# Project 2 - FYS-STK4155

Elias Roland Udnæs, Jacob Lie & Jonas Thoen Faber
(Dated: November 13, 2020)

In this project we extend our knowledge in machine learning by introducing the Stochastic Gradient Descent (SGD) algorithm and building a feed forward neural network (FFNN) with backpropagation. Finally we implement logistic regression with the neural network. The methods will be used to predict a data set generated by the Franke function or MNIST data (handwritten numbers). Evaluations of the models are performed with cost functions mean squared error (MSE) and cross-entropy. Stochastic gradient descent without a penalty parameter resulted in an MSE of 0.0102. Introducing a penalty parameter did not improve the model, with the best observed MSE being 0.0219. The neural network generally produced better results with a high learning rate ($\eta = 1$) on the Franke data. We did not find a learning schedule that improved the results compared to a constant learning rate. Later, the network was trained to recognise hand-written digits with softmax as activation function in the output layer. With the cross-entropy as cost function, the network (one hidden layer with 100 neurons) scored at best with an accuracy of 98.1 % with $\eta = 0.1$, $\lambda = 1$, mini-batch size of 100 and a training schedule of 200 epochs. The logistic regression method computed a best accuracy of 96.7 % with equal $\eta$, $\lambda$, activation functions but cross-log entropy as cost function.

## I. INTRODUCTION

Since the birth of the world wide web in 1991, the amount of information available to common people has exploded. Billions of bytes with data are generated each day. Naturally, the huge growth of data sets produced each day requires efficient analysis tools. Where standard regression methods like ordinary least squares (OLS) analysis might be suited for finding patterns in data sets with a limited amount of information, these methods are quickly outclassed when it comes to big data. With several features and millions of entries, OLS requires matrices that can be too large for our computers to handle.

Depending on the type of data, analysing it correctly can have very profitable outcomes. Therefore, we need to come up with something else when OLS falls short. This is where stochastic gradient descent (SGD) comes into play. The SGD method is not dependent on working with the entire data set at once, and can comfortably handle big data.

However, things are not always this easy. Stochastic gradient descent does not solve all our problems, like managing complex decision making and classification problems. Say a car company want to build an autonomous car that can drive safely in a crowded city. Then the car needs to distinguish between e.g. a traffic light and a human being. So how does the car know which of these two objects it is approaching?

The previous example falls straight into the classification problem, where regression and gradient descent does no good. Although stochastic gradient descent cannot be directly applied to these kinds of problems, it can be implemented into a neural network that handles classification. A neural network can be capable of complex decisions, and can separate a human from a traffic light.

Our goal in this project is to explore cases where standard regression methods falls short, and using gradient descent methods capable of analysing these cases. We will produce a neural network capable of making complex decisions to predict outcomes based on irregular inputs. The neural network will be trained by introducing a large amounts of data such that it will adjust the parameters that builds up the network by itself.

First off we will compare SGD with OLS on the Franke data set. Then we will model the Franke function with our neural network. When we observe good results from this model we extend our scope to classification problems, with handwritten digits as our data set. We will train the network on the handwritten digits and measure the accuracy the network predicts on the test data. Lastly, we apply the logistic regression method to predict handwritten digits.

## II. THEORY

### 1. Gradient Descent

The goal of machine learning is to minimise the cost function. Looking at our paper for regression [1] we recall:

$$\frac{\partial C}{\partial \beta_i} = 0.$$

Where $C$ is the cost function, and $\beta_j$ is the parameters for the regression line $\tilde{y_i} = \beta_0 + \beta_1 x_{i,0} + \cdots + \beta_n x_{i,n}^n$. However, as we discussed, this equation might not have a solution. We therefore introduce Gradient Descent.

Gradient descent is a method of estimating weights ($\beta$) to a model in regression analysis or neural networks. Given a data set with inputs $\vec{x}$ and outputs $\vec{y}$, we want to find a model predicting events $\tilde{y}$. The input set $\vec{x}$ has $n$ data points and $m$ features, making it a $n \times m$ matrix. The features depend on the thing we wish to observe. Do you want to predict if a tumor is malign or benign, then features might be color, volume, shape etc.

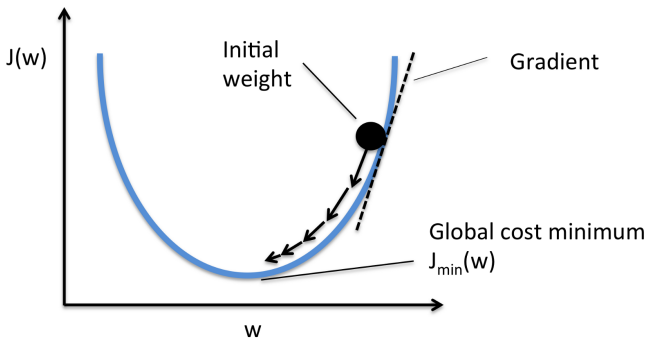


Figure 1. visualisation of Gradient Descent, where you choose an initial set of weights and compute the gradient of the cost function. Then following the gradient down to the global/local minimum. [2]

The output vector $\vec{y}$ consist of the points you want to predict.

First we choose a random set of weights, equivalent to number of inputs. The input is sorted in a feature matrix. The model we wish to optimise then becomes.

$$\tilde{y} = \boldsymbol{X}^T \boldsymbol{\beta}.$$

You compute the cost with this set of weights, and then find the gradient of the cost $\nabla_\beta C(\beta) =$

$$\begin{bmatrix} \frac{\partial C}{\partial \beta_0} \\ \frac{\partial C}{\partial \beta_1} \\ \vdots \\ \frac{\partial C}{\partial \beta_n} \end{bmatrix}$$

The gradient tells us where we have the steepest slope of the curve. It is visualised in figure 1. Expanding to multivariate (multifeature) cases is easy enough, and requires that we differentiate w.r.t. all variables. Each individual weight is updated in such a way, that we follow this gradient to a local minimum in the cost function:

$$\beta_{i+1} = \beta_i - \eta \frac{\partial C}{\partial \beta_i}.$$

Where $\eta$ is the learning rate. This decides how fast we wish to go in direction of the minimum. Choosing the right learning rate is crucial in machine learning. As choosing a rate too small, might result in the model moving so slowly and you never reach the minimum. Or by choosing a rate too large, you might overshoot the minimum, and each iteration you oscillate between lower and higher costs, not converging towards any minimum.

A solution to this problem might be to implement a learning schedule. Where the learning rate is large in the beginning, but as the cost moves closer to it's minimum, you reduce the learning rate.

*Momentum Gradient Descent*

Gradient descent might be computational expensive, using a method called Momentum Gradient Descent, we can increase the pace at which the model moves down the cost function. The algorithm is:

$$v_{i+1} = \gamma v_i + \eta \frac{\partial C}{\partial \beta_i},$$

$$\beta_{i+1} = \beta_i - v_{i+1}.$$

The $\gamma$ factor decide how large you want the acceleration towards the minimum to be. We increase the change to the weight, without changing the learning rate $\eta$. We might still overshoot the minimum, but the velocity $v_{i+1}$ decreases as $\eta \frac{\partial C}{\partial \beta_i}$ decreases. Using the analogy of a ball down a hill, we see that the velocity $v_{i+1}$ is large in the beginning, but as acceleration $\eta \frac{\partial C}{\partial \beta_i}$ decreases, so will the velocity, and we'll eventually reach the bottom of the hill.

