

As we can see in both example network diagrams in fig. 1, each neuron has one or multiple inputs. In the following, we consider the $j^{\text{th}}$ neuron in the $l^{\text{th}}$ layer. Each of the inputs is multiplied by a weight, $w_j$, when passed into a neuron, and the result is called $z_j$. An activation function $a(z_j) = a_j$ evaluates the signal, $a_j$ becomes the output from each neuron. We denote the weight associated with the input from the $k^{\text{th}}$ neuron in the previous layer $w^l_{jk}$. Further, wee add a bias, $b_j$ to each of the neurons in a hidden layer to prevent outputs of only zeros. By summing the weighted inputs and bias, and feeding this to a function $f$, we obtain the activation $a^l_j$

$$a_j^l = f^l(z_j) = f^l\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right).$$

Using matrix notation, the activation for the whole layer $l$ can be written more compactly as

$$\mathbf{a}^l = f(\mathbf{w}^l \mathbf{a}^{l-1} + \mathbf{b}^\mathbf{l}). \tag{12}$$

The activation is fed forward as input to the neurons in the next layer.

There are several possible choices of activation function. Which activation function to chose depends on the type of data set, how the data set is scaled and whether we use the neural network for classification or regression. In a feed-forward neural network, we require non-constant, bounded, monotonically-increasing and continuous activation functions. One can design a network with different activation functions for each hidden layer, but in this work we study networks with the same activation function for each layer.

Common choices for the activation functions are the sigmoid function (eq. (2)), the Rectified Linear Unit (ReLU) eq. (13), the Leaky ReLU, the tanh-function and the softmax function. For classification problems, the sigmoid function is often used in introductory examples and works well for binary classification problems. For multi-class classification, the softmax function is often preferred because it forces the sum of probabilities for the possible cases to be one. For regression, the ReLU and its variants are commonly used.

$$f(z) = \begin{cases} 0 & \text{for} \quad x < 0 \\ z & \text{for} \quad x \geq 0 \end{cases} \tag{13}$$

$$f(z) = \begin{cases} z & \text{for} \quad z \geq 0 \\ 0.01z & \text{for} \quad z < 0 \end{cases} \tag{14}$$

### 2.3.2 Backpropagation

In order to find the aforementioned gradient of the cost function, a method called *backpropagation* is used. This method revolves around calculating the gradient one layer at a time, starting with the last layer's activation, comparing it to the expected values and from there working backwards through the layer structure to find the total gradient.

Starting with a single data point $(\mathbf{x}, \mathbf{y})$, we feed forward through the neural network to find the predicted value, given the current weights and biases. This

prediction will most likely have some error, defined by the cost function (the details of which will be discussed below), and this error is what we want to find. We define it as the derivative of the cost function with regards to the individual weights,

$$\frac{\partial C}{\partial w_{ij}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{ij}^l} = \delta_j^l a_i^{l-1,T}.$$

Here we see that we've defined $\delta_j^l$ as the $j$-th error in the $l$-th layer. The $z$'s are the weighted inputs to the activation function and the $a$'s are the final activations of each neuron. For the biases, things are much simpler.

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

So now we only need to find the errors $\delta_j^l$ to find the complete gradients for each data point. For the last layer, this is found easily enough, by applying the chain rule

$$\delta^L = \frac{\partial C}{\partial z^L} = C'\left(\sigma(z^L)\right)\sigma'(z^L)$$

where $\sigma$ is the activation function and $L$ is the last layer of the network. To do this further back through the network, we only need some clever chain rule trickery, since we know the gradients form the succeeding layer.

$$\delta^l = \frac{\partial C}{\partial z^l} = \frac{\partial C}{\partial z^{l+1}}\frac{\partial z^{l+1}}{\partial z^l} = \delta^{l+1}w^{l+1,T}\sigma'(z^l)$$

So doing this for all layers, we can find the derivative of the cost function for every weight and bias, which is what the complete gradient for that particular data point consists of. This procedural stepping backwards through the layer to find the complete gradient is what gives the algorithm it's name.

