UiO : Department of Physics
University of Oslo

FYS-STK4155: Applied Data Analysis and Machine Learning

Project 2

Classification and Regression: From Linear and Logistic Regression to Neural Networks

Oda Hovet Ilse Kuperus Erik Alexander Sandvik

November 20, 2020

Abstract

This paper is divided into three main parts. First we assess the performance of OLS and Ridge regression. Although these are well established traditional linear regression methods with analytical solutions, the catch this time is that we use gradient descent methods to minimize the cost function numerically. We use this for making linear model of the Franke function. Second, we use logistic regression on the classification problem of assigning the correct digits to images of handwritten digits. Third and finally, we implement a neural network for both regression and classification on the same problems and see how it compares to the traditional linear regression methods and logistic regression. We were able to achieve a respectably low mean squared error with the neural network in the regression problem, although it does not perform as well as OLS and Ridge regression. Similarly, we were able to implement a neural network that correctly classifies $\sim 95\%$ of the digits in the test data, but it still lags behind logistic regression where we achieved an accuracy of $\sim 96\%$.

Link to GitHub github.com/erikasan/fys-stk-project2

Contents

1 Introduction **1**

2 Theory **1**
2.1 Linear Regression (revisited) . . . 1
2.2 Logistic Regression . . . 2
2.2.1 The Softmax Function . . . 2
2.3 Optimization and Gradient Descent . . . 2
2.3.1 Batch Gradient Descent . . . 3
2.3.2 Stochastic Gradient Descent . . . 3
2.3.3 Mini-Batch Gradient Descent . . . 4
2.4 Neural Networks . . . 4
2.4.1 The Backpropagation algorithm . . . 6

3 Method **10**
3.1 Linear Regression - Gradient Descent . . . 10
3.2 Logistic Regression . . . 10
3.3 Neural Network . . . 11

4 Results and Discussion **13**
4.1 Gradient Descent and Hyperparameters . . . 13
4.2 Logistic Regression . . . 18
4.3 Neural Networks . . . 22
4.3.1 Regression with Neural Networks . . . 22
4.3.2 Classification with Neural Networks . . . 25

5 Conclusion **29**

References **30**

1 Introduction

Neural networks and logistic regression are two very useful techniques in machine learning to classify data. Both these methods can be used in cases where data, like for example images, need to be classified into groups. By using neural networks or logistic regression for this we can easily classify a lot of data in a short time. In this project we will look at both neural networks and logistic regression in a few simple examples. When using the logistic regression we will test the method by classifying handwritten digits, this is a multi class logistic regression. We also explore its application for linear regression problem for the Franke function. We investigate the importance of related parameters such as learning rate, regularization parameter, choice of activation functions, number of hidden layers and number of nodes in neural networks.

The inputs of the neural network are combined by weighting parameters to produce an output. In order to predict the output, the network must be trained. The network is trained by optimizing the weights, and armed with insights from investigating optimization methods, we use mini-batch gradient descent update rule for this. We try to find combinations of hyperparameters that give better prediction in the flexible neural network. We test our network on the classification problem for hand-written digits 0-9 from the MNIST data set. Finally, we compare and summarize the performance of the methods in terms of accuracy, cost and finding reasonable parameters.

This project is structured by first presenting a theoretical overview of optimization methods, logistic regression and neural networks in Section 2. In Section 3, we provide an approach to study linear-regression and logistic regression problems. This is followed by results and discussion of the implementation in Section 4. Lastly, we provide a conclusion and suggestions for future work in Section 5.

2 Theory

2.1 Linear Regression (revisited)

Linear regression models predict data for a continuous dependent variable by learning the coefficients of a functional fit. The prediction is done by computing a weighted sum of the input features. Given a set of p inputs $\mathbf{x}^T = (x_0, x_1, \ldots, x_{p-1})$ and a corresponding output y, we make a prediction $\tilde{y}$ of y by a linear model

$$\tilde{y} = \sum_{j=0}^{p-1} x_j \beta_j$$

where $\beta_j \in \mathbb{R}$ are unknown coefficients. The three regression methods Ordinary Least Squares (OLS), Ridge and Lasso were used to find the optimal coefficient vector β that minimizes the cost function Mean Squares Error (MSE). For theory and a discussion of these methods, the reader is referred to read the study by the authors in [1].

2.2 Logistic Regression

Logistic regression is a statistical method that uses logistic functions to model variables. Logistic regression is commonly used for classification cases where it predicts a probability that an instance belongs in a certain class. In most cases logistic regression is used in binary cases, where a logistic function is used to model a binary dependent variable. In a binary case the model is predicting which of the two classes the variable belongs in based on the probability. If the probability for one of the instances being in class is over 50%, then the model will predict this being the class for the instance. Just like the linear regression case logistic regression computes the coefficients for β and uses these to find the prediction, however for logistic regression the results are not found directly from these coefficients but are rather found from using these coefficients in a logistic function. There are different functions that can be used. In a binary case the Sigmoid function is often used as this gives a probability that the instance belongs to the positive class. However logistic regression can also be used with more than two classes, in this case we can use a different logistic function. The one we will use for our exploration in this project is called the Softmax function.