*Feed Forward Neural Network with Back Propagation*

A neural network is a set of neurons that predict something by tuning a set of weights and biases. The network is visualised in figure 2. The neural network has an input layer, containing datapoints and features. The data points might be people (n) and the features, as mentioned in II 1, might be weight, age and sex (x,y,z). There are one neuron per feature in the input layer, and one output corresponds to one datapoint. However, we stack all the datapoints on top of each other in a feature matrix, and pass either the whole or parts of the matrix through the neural network. e.g. for a case we introduced above, we might want to predict if a person has diabetes or not. The feature matrix would look like this:

$$\begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & z_n \end{bmatrix}$$

Each neuron are assigned a weight. The neurons then calculates a combined weighted sum:

$$w^{(1)}_{0,0}\, x^{(0)}_0 + w^{(1)}_{0,1}\, y^{(0)}_0 + w^{(1)}_{0,2}\, z^{(0)}_0 + b^{(1)}_0.$$

Where $w^1_{0,0}$ indicates the weight of the $0^{th}$ input and the $0^{th}$ neuron, in layer $^{(1)}$. $x^{(0)}_0$ indicates we're on the neuron corresponding to the feature $x$. The subscript is the datapoint and the superscript is the layer. The weights decide how strong the activation in the following neuron is. The $b$ is bias, and enables the network to shift the weighted sum either up or down. Imagine fitting the line $y = b + ax$. $b$ enables us to move the line either up or down, to create a better fit. Bias has the same effect.

It is normal to put the weighted sum through an activation function $\sigma$, explained in II 2. **Important:** We will, through the rest of the paper, use the notation $a = \sigma(z)$. $z$ is the weighted sum $w_{j,k}^{(l+1)} a_k^{(l)} + b_k^{(l+1)}$.

The weighted sum is passed onto each neuron in the following (hidden) layer. They calculate their sum and pass it on to either the output layer or another hidden layer. As argued by Pankaj Mehta [3], number of hidden layers is important as to not overfit the data. Increasing number of hidden layers adds more complexity to the model, and might help learning more complex patterns between the features of the data. Adding hidden layers combined with activation functions might be wise, if you have a large feature space.

Lastly the network passes the final sum onto the output layer. The number of neurons are decided by what you wish to predict. If you want to predict a point in $\mathbb{R}^1$, then there is only one output neuron. Or you might have a classification problem where you want to decide if someone is angry, sad or happy. Then you have three neurons representing each category.

The process of passing input into a neural network and calculating weighted sums until you reach the output layer is called feed forward. Feed forward alone is somewhat pointless, because you often initialise random weights, therefore the neural network has no idea of how the data behave.
The solution is to introduce the back propagation algorithm.
The essence of back propagation is to use the average cost of an initial feed forward pass with all datapoints, to compute gradients that we can use to update the weights and biases of the neurons. If you have $L_3$ layers, you want to see how the weights and biases in layer $L_2$ and $L_1$ influenced the final output cost. Then making another feed forward and repeat the process. This is where the gradient descent, which we discussed in II 1, comes in. Our explanation is heavily inspired by Michael Nielsen's book *Neural Networks and Deep Learning* [4]. We want to find the gradient of the cost function $\nabla_a C$. The cost function is dependent on the layers $a^l = \sigma(z^l)$ that produced the weighted sum $z^l$. The weighted sum is again dependent on the weights and biases $w$ and $b$. We therefore need to use the chain rule to find the total derivative of $C$:

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w} \quad \& \quad \frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial b}$$

We'll first focus on $\partial C/\partial w$. If the cost function is *Mean Squared Error*, which is defined as $1/m \sum_{i=0}^{m} (\tilde{y} - y)^2$, with $\tilde{y}$ being the model, in our case $a^l$. This is easy enough to differentiate w.r.t. a:

$$\frac{\partial C}{\partial a^l} = \frac{2}{m}\sum_{i=0}^{m}(a_i^l - y_i). \tag{1}$$

We can also find $\partial a/\partial z$. We know $a^l = \sigma(z^l)$, so we get:

$$\frac{\partial a}{\partial z^l} = \sigma'(z^l). \tag{2}$$

The final expression $\partial z/\partial w$ is also trivial, as $z^{l+1} = w^{l+1}a^l + b^{l+1}$. So the derivative is simply $a^l$.
This works for the last layer, but we want to propagate backwards from the last output layer to the first hidden layer. Combining 1 and 2 we get $\partial C/\partial z_j^l$. But we want an expression that has the values from the $l+1$ layer, not $l$. Remember we're propagating backward. Imagine the figure 2, you're on the hidden layer and want to change your weights and biases based on the cost computed by the output layer. You simply want to calculate:

$$\frac{\partial C}{\partial a_k^{l+1}}\frac{\partial a_k^{l+1}}{\partial z^{l+1}} = \frac{\partial C}{\partial z_k^{l+1}}.$$

For simplicity we'll not write the sums $\sum_j$ or $\sum_k$, but k will represent neurons in layer $l$ and $j$ will represent neurons in layer $l+1$.

Applying the chain rule to $\partial C/\partial z_j^l$, we obtain $\partial C/\partial z_k^{l+1}$.

$$\frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l}.$$

We know that $z_k^{l+1} = w_{k,j}^{l+1} a_j^l + b_k^{l+1}$. We have already defined $a_j^l$ as $\sigma(z_j^l)$, so finding $\partial z_k^{l+1}/\partial z_j^l$ is trivial:

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{k,j}^{l+1}\sigma'(z_j^l). \tag{3}$$

We still need $\partial C/\partial z_k^{l+1}$, but remember we have the initial cost from the feed forward pass, so these values are already known.

Now we have $\partial C/\partial w$, but as mentioned, the cost is also dependent on the bias.

We need to find $\partial C/\partial b$. However, the expression is almost equal to $\partial C/\partial w$. The only exception is $\partial z/\partial b$, which is simply 1.
Now we have all the expressions we need, and are able to update all the weights and biases:

$$w_k^l = w_k^l - \eta\frac{\partial C}{\partial w_k^l} \tag{4}$$

$$b_k^l = b_k^l - \eta\frac{\partial C}{\partial b_k^l} \tag{5}$$

### 2. *Activation Function*

Let's say you wanted to predict if someone were likely to buy life insurance. You might add features such

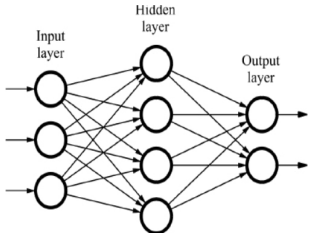


Figure 2. Visualisation of neural network with three layers. [5].

as age, annual income, preexisting conditions etc. Already you have a three dimensional feature space. However, in a neural network you calculate the weighted sum $\boldsymbol{z^{l+1}} = \boldsymbol{w^{l+1}a^l} + \boldsymbol{b^{l+1}}$. This will almost perfectly predict a linear, quadratic or cubic line. But if the correlation between the features are non linear, we will never bee able to predict the pattern well enough. We therefore need to add non-linear activation functions to the hidden layers.
Furthermore, the values of the neurons are not restricted, and might take on high values that are numerically unstable. The activation functions can be used to squeeze the weighted sum into a reasonable value.

### 3. *Logistic Regression*

In our paper about regression methods [1] we discussed ways of fitting data by tweaking $\beta$ parameters. We discussed ordinary least squares, ridge and lasso regression. However, what if you want to predict if something is e.g. true or false (0 or 1), rather than fitting a point in space. Then your function is discrete, not continuous. Logistic regression is used for this task, as it takes an input, and predicts which class the input should belong to. You can view it as a neural network without any hidden layers. In section II 2 we discussed that activation functions may be used to find non linear correlations between features. This is the case for logistic regression. We need to squeeze the weighted sum found by the neural network into a value between 0 and 1. The sigmoid function solves this problem.

$$\frac{1}{1+e^{-x}} \tag{6}$$

For $x \to 0$ it returns $1/2$, for $x \to \infty$ it returns 1 and for $x \to -\infty$ returns 0.