## 2.4 Quality of models

We measure the performance of the different models used in this work. For classification, we measure how accurate the predictions given by the model are by the so called accuracy score. The accuracy score is given by

$$\text{accuracy} = \frac{1}{n}\sum_{i=0}^{n} I(t_i = y_i), \tag{15}$$

where $n$ is the number of samples, $t_i$ represents the target output and $I$ is the indicator function, which returns 1 if $y_i = t_i$ and 0 otherwise.

For regression, we use the MSE and R2-score to measure the performance.

## 2.5 Franke's function

Franke's function is a function which is often used to test different regression and interpolation methods. The function has two Gaussian peaks of differing heights, and a smaller dip [3]. Therefore the dataset is perfect for studying how a neural network can be used to predict the well known function. It's expression is

$$\begin{aligned} f(x,y) = \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \\ + \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) - \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right). \end{aligned} \tag{16}$$

In the defined interval $x, y \in [0, 1]$. For more information about Franke's function see [1]. An illustration of Franke's function can be seen in figure 2

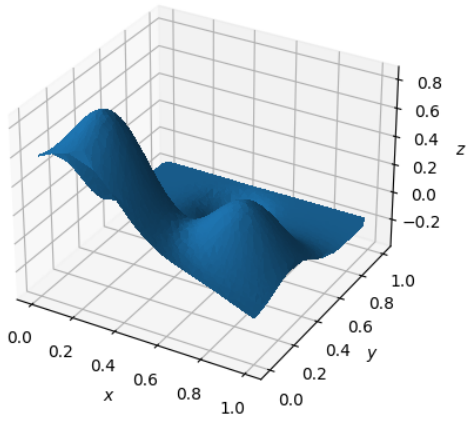



Figure 2: Shape of Franke's function which will be used as a interpolating goal

# 3 Method

## 3.1 datasets

It is always useful to study different datasets and see how different methods work on different datasets. That is why in this article, we study three different datasets. The first dataset is generated from the two-dimensional Franke's function. The second dataset is the well known MNIST dataset, while the last dataset is the Wisconsin cancer data which can be solved as a binary problem.

### 3.1.1 Franke's function

The dataset occuring from Franke's function is produced and implemented the same way as explained in the theory and [1].

### 3.1.2 MNIST

The famous MNIST dataset is a collection of handwritten numbers, as $28 \times 28$ grayscale images. It comes in two sets, a training set with $60,000$ images, and a testing set with $10,000$ images. In this report, we will model the inputs as a $28 \times 28 = 784$-dimensional vector, and the output as a 10-dimensional state vector, with each dimension representing the corresponding digit.

### 3.1.3 Wisconsin cancer data

The Wisconsin cancer dataset is a dataset providing 30 predictors for 569 patients each. Of the 569 patients, 357 possess a cancerous tumour while the 212 remaining patients do not possess cancerous tumors.

The dataset is easily accessed using scikit-learn. The design matrix is a 569x30 matrix while the target vector is a vector containing 569 binary values. A patient having the target value 1 represents a cancerous tumour, while the value 0 represents a non cancerous tumour. More info on how to access and use the dataset can be studied at [4].

## 3.2 Logistic regression

The implementation of the logistic regression is implemented pretty straight forward. The logistic regression functions is set up in a class containing the most important functions like the SGD, a prediction function and the learning rate.

The clue within machine learning algorithms is the need to train a model. The whole dataset is naturally split into test and training parts using standard scikit-learn functions. This is followed by a scaling of the data by subtracting the mean to adjust for data points with irregular values, using the same library. In the logistic regression class, the training dataset used to train the model is sent into

the SGD function within the class. After the training is done, the prediction is ready to be set in motion. The test data is sent into the prediction function, which returns a predicted response vector for the dataset. Which then can be used to calculate the accuracy.

Logistic regression was also implemented and performed on the same dataset using nothing but the scikit-learn library. This was to see how close the self written code can compare to the regression accuracy achieved by scikit-learn.

### 3.3 Neural network

The neural network is implemented as a class containing important algorithms like the training algorithm SGD, back propagation, feed forward function; that feeds forward the input X and stores all the layer's activation's in a list, with the possibility to return the weighted inputs, which are useful in the backpropagation algorithm.