2.2.1 The Softmax Function

The softmax function is the logistic function we will be using to predict the class the given instant belongs to. The softmax function is a generalization of the logistic function to multiple dimensions. The softmax function in given by,

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

where K is the number of classes and z is the input variable. This input variable that needs to be inserted into the softmax function to get the probability distribution is called the score. The score is found by using the coefficient matrix for the different classes θ and taking the dot product with the design matrix X, $s = \theta^T \dot{X}$. Thus we insert this into the softmax function to get the probability distribution $p_i = \sigma(s)_i$. This function will return a probability distribution of K probabilities which in total will sum to 1. Like the sigmoid function the softmax function predicts the instance to be in the class that has the highest probability.

2.3 Optimization and Gradient Descent

Gradient descent (GD) is an optimization algorithm. In machine learning, it is often used for minimizing a cost function by adjusting the parameters iteratively. Suppose we define a function $f(\boldsymbol{x})$ where $\boldsymbol{x} \equiv (x_1, ..., x_n)$, then $f(\boldsymbol{x})$ has the steepest decrease in the direction along $-\nabla f(\boldsymbol{x})$. This can be shown as:

$$\boldsymbol{x}_n = \boldsymbol{x}_{n-1} - \eta_{n-1} \nabla f(\boldsymbol{x}_{n-1})$$

where η is the step size or learning rate, $\eta_{n-1} > 0$. If η_n is sufficiently small, the sequence converges to a local minimum of the cost function. The sequence $\{\boldsymbol{x_n}\}_{n=0}$ would ideally converge to a global minimum of f. This is especially true for convex functions where all local minima are global minima. However, in machine learning, there can be several local minima, which can lead to poor performance [2].

One advantage of GD is that it is conceptually simple and easy to implement. However, some disadvantages are that it is sensitive to initial conditions and choice of learning rates. The learning rate hyperparameter determines the size of steps and should be chosen in order to match the landscape. If the learning rate is too small, the algorithm goes through too many iterations before converging, which can take a long time. On the other hand, a too large learning rate will make the algorithm diverge with larger values which will make it difficult to find an optimal solution. The learning rate is also treated equally and set by the steepest direction. In addition, GD is expensive for large datasets and even with random initialization, it can spend exponential time to escape saddle points. The optimal solution would be to have large steps at flat directions and smaller steps in steep directions [2] [3].

2.3.1 Batch Gradient Descent

Batch GD uses the whole batch of training data at each step. The gradient vector, $\nabla_\theta MSE(\theta)$, for the cost function of each model parameter, contains all partial derivatives for the the cost function.

$$\nabla_\theta MSE(\theta) = \begin{pmatrix} \frac{\partial}{\partial\theta_0} MSE(\theta) \\ \frac{\partial}{\partial\theta_1} MSE(\theta) \\ \vdots \\ \frac{\partial}{\partial\theta_n} MSE(\theta) \end{pmatrix} = \frac{2}{m}\boldsymbol{X}^T \cdot (\boldsymbol{X} \cdot \theta - \boldsymbol{y})$$

where $\boldsymbol{X}$ is a full training set for calculations at each GD step. Therefore, batch GD is slow for large training sets, but still scales the number of features well. For the GD step, subtract $\nabla_\theta MSE(\theta)$ from θ to descend:

$$\theta^{(\text{next step})} = \theta - \eta \nabla_\theta MSE(\theta)$$

where the learning rate η determines the step size.

2.3.2 Stochastic Gradient Descent

The concept of Stochastic Gradient Descent (SGD) is that the algorithm randomly selects data from the training set at every step. Subsequently, based on this randomly selected data, the gradients can be computed [3]. The gradients can be computed as a sum of i-gradients:

$$\nabla_\beta C(\beta) = \sum_i^n \nabla_\beta c_i(\boldsymbol{x}, \beta)$$

For Batch GD which is slow for large training sets, SGD is fast since it does not iterate through as much data. The stochasticity also means that the cost function bounces up and down and decreases over average. The irregularity of SGD can be an advantage for escaping local minima, but at the same time it never settles at a local minima. A solution to this is to gradually reduce the learning rate with simulated annealing [4] [3].