Nevertheless, there are many classification problems that aren't binary. An example is predicting handwritten digits in the MNIST dataset. You have numbers between 0 and 9, and each neuron in the output layer represent one of these numbers. In this case you want the neuron corresponding to the number you want to predict, to be most active. While all other neurons are as close to 0 as possible.
The activation function best suited for this problem is the softmax function. It will expand the sigmoid function used for binary classification, to allow multiple classification:

$$\frac{e^{z_i}}{\sum_{j=0}^{n} e^{z_j}}. \tag{7}$$

It takes the weighted sum $w^{l+1}a^l + b^{l+1}$, for each output neuron and squeezes the values between 0 and 1. The function differs from the sigmoid, because it divides each $z_i$ with the sum over all $z_k$ return values that sum to 1, like a probability density function.

### 4. *Cross Entropy*

When making classification neural networks, it doesn't make sense to find mean squared error. Therefore we introduce the cross entropy loss function. The definition is:

$$-\ln \prod^{C} P(y_c = 1)^{y_c}.$$

Where we find the probability $P$ of the prediction vector $y_{i,c}$ to be equal to 1, raised to the power of the prediction vector and multiplied over number of classifications. The assumption in this expression is the use of one-hot encoding. We'll use the handwritten digit classification problem introduced in section III 2. The one-hot vector prediction vector for the digit 2 will look like this:

$$y_2 = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0].$$

From the output neurons of the network we get a 10 element long array with values between 0 and 1. If the network predicted the right number, then activation in element three should be biggest. All the values are elementwise raised to the power of the one hot vector, and each value but the prediction is put to 1. Then you take the natural log of the result, and all values that are 1, becomes 0.
Cross entropy is visualized in figure 3. Where we see that the right prediction gives 0 loss, while the wrong prediction of 0 gives $\infty$. This way the cross entropy punishes wrong predictions, more than the mean squared error.

Remembering the explanation of the softmax function from section II 3, we understood the function to be a probability density. With this interpretation we understand that the cross entropy cost function finds the difference between our predicted probability function $y_c$ and the correct probability function represented by the one-hot vector. We need the derivative of the Cross Entropy, to calculate the gradients. However, it is not as straight forward as with mean squared error. We will not do the derivation, but will use the result obtained in [6]:

$$\frac{\partial C}{\partial p_c} = p_c - y_c. \tag{8}$$

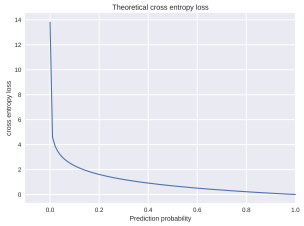


Figure 3. Visualization of the cross entropy loss based on output prediction. If the prediction is correct, and the neural network has predicted $\approx 1$ for the right category, then the loss is 0.

Where $p_c$ is the probabilities obtained by the neural network, and $y_c$ is the one-hot vector.

## III. METHOD

### 1. Stochastic Gradient Descent

We started off by expanding the analysis performed in our paper on regression [1]. The idea was to employ the gradient descent method outlined section II 1. However, gradient descent is subject to weaknesses we want to avoid. One drawback of this method is that we have to calculate the gradient over very many data points. This is very slow if we have many features in our data.

To speed up calculations, we employed stochastic gradient descent instead. We split the training data into random subsets called called mini-batches. The gradient was calculated for a mini-batch and the weights were updated, in essence gradient descent on the mini-batch. Then we picked another mini-batch repeating the same procedure. When the mini-batches were exhausted, we had completed a so-called epoch. Then we assigned new mini-batches at random and iterated another epoch. Epochs were iterated until the cost function stagnated, an indication that we had reached a minimum.

Picking up the thread from our paper on regression [1], we generated a data set from the Franke function which we sought to model by minimising the cost function.We split the data in train and test and performed stochastic gradient descent. Different values for learning rate, momentum, and penalty parameter were explored. These parameters are referred to as hyperparameters. We also explored how different mini-batch sizes and number of epochs affected the cost function. An ordinary least squares analysis was performed on the same data set as a benchmark to compare with SGD.

The measures we used to verify our models were mean squared error and the R2 score function. We calculated these quantities on the test data.

### 2. Neural network

Moving on, we want to analyse discrete data set. That is data that can only assume certain values. For this purpose, we wanted a method capable of complex decision-making. Therefore we created a feed forward neural network (FFNN), taking a flexible number of hidden layers and neurons.

We made a class in python, `DenseLayer` creating the layers in the neural network as objects. This class assigns number of inputs and outputs to a layer, and the activation function associated with this layer. The inputs corresponds to number of neurons in the previous layer, and outputs corresponds to number of neurons in this layer. The first layer is the first hidden layer, with inputs as the number of features in the input data. Every layer is initialised with random weights and biases following a normal distribution centered at zero. The `__call__` method of this class calculates the weighted sum, activations and the derivative of the activation function for a layer.

Our neural network is implemented in python as the class `NN` which is short for Neural Network. It takes a list of layers as input, where each layer in the list is an instance of the class `DenseLayer`. The network works through the two methods `feedforward` and `backprop`. The `feedforward` method updates weighted sums, activations and derivative of the activation functions by looping through the `__call__` method of `DenseLayer`. The `backprop` method executes the back-propagation algorithm explained in section II 1.

The cost function we used to train the network the following cases is mean squared error (MSE).

First, we tested the neural network on the data set generated by the Franke function. The input layer had two nodes corresponding to the coordinated $x$ and $y$ of the Franke function. The output layer had one node corresponding to $z = f(x, y)$. Creating different instances of the neural network with a different number of layers and neurons, we could observe how the network performed compared to stochastic gradient descent and ordinary least squared analysis.

Again we took advantage of the stochastic gradient descent method, as we split the training data into random mini-batches. The mini-batches were then run through a feedforward and a back-propagation until all mini-batches were exhausted. The network was trained over a number of epochs, until the accuracy was deemed good enough.

We tested different values for the hyper-parameters to see how they influenced the network.

Secondly, we analysed a discrete set. The data set we studied was handwritten digits from the MNIST data base [7]. The data set is provided in the scikit learn python library scikit learn [8]. The input layer had 64 nodes corresponding to the 64 pixels in each picture of a handwritten digit. We split the images in train and test data. Because this is a classification problem, we created

one-hot vectors corresponding to the labels of each image. The one-hot vectors consist of ten elements (one for each digit) with zeros everywhere except for the index of the digit we are looking at. Here we insert a one, hence the one-hot name. Then we trained the neural network with the pixels as input against and compared the output against the one-hot vectors.

We used the sigmoid activation function in the hidden layers, and the softmax function as activation in the output layer. The performance measure we used on the network was accuracy, i.e. the number of correct classifications the network outputted. We tested both MSE and cross-entropy as cost functions in the network.

As a sanity test of our network, we compared it with the open source library TensorFlow [9]. We created identical neural networks using API provided by Keras [10], and performed similar analyses on the data sets.

### 3. *Logistic regression*

Lastly, we wanted to compare a different method with the classification of handwritten digits from the neural network. The natural choice was to compare with logistic regression. We could implement logistic regression by using the already existing neural network. In this case the network is set up with only two layers, input and output.

The characteristic trait of logistic regression is calculating probabilities. Therefore we used cross-log entropy as the cost function. The activation function we used was the softmax function in the output layer. We created a special back-propagation method which has no hidden layers. The method is called `backprop2layer`, and is a member of class `NN`.