The algorithms are implemented according to the theory about neural networks. The neural network is also implemented with the option to choose the amount of layers and neurons within each layer. This is really practical when switching between datasets. This is because the problem to be solved often gives different amounts of neurons in the output layer logical to use. For classification of numbers, it was logical to use an output layer with ten neurons, one for each digit. To classify binary problems there would be enough with one neuron as the output layer. Either the neuron is "yes" or "no".

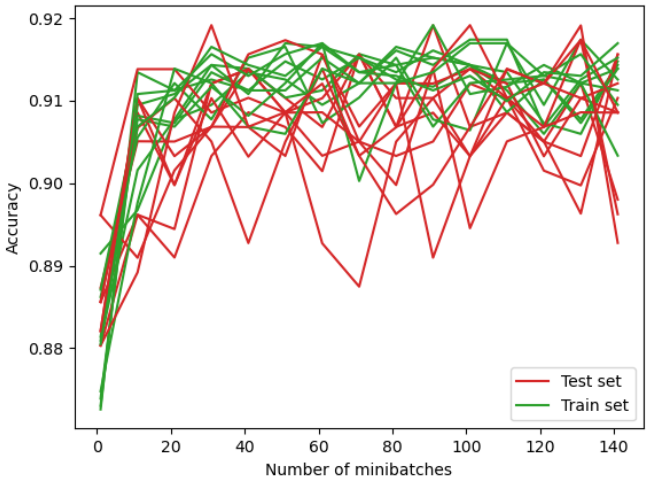
The neural network is having a variety of different variables to be chosen. The parameters are the amount of epochs, size of mini-batches that is sent into the SGD function, the learning rate, activation function, the cost function of the network and the activation function for the last layer.

## 4 Results

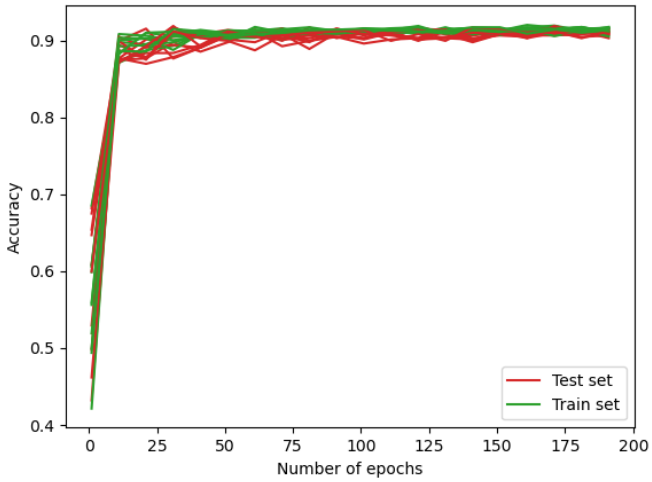
### 4.1 Classification

#### 4.1.1 Logistic regression on the cancer dataset

The classification problem with the breast cancer dataset was done by sorting the patients into two piles. One with a cancerous tumour and one without a cancerous tumour. As said in section 3.2 the data is sent into the SGD algorithm. Along with the data, there is two other variables sent into the function, amount of epochs and number of mini batches. Both the two variables have an important role on the accuracy. Therefore the accuracy was plotted as functions of the two variables, showing optimum parameters without increasing the computational complexity too much. The results from the accuracy plots can be seen in figure 3. This was calculated with step sizes of 10 for both epochs and mini batches in the intervals [1-150] and [1-200].




(a)




(b)

Figure 3: In both figures the calculations have been done ten times to secure a clear overview of the trend, due to some random variables as a result of the stochastic approach. (a) Plot of the accuracy vs. the number of epochs in logistic regression with constant epoch=110 (b) Plot of the accuracy vs. the number of mini batches in logistic regression with constant mini batch=40

Regarding the classification of the dataset, the chosen number of mini batches, and amount of epochs was set to 35 and 140 respectively. This was because the accuracy stops to increase around these values in figure 3, without being too computational complex.

The accuracy of the self written code applied to the Wisconsin breast cancer dataset with 35 mini batches and 140 epochs gave an accuracy of 0.965. The same dataset classified by scikit-learn gave an accuracy of 0.982.

Testing different amount of epochs was also done and timed to see how the increase of epochs afects the time to run the algorithms. The result of this can be seen in table 1.