## IV. RESULTS

We performed OLS and SGD regression on the Franke function data set. The data set contained $10^4$ data points with added noise from a normal distribution centered around zero with standard deviation $\sigma = 0.1$. From various trials with different momentum terms $\gamma$, we found that $\gamma > 0.95$ gave unstable results. Choosing a too small momentum term did not improve the stochastic gradient descent very much. We selected $\gamma = 0.9$ for all later calculations.

First, we performed regression without a penalty parameter and with a constant learning rate $\eta = 10^{-3}$. We analysed SGD with mini-batch sizes of 30 after 200 epochs. The 2D polynomial we fitted to the Franke function was of degree 10. In this case, stochastic gradient descent yielded an MSE of $9.520 \cdot 10^{-3}$ and an R2 score of 0.8934. Ordinary least squares yielded an MSE of $2.589 \cdot 10^{-3}$ and an R2 score of 0.9710.

Then we experimented with different mini-batch sizes and epochs, still with a 2D polynomial of degree 10.

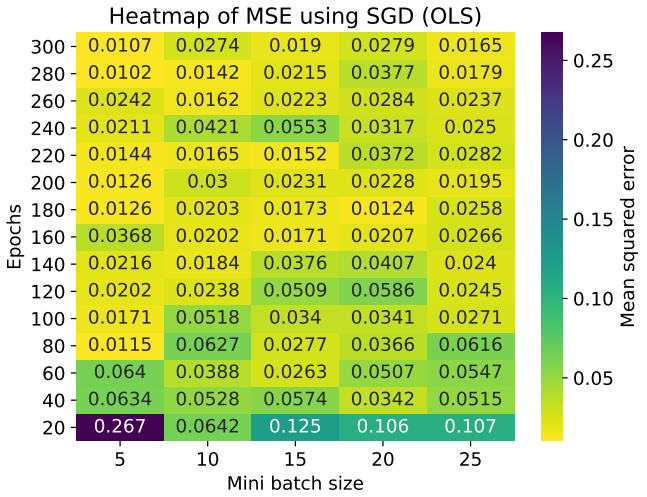


Figure 4. heatmap of the calculated mean squared error of the Ordinary Least Squared method with implemented Stochastic Gradient Descent algorithm. Learning rate was $\eta = 10^{-3}$.

A heatmap was made of the MSE for stochastic gradient descent with respect to the number of epochs and the size of the mini batches. We introduced the hyperparameter $\lambda$ which makes SGD analogous to Ridge regression. The penalty parameters were chosen from $\lambda \in [0.1, 0.01, 0.001]$. The Ridge method that produced the best results in terms of MSE is shown in figure 5. For SGD without a penalty parameter ($\lambda = 0$), results are shown in figure 4.

By comparing the two figures, we see that we get a better model of the Franke function without introducing a penalty parameter. This is consistent with the results in our paper on regression [1]. The best MSE value we found without a penalty parameter was 0.010213. This value was found after 280 epochs with a mini-batch size of 5. From this heatmap 4 we can see that mini-batch size is not crucial for good results, the determining factor is having a large enough number of epochs.

With $\lambda = 0.1$, the best MSE we found was 0.021808. This value was found after 80 epochs with a mini-batch size of 25. In the heatmap with penalty parameter 5 we see that nearly every value is very similar. It seems that mini-batch size does not matter very much as long as we perform over 20 epochs.

We also tried larger mini-batch sizes, which did not yield better results. However, larger mini-batch sizes reduced computation time.

We performed similar analyses to estimate the optimal learning schedule. We produced a heatmap as function of $t_0$ and $t_1$. We have used the optimized mini batch size of 5 from the previous heatmap to determine the best learning schedule parameters. We performed 200 epochs to obtain our results. Again, we will only show the SGD heatmap with the hyperparameter $\lambda$ that produced lowest MSE values. The Ridge method that produced the best results in terms of MSE is shown in figure 7.

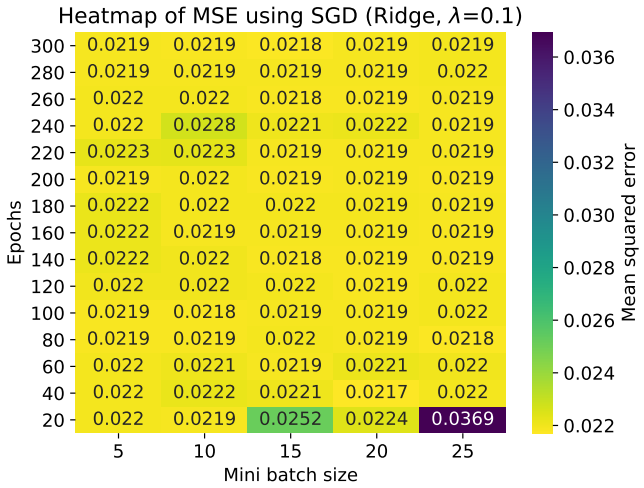


Figure 5. Heatmap of the calculated mean squared error of the Ridge method with implemented Stochastic Gradient Descent algorithm. Note that $\lambda = 0.1$.

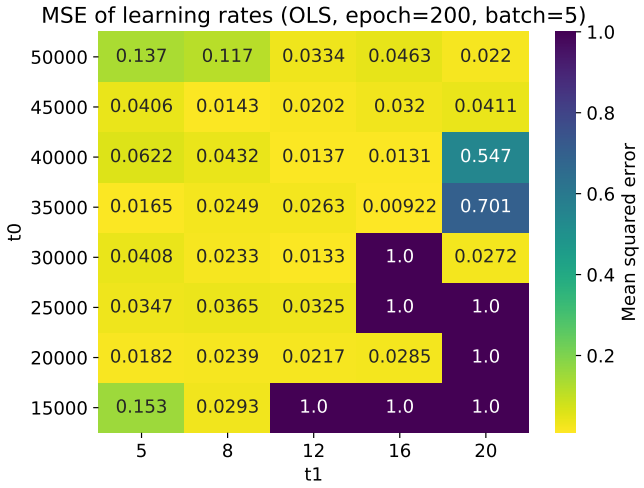


Figure 6. heatmap of the mean squared error by OLS method with implemented Stochastic Gradient Descent algorithm. Note that this heatmap has optimized parameters in terms of number of epochs and mini batch size from previous results. Note also that all MSE that are higher than 1 is set to equal 1.

For SGD without a penalty parameter ($\lambda = 0$), results are shown in figure 6.

Without a penalty parameter, we can see from figure 6 that the best MSE score is $9.215 \cdot 10^{-3}$ which is obtained for $t0 = 35000$ and $t1 = 16$. The MSE with mini-batch size of 30 after 200 epochs with a constant learning rate of $\eta = 10^{-3}$ was $9.520 \cdot 10^{-3}$. The best learning schedule yielded a result which was only $\sim 3\%$ better than a constant learning rate. We also observe that figure 6 does not in general give good MSE values.