Table 1: The time and accuracy to run the logistic regression methods with different amounts of epochs. The amount of mini batches is set to 35. The time is started when the function is called, and ends when the function is done.

| | Epochs= 140 | Epochs= 750 | Epochs= 15000 | Scikit-learn |
|---|---|---|---|---|
| Accuracy % | 96.5 | 98.2 | 97.7 | 98.2 |
| Time (s) | 0.1562 | 0.8736 | 16.57 | 0.0030 |

#### 4.1.2 Neural network for classification

Training our neural network with the MNIST dataset, we've classified hand-written digits with low success. Figure 4 shows 9 randomly selected samples and our prediction of them:

### 4.2 Regression

#### 4.2.1 Stochastic gradient decent

Figure 5 depicts the MSE as a function of the learning rate.

Figure 6 depicts the MSE as a function of the number of epochs.

When comparing our implemented SGD regressor to SkLearn's SGDRegressor and Ridge, we see from table 2 that the lowest MSE value is obtained for SkLearn's Ridge function.

Table 2: Obtained MSE values using different methods. The lowest MSE value is obtained using SkLearn's Ridge function.

| | SkLearn SGDRegressor | SkLearn Ridge | SGD OLS |
|---|---|---|---|
| MSE | 0.015 | 0.008 | 0.013 |

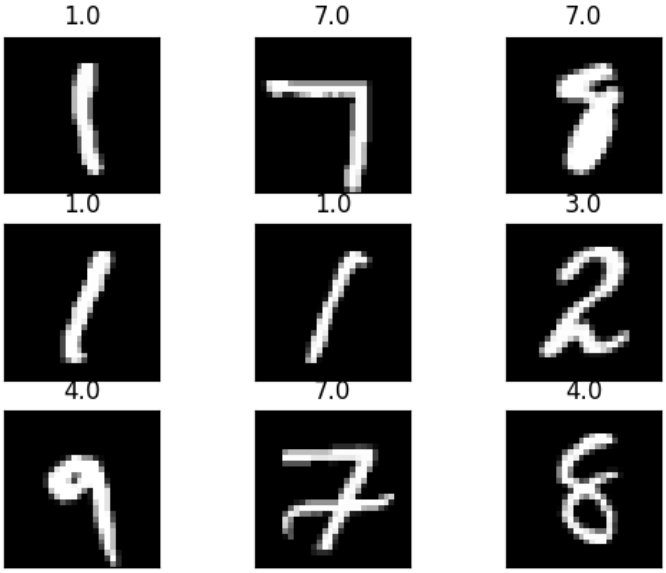



Figure 4: 9 randomly selected samples from the MNIST testing set and our neural networks' of them.

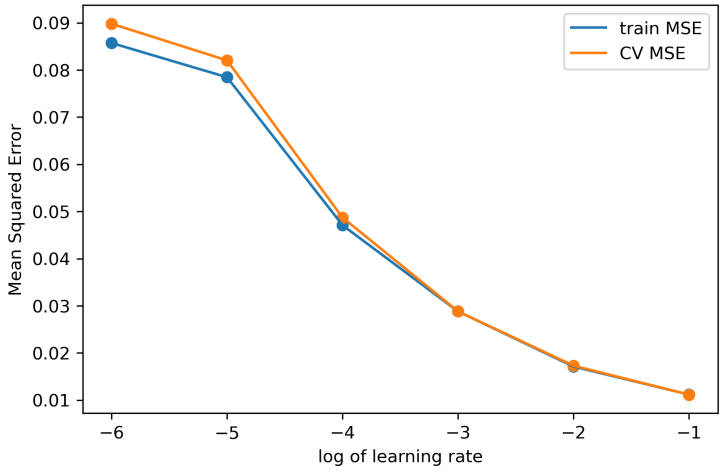



Figure 5: The MSE as a function of $\log_{10}(\eta)$ for $d = 3$, 1000 epochs and a batch size of 10.

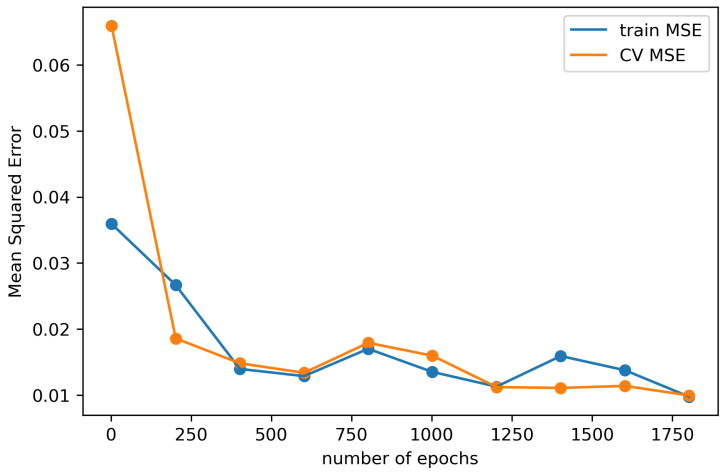



Figure 6: The MSE as a function of the number of epochs for $d = 3$, learning rate of 0.1 and a batch size of 10.

#### 4.2.2 Neural network for regression

Figure 7 shows the approximation of Franke's function by performing regression with the nerural network, plotted in 3D. The plots have been calculated by using different activation functions. The used design matrix goes up to degree 12.

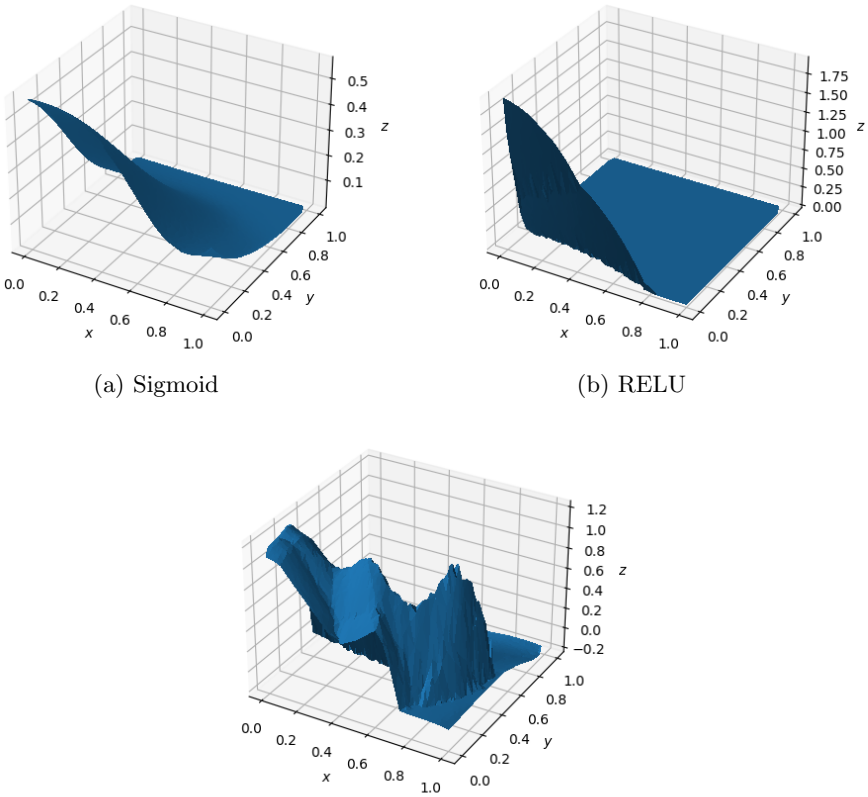
The activation function that performed best was the leaky RELU function, followed by the sigmoid function. The worst activation function was the ordinary RELU activation function.

# 5 Discussion

## 5.1 Linear regression performed on Franke's function

Linear regression was performed on the Franke's function dataset by using a neural network and different activation functions. The leaky RELU as an activation function performed best which is understandable thinking about the fact that the sigmoid is a logisitc function, while the RELU is often used for regression.