With a penalty parameter of $\lambda = 0.1$, we observe the best MSE value to be 0.02185 from figure 7. This is not

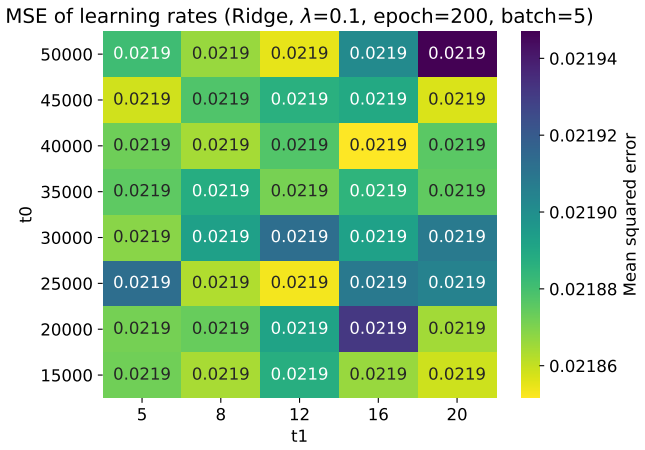


Figure 7. heatmap of the calculated mean squared error by Ridge method with implemented Stochastic Gradient Descent algorithm. Note that this heatmap has optimized parameters in terms of number of epochs and mini batch size from previous results. Note also that all MSE that are higher than 1 is set to equal 1.

better than the best MSE using a constant learning rate with $\lambda = 0.1$, which was marginally lower at 0.021808.

Next, we used our neural network to analyse the Franke data set. The network consisted of two hidden layers with 10 and 6 nodes, both with sigmoid as activation function and a constant learning rate. No activation was used in the output layer (i.e. $\hat{a}^L = \hat{z}^L$).

We trained the neural network for different constant learning rates and recorded MSE on the test data after different number of epochs. These results produced the heatmap shown in figure 8. From the figure we see MSE generally improving with a higher number of epochs. The same trend is true for the learning rate. The best network had a constant learning rate $\eta = 1.0$, with an MSE of 0.0109 after 1000 epochs. However, after only 50 epochs the network with $\eta = 1$ produced good results with an MSE of 0.0162.

Throughout the previous analysis, we did not introduce a penalty parameter in our network. Introducing a penalty parameter, we repeated a similar analysis. We trained the same network on the Franke data set for 100 epochs while varying learning rate and penalty parameter $\lambda$. The results from this analysis can be seen in figure 9. The penalty parameter yielding best results in this figure was the lowest, $\lambda = 0.01$. The lowest MSE was 0.013, with $\eta = 0.5$. In general, MSE values from the same learning rate after 100 epochs is not better with a penalty parameter compared with figure 8 where $\lambda = 0$. Therefore a penalty parameter is not beneficial for training the network on the Franke data set.

We compared our network with a similar network from the Keras library. We initialized a network with two hidden layers both taking sigmoid as activation function.

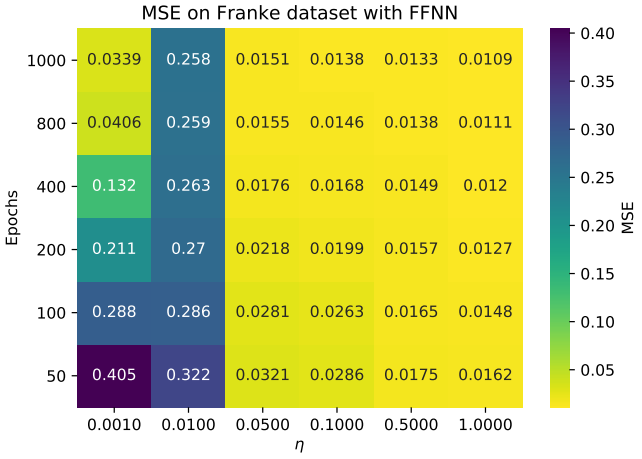


Figure 8. Mean square error as a function of learning rate $\eta$ and epochs. The values were found using our own Neural Network with Stochastic Gradient Descent with a mini-batch size of 100.

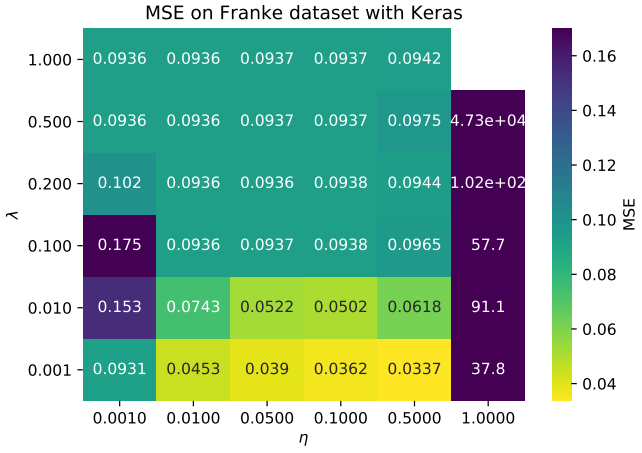


Figure 10. Mean square error as a function of learning rate $\eta$ and regularization parameter $\lambda$. The values was found using Keras [10]. We trained the network for 100 epochs using a mini-batch size of 100.

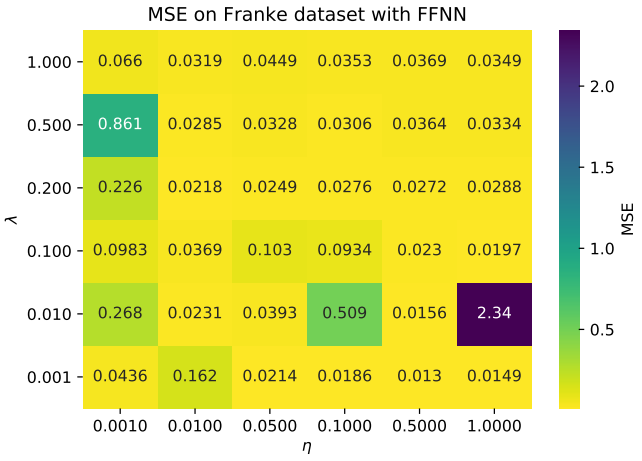


Figure 9. Mean square error as a function of learning rate $\eta$ and regularization parameter $\lambda$. The values were found using our own Neural Network with Stochastic Gradient Descent. We trained the network for 100 epochs using a mini-batch size of 100.

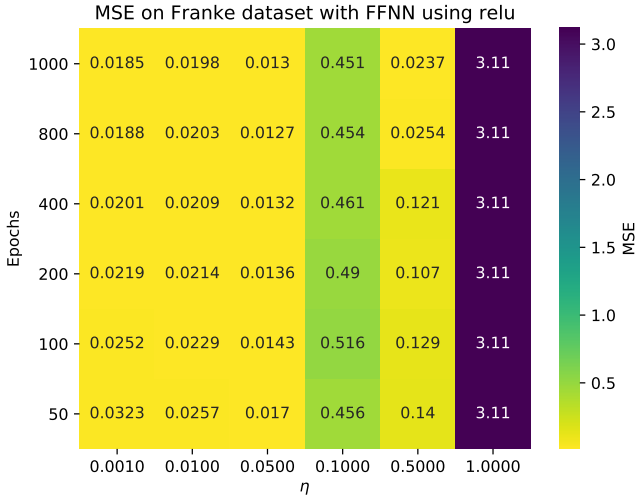


Figure 11. Same as figure 8, but with ReLU as activation function in the hidden layers.

Varying learning rate and penalty parameters the same way as above, we obtained the results presented in the heatmap in figure 10. Again, we trained the network for 100 epochs. The lowest MSE produced from Keras' network was 0.0337. In general, this network performed worse than our own FFNN. In this case, a learning rate of $\eta = 1$ made the network unstable and resulted in overflow when the penalty parameter was 1.

To extend our analysis, we used a different activation function in the hidden layers. Using the same setup of neural network, we found the rectifier activation function (ReLU) to be suitable for our purposes. Varying learning rate and penalty parameter, we obtained the results presented in the heatmap in figure 11. In this heatmap we observe that training for more epochs gives a better model in terms of MSE. Different from figure 9, a learning rate of 1 gives a network which predicts test data bad and does not improve with more training. Here the best MSE value is 0.013, which was obtained after 1000 epochs with $\eta = 0.05$. With ReLU as activation function, the network yields good results quickly. We observe that we get a low MSE (0.017) after only 50 epochs with $\eta = 0.05$.