The regression done by the neural network was a bit worse than the results achieved in project 1. To see the results from project 1 please have a look here [1]. In project 1 the regression was done by ordinary least squares and ridge regression. Then the parameters were optimized to give the best possible result. The fine thing about numerical algorithms is that approximately all algorithms have the possibility to fine tune parameters and increase the computer power to




(a) Sigmoid

(b) RELU

(c) Leaky RELU

Figure 7: Prediction of Franke' function using a neural network, with different activation functions.

make the algorithm achieve better results. The case is the same for this neural network, for example there would be possible to add more layers and nodes to make the neural network perform better.

## 5.2 Logistic regression compared to scikit-learn

After testing the logistic regression code and compared it to the logistic regression done by scikit-learn, it shows that that the difference is quite small. The logistic regression done by scikit-learn managed to classify 98.2% of the data into the right class. On the other hand the self written algorithms managed to get 96.5% of the cases right, which is quite good. Keeping in mind that the whole dataset consists of 569 patients, making the self written code placing 549 of the patients into the right class. Scikit-learns 98.2% made 559 correct choices

which is only 10 more. Even though the accuracy of 96.5% was not the best accuracy possible, it is assumed in this paragraph because the amount of epochs and mini batches was chosen as a optimum choice regarding the computational time and accuracy.

It is possible to increase the accuracy by using more computational effort, a possible way of doing this is to increase the number of epochs, which makes the data go through the model more times. Since the cancer dataset only consists of 569 patients, this opened up the possibility of testing the effects of increasing the amount of epochs up to a ridiculous amount. In table 1 it is easy to see that increasing the epochs, increases the accuracy up to a certain sweet spot, then the accuracy decreases because the model is overfitting, which can be read more about in [1]. It is also important to notice the large increase in computational time when increasing the epochs. By increasing the 140 epochs by a factor of 5.4 and 107, the time increased by a factor of 5.6 and 106 respectively, which makes the time approximately proportional with the amount of epochs. Even though the best result was achieved by increasing the time by a factor of 5, this increase could get very large when working with a large dataset. It is also important to notice that the fast execution of scikit-learns calculations, probably was due to the dataset being really small.

### 5.3 Classification of MNIST dataset, using neural network

Due to time-constraints, we did not get to analyze our NN's performance with the MNIST dataset and find the ideal network structure. As a consequence, the results of our classification were lamentable, to say the least. The network seems to be classifying 1s, 4s and 7s pretty well, but this only anecdotal evidence. Given more time, we would probably utilize a more optimized (and importantly, speedy) implementation of the feed-forward neural network to find an ideal layer structure before using it on our own - much slower - neural network code. This would also allow us to do proper validation of our predictions and see how fast they converge in accuracy.

## 6 Conclusion

The aim of this work was to gain a further understanding of machine learning concepts and tools by studying logistic regression along with a feed forward neural network.

### 6.1 Classification

After applying the logistic regression model and and the feed forward neural network on the different classification datasets, we conclude that the neural network need a bit more fine tuning to give satisfactory results for classification cases, while the logistic regression model gives quiet presentable results.

### 6.2 Regression

Regarding the regression, the OLS and Ridge method seem to be the best methods when solving regression cases. This might not always be the case in view of the fact that the feed forward neural network has some elbow room for optimization.

### 6.3 Prospect for the future

The main prospect for the future would of course be to fine tune the neural network and try out even more neurons and hidden layers as far as the computing power approves it.

An interesting prospect for the future could also be to try these methods on more complicated and larger datasets, to see how they fare in other scenarios.

The future work of the logistic regression model would naturally be to push the accuracy score even closer towards the 100%.

## References


[1] K. M. Aasrud, P. K. Niane, and I. Due-Sørensen, *Applying regression and resampling techniquesto norwegian terrain data with franke'sfunction as test function*, 2020. [Online]. Available: `https://github.com/kmaasrud/reg-resample-fys-stk4155/blob/master/doc/main.pdf`.

[2] T. Hastie, R. Tibshirani, and J. Friedman, *Elements of Statistical Learning: Data Mining, Inference, and Prediction*, eng, ser. Springer series in statistics. New York: Springer, 2009, ISBN: 0387848576.

[3] R. Franke, *A Critical Comparison of Some Methods for Interpolation of Scattered Data*, eng. 1979.

[4] 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. [Online]. Available: `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html`.

# A Appendix

## A.1 Source code

All the source code is located in this GitHub repository.