Lastly, we experimented with different methods of initialising weights and biases. The layers of the network were initialised with weights and biases from a uniform distribution over $[0, 1)$. Varying learning rate and penalty parameter, we obtained the results presented in the heatmap in figure 12. From this figure, we see the same trend from figure 9 and 11 that an increasing num-

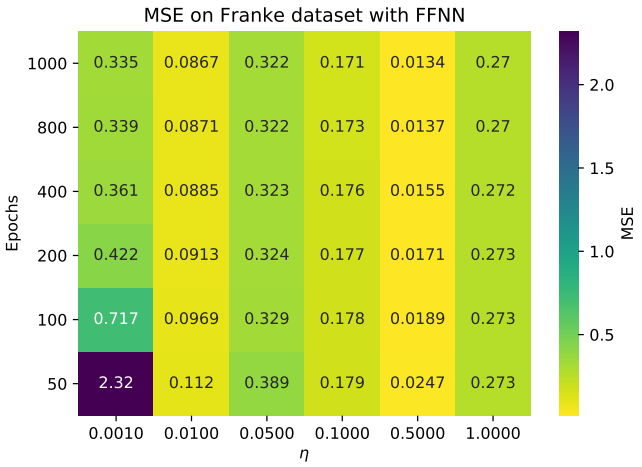


Figure 12. Same as figure 8, but with weights and biases initialised with random samples from a uniform distribution over $[0, 1)$.

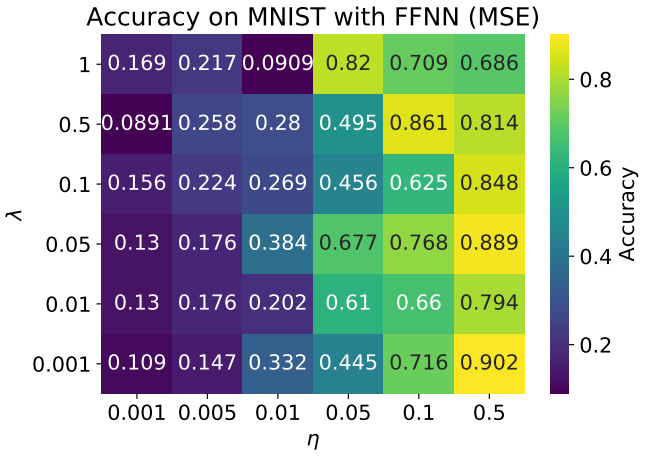


Figure 13. Accuracy of predictions on test data from MNIST dataset, with a network consisting of three hidden layers with 25, 100, and 50 neurons. MSE was used as cost function.

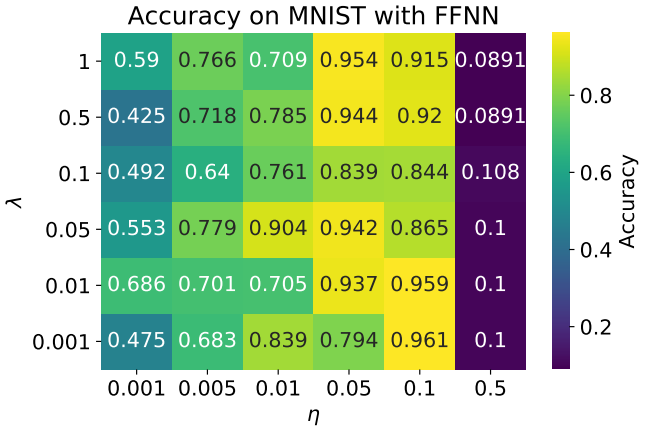


Figure 14. Accuracy of predictions on test data from MNIST dataset, with a network consisting of three hidden layers with 25, 100, and 50 neurons. Cross-entropy was used as cost function.

ber of epochs lowers the MSE. However, the dependency on learning rate $\eta$ seems more random. We also get worse MSE scores in general, and the network needs more epochs to reach good MSE values.

Next we trained the neural network to recognise handwritten digits. 70 % of the total data set were used to train the network, while the rest went to testing. After some trial and error, it was found that a mini batch size of 100 yielded good accuracy on recognition of both train and test data. The first network is build up of three hidden layers with 25, 100, and 50 neurons respectively. We trained our network over 200 epochs. For each completion of the epochs we chose a constant learning rate, and penalty parameter. For digit recognition we found that a penalty parameter greatly improved accuracy.

First, we used MSE as the cost function in the network. A heatmap of accuracy on test data from this network is shown in figure 13. The best accuracy obtained on the test data was 90.2% from a learning rate of 0.5 and a regularization parameter of 0.001.

Moving on, we changed the cost function to cross-entropy. A heatmap from this analysis is shown figure 14. The best accuracy obtained on the test data was 96.1% which is better than when we used MSE as cost function. We see that a too large learning rate makes the network unable to predict digits other than pure guessing with accuracy of approximately 1/10. A too small learning rate makes the network learn too slow.

We analysed a different network using the same values for learning rate and penalty parameters. Now we set up a network with just one hidden layer with 100 neurons and sigmoid as activation function. A heatmap of MSE as a function of learning rate and penalty parameter is given in figure 15. The best accuracy yielded from this network is slightly better than the previous case at 98.1 % accuracy on the test data. With this network, a higher penalty parameter was favorable.

Training the same network again, we produced a confusion matrix of the predictions on the test data. This time we got an accuracy of 97.2%, which is not identical to the previous accuracy because the weights were initialised a little different this time. The confusion matrix is presented in figure 16. We see that the network predicts the digits with varying accuracy. The number 7 is the network's weak point, as it is "only" predicted with 93% accuracy.

The last thing we did was to develop a logistic regression code to compare with our neural network. Logistic regression is shown in figure 17 as a function of learning rate and regularization. In this case we have used 200 epochs with a mini batch size of 100. The accuracy reached a saturation point, and we therefore chose to use a constant number of epochs. The best accuracy

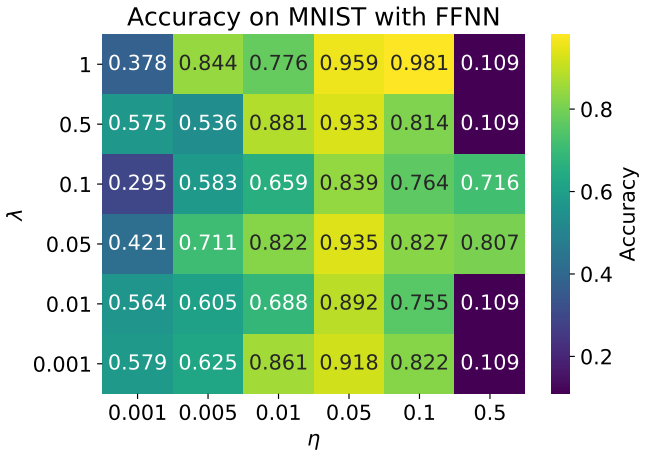


Figure 15. Same as figure 14, with network consisting of one hidden layer with 100 neurons.

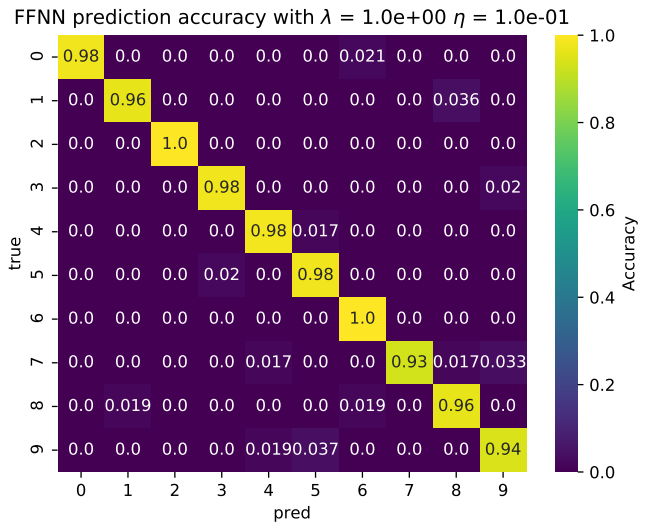


Figure 16. Confusion matrix on MNIST test data produced by a network consisting of one hidden layer with 100 neurons. Cost function was cross-entropy.

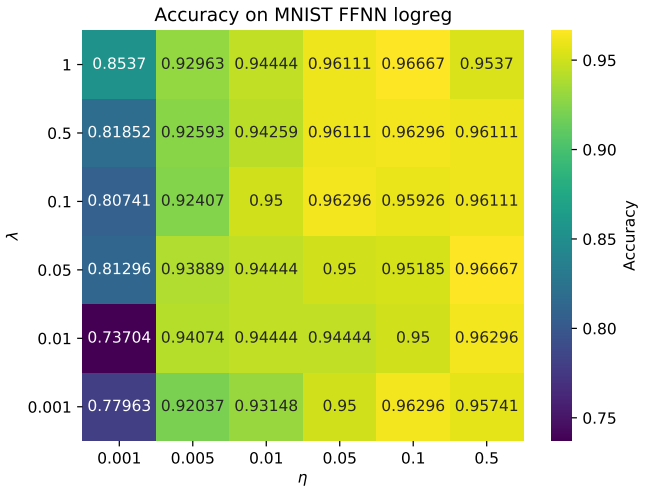


Figure 17. The accuracy of the model recognising handwritten digits from the MNIST database. The model is trained by the logistic regression method with SGD. We used varying learning rates $\eta$ and regularisation parameters $\lambda$.

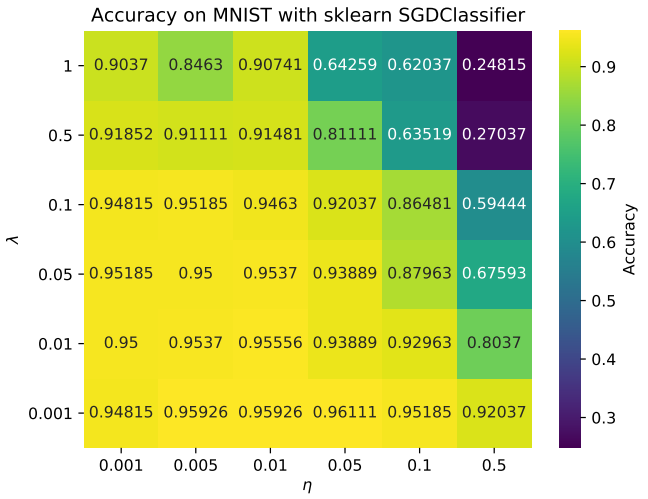


Figure 18. The accuracy of the model recognising hand written digits from the MNIST database. The model is trained by the logistic regression method with SGD, using scititlearn's SGDClassifier.

we found was $\approx$ 97%, with 0.1 learning rate. In figure 18 we found the same heatmap, using scikit learn `SGDClassifier` to calculate accuracy on MNIST digits. The best accuracy was $\approx$ 96%. Comparing them, we see that they are within the same range of $96 - 97\%$ accuracy, but our network used $\approx$ 23 s computation time, while scikitlearn used $\approx$ 2 s.

Lastly we created a confusion matrix that is another way of showing the performance of our network. The result can be seen in 19. High values on the diagonal, indicates many right predictions.

## V. DISCUSSION

First off we created our own Stochastic Gradient Descent algorithm which produced results as shown in figure 4 and 5. By studying these plots, it is overall clear that MSE decreases with increasing epochs. However, the MSE reaches a saturation point around 300 epochs. When we calculate the gradient on the Franke function we only have two partial derivatives ($x$ and $y$) and it is not very expensive to calculate the gradient. Therefore it is not necessary to choose very small mini-batches for efficiency. In fact, too small mini-batch sizes in this case

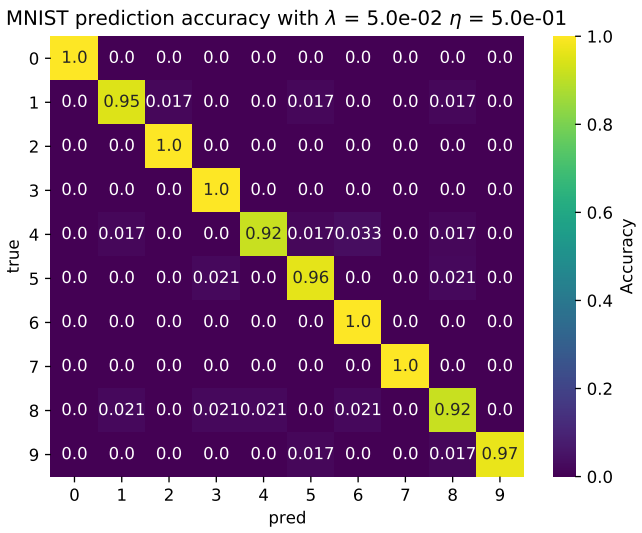


Figure 19. The confusion matrix showing accuracy of the neural network on predicting MNIST digits using logistic regression. High values on the diagonal, indicates that labels correctly, and doesn't confuse e.g. an 8 as a 9.

will reduce efficiency because we have to loop over very many mini-batches. Therefore it will be advantageous to choose larger mini-batch sizes than 5.

MSE values vary more without a penalty parameter as seen in figure 4. Generally the MSE values were more stable with the Ridge equivalent of SGD from figure 5. Because of this, the OLS equivalent SGD sometimes produce lower MSEs, and sometimes higher MSEs. If we compare the SGD results we obtain with OLS, we note that the results of MSE is not as good. It is in fact a factor of $\sim 4$ greater in this project as we have used the SGD method. The SGD method converges to a solution of optimal parameters by a stochastic approximation relatively fast but the drawback is that it is not as accurate when trying to lower the MSE to a minimum.

After extracting the satisfied values of the number of epochs as well as mini batch size, we made another heatmap of the learning rates as function of the parameters $t_0$ and $t_1$ as shown in figure 6 and 7. It is clear that the OLS method are more sensible to the learning schedule than the Ridge method when analysing the MSE. Ridge method tends to stabilise with MSE at approximately 0.02 while OLS deviates a lot relatively speaking.

We did not find a learning schedule that improved stochastic gradient descent compared to a constant learning rate of $\eta = 10^{-3}$.

For our neural network we chose the mean squared error as our cost function when analysing the Franke data set. It represents, in an intuitive way, the difference in predicted values $\tilde{y}$ from the actual values $y$. The cost function is essential when finding gradients, and is easier to implement if it has a simplistic derivative.

The best obtained value for MSE using Stochastic Gradient Descent with our neural network was $\approx 0.0109$ (figure 9). With Keras we got $\approx 0.03$.

Compared with OLS MSE of $2.589 \cdot 10^{-3}$ it falls short. However, keep in mind that OLS has an analytical solution to the equation:

$$\frac{\partial \tilde{y}}{\partial \beta_j} = \boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}).$$

While gradient descent is an iterative scheme that is not guaranteed to minimise the cost function. Another point is that the training data we used in our regression paper [1], wasn't split into mini-batches. This introduces a randomness to the network, that combined with randomly initialised weights, will have an affect on the result.

We found that initialising the weights and biases following a normal distribution worked well when using the Sigmoid activation function in the hidden layer. With weights and biases initialised uniformly (see figure 12), the network performed worse. It was clear that the deviations in performance was caused by the randomness of the network, as the minimum MSE values were sporadically placed in the heatmap.

When we tried ReLU as activation function with normally distributed weights and biases, we got a fast convergence, but the MSE did not improve. A problem often occuring is dying ReLU's, but running the network with Leaky ReLU did not improve the results any further.

A classification problem were added to the test to the neural network, using mean squared error and cross-entropy as cost functions. From the case where we used MSE as cost function, the overall accuracy was poor. Almost none of the cases exceeded the 90 % accuracy mark so that the neural network could probably not be used in any practical forms. MSE is not a good idea in multinomial classification. The reason being that MSE does not differentiate as hard between a wrong and a right prediction. If the right prediction was the one hot vector $[0, 1, 0]$, and the network suggests $[0.5, 0.4, 0.3]$, then MSE will be $[0.5^2, 0.4^2, 0.3^2]$. We know that the prediction was wrong, but the MSE error does not reflect how wrong the answer was. The network will therefore continue by slowly changing weights and biases to improve the result, and will most likely not reach the minimum.

With this argument, it makes sense that our best result was for the highest learning rate 0.5. Because it helps to increase the pace of which the network moves toward the minimum when the gradient becomes too small.

When exchanging MSE and introducing Cross entropy in the three layer network we get an improved result for both three and one hidden layer network.

In figure 15, we see that the simpler network with just one hidden layer performs better than the more complex network with three hidden layers. A likely culprit which is behind the poor performance of the larger networks is a vanishing gradient. This is known as the vanishing gradient problem. In deep networks, the gradient gets smaller as we move backwards in the hidden layers towards the input layer. This results in the early hidden layers learning very slowly.

Vanishing gradients results from the gradient of the activation function. The traditional activation function sigmoid which we used has gradients in the range (0, 1). Because back-propagation computes gradients by the chain rule, we multiply each layer with a number with a number smaller than one. Effectively this shrinks the gradient towards zero as we go backwards to the input layer.

The vanishing gradient problem might be solved by using different activation functions such as rectifiers (e.g. ReLU), as proposed by Glorot et. al. [11]. For the classification problem, we did not get sensible results by using ReLU as activation functions in the hidden layers. With our network, we experienced overflow. A possible solution to this problem, is using Glorot Initialisation [12] when initialising weights and biases in the layers of the network. We recommend exploring this feature in future projects. It would also be recommended to examine more complex data sets with deeper networks

Comparing the results for logistic regression obtained using logistic network in figure 17 with the results obtained in figure 15. We see that the neural network obtained a peak accuracy of $\approx 98\%$, while the logistical regression obtained $\approx 97\%$.
Cross entropy is used to penalise wrong predictions (see section II 4). This ensures that we don't get the vanishing gradients, which was a challenge when using the sigmoid function with mean squared error. The reason is that the $\sigma'(z)$ (where $\sigma$ is softmax) term vanishes from the $\partial C/\partial w$ expression, as proven by Nielsen [4].

Looking at figure 17 and 18, we see that our logistical regression performed better than scikitlearn. However, taking into account the time aspect. We see that scikitlearn was almost one minute faster than our network. This has an crucial impact in scaling the network to allow larger datasets. An improvement could be to implement a parallel computing. There are large triple and quadruple for loops, that could benefit from splitting the loops between cores in a cpu.

The confusion matrix in figure 19 gives us above 90% accuracy for every digit, confirming the high accuracy. The lowest accuracies was given when recognising the digits $1, 4$ and 8. There is not a clear pattern as to why these digits were harder to predict. 4 was miscategorised as an 8, and vice versa. While 1 was most often miscategorised as $2, 5$ and 8, when they from a visual standpoint look different. However, the differences in accuracy across the numbers might be caused by the randomness of the initialised weights.

## VI. F

We see that both the neural network and logistic regression with cross entropy performs very well on the mnist data. The neural network obtained a peak result of 98%, while logistic regression obtained 97%. Therefore a more complex dataset, will benefit from having a deeper neural network. However, even though we did not find any overfitting. This might be a problem when training another dataset, with less complexity. For future projects, it would be beneficial to run several datasets trough our network, to increase it's flexibility.

## VII. CONCLUSION

In this paper we have examined Stochastic Gradient Descent (SGD), Feed Forward Neural networks and Logistic Regression. Our work has been continously compared with scikitlearn and Keras to confirm that our work is correct. We implemented Stochastic Gradient Descent to fit a 2D polynomial to the Franke function. By splitting training data into reasonably sized mini batches, and performed Gradient Descent, we found a minimum MSE of $9.520 \cdots 10^{-3}$, while the OLS regression solver we made in our paper [1] obtained a minimum value of $2.589 \cdots 10^{-3}$ which is almost 4 times better than SGD. The reason being the exact analytical solution of OLS. We chose to initialise the weights randomly, and aquired acceptable results. However, for future papers it might be interesting to examine different weight initialising techniques.

We introduced the Feed Forward Neural Network with Back-propagation MSE as cost function and varying activation functions to predict both Franke function and the MNIST handwritten digit dataset.
We saw that MSE saturated at a satisfactory value around 200 epochs and a mini-batch size of 100. Also using sigmoid as our activation function proved to be satisfactory. We tested different learning rates $\eta$ and regularisation parameters $\lambda$, and the best MSE obtained from predicting Franke Function values was 0.0109.

For the MNIST dataset we used Cross Entropy as a cost function. It is better suited for multinomial classification. The best accuracy achieved predicting handwritten digits was $\approx 98\%$. The confusion matrix in figure 16 confirms the accuracy obtained in the heatmap in figure 15.
Lastly we introduced logistic regression with Cross Entropy as cost function, and Softmax as activation function. The highest accuracy was $\approx 97\%$. The confusion matrix in figure 19 confirms the good performance. For later papers it would be of high interest to test our network on other datasets, and see how well it performs. Preferably a dataset with more features, and more inputs. We are barely touching the surface of what Machine Learning has to offer, and the opportunities are endless.

[1] E. R. Udnaes, J. Lie, and J. T. Faber, "Project 1 - fys-stk4155," 2020.
[2] E. Kamperis, "A blog on science." Accessed: 2020-11-05.
[3] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, "A high-bias, low-variance introduction to machine learning for physicists," *Physics Reports*, vol. 810, p. 1–124, May 2019.
[4] M. A. Nielsen, "Neural networks and deep learning," 2018.
[5] databricks, "Neural network." Accessed: 2020-11-05.
[6] P. Dahal, "Classification and loss evaluation softmax and cross entropy loss." https://deepnotes.io/softmax-crossentropy, 2018.
[7] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.
[8] 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.
[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.
[10] F. Chollet *et al.*, "Keras." https://github.com/fchollet/keras, 2015.
[11] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," vol. 15 of *Proceedings of Machine Learning Research*, (Fort Lauderdale, FL, USA), pp. 315–323, JMLR Workshop and Conference Proceedings, 11–13 Apr 2011.
[12] J. D. McCaffrey, "Neural network glorot initialization." https://jamesmccaffrey.wordpress.com/2017/06/21/neural-network-glorot-initialization/, 2017.

## VIII. SOURCE CODE

Source code and additional results can be found in the github repository https://github.com/jacobllie/FYS-STK4155.