GD has a tendency to slow down when approaching minimum and the gradient becomes smaller. One way to remedy this is to add momentum. SGD with momentum is called momentum-based SGD with momentum parameter η. This saves the direction for moving the parameter space and can be implemented as:

$$\begin{aligned}\boldsymbol{v}_t &= \gamma \boldsymbol{v}_{t-1} + \eta_t \nabla_\theta J(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \boldsymbol{v}_t\end{aligned}$$

where $0 \leq \gamma \leq 1$ and is the momentum hyperparameter. Momentum-based SGD is advantageous since it uses the gradient descent algorithm as an acceleration. This means that the GD algorithm is faster in areas with flat landscape and slower in steeper landscapes [4] [3].

2.3.3 Mini-Batch Gradient Descent

In GD, we need to sum over all n data points. This makes gradients computationally expensive to calculate for large datasets. To manage this problem, one can compute the gradients on small random subsets of the data. These subsets are called mini-batches. The advantage of Mini-batch GD is that mini-batches introduce stochasticity and thereby reduce the risk of getting stuck in a local minima. In addition, the advantage of Mini-batch GD over SGD is the performance boost from hardware optimization of matrix operations [3].

2.4 Neural Networks

A neural network is composed of so-called ”neurons” or ”nodes” structured in layers. A single neuron takes one or more scalar inputs $x_1,\ x_2, \ldots$ and returns a single scalar output a called the activation of the neuron. Every neuron has a set of weights $w_1, w_2, \ldots$ for each input and a bias b. In addition, every neuron has a so-called activation function $\sigma(z)$ that takes a weighted sum of the inputs and returns the activation of the neuron. In mathematical notation the activation is given by

$$a = \sigma(z),\ z = w_1 x_1 + w_2 x_2 + \ldots + b,$$

which we may also write as $a = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$ where $\mathbf{w}$ and $\mathbf{x}$ are vectors containing the weights and inputs respectively. The choice of activation function is completely arbitrary, however among some (historically) popular choices is the Heaviside step function

$$H(\theta) = \begin{cases} 0 & \theta < 0 \\ 1 & \theta \geq 1 \end{cases},$$

in which case the neuron is referred to as a ”perceptron” returning a binary output corresponding to the neuron being ”on” or ”off”. However, a problem with the Heaviside step function is that it is not continuous, a feature required by the backpropagation algorithm which is to be explained in a later section. The Heaviside step function has therefore been (historically) replaced by the sigmoid function

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

which has the form of a continuous ”smoothed out” Heaviside step function. Other popular choices include the Rectified Linear Unit (ReLU) function

$$\mathrm{ReLU}(z) = \max(0,\ z),$$

and the modified ReLU function (also called the ’Leaky ReLU’)

$$\text{Leaky ReLU}(z) = \max(\alpha z,\ z),\ \alpha \in \langle 0,\ 1\rangle$$

Regarding the inputs of a neuron, they are the outputs of neurons from the previous layer. Meanwhile, the output of our neuron, along with the output of the other neurons in the same layer, are sent as inputs to the neurons in the next layer, and so on. Eventually, the final layer called the output layer is reached. The output of the output layer is regarded as the output of the neural network as a whole.

Now, obviously, the range of the activation function in the output layer is also the range of what possible outputs the neural network can produce as a whole. So the activation function of the output layer must be chosen carefully depending on the problem at hand. For example, when doing regression, you may not want to pick an activation function of limited range in the output layer. Meanwhile, if the output of the neural network is to be interpreted as probabilities, the output of a node in the output layer must be in the range [0, 1], and the outputs of all the nodes in the output layer must sum up to 1. This is why in the following, we will distinguish between the activation function $f(z)$ of the output layer, and the activation function $\sigma(z)$ of some hidden layer.

Again, each of these neurons has an activation function $\sigma(z)$, a bias b and a set of weights $w_1, w_2, \ldots$ where the number of weights is equal to the number of neurons in the previous layer. The exceptions are the neurons of the first layer (the input layer) where the outputs are equal to the inputs. To keep track, we denote a weight in the neural network by w^l_{jk}, a bias by b^l_j and an activation of a neuron by a^l_j. For example, w^l_{jk} means that it is the k-th weight of the j-th neuron in the l-th layer.

If we collect the weights of the l-th layer in a matrix $\mathbf{W}^l$, and the biases and the activations of the l-th layer into column vectors $\mathbf{b}^l$ and $\mathbf{a}^l$ respectively, we can write the output of the l-th layer as

$$\mathbf{a}^l = \sigma\left(\mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l\right)$$

where it is understood that the activation function is applied elementwise when the argument is a vector. Implementing this expression iteratively is known as the ”feed forward” algorithm.

2.4.1 The Backpropagation algorithm

The goal now is to choose the weights and the biases of the network in such a way that it emulates a desired output $\mathbf{y}$ from a corresponding input $\mathbf{x}$ as much as possible. This is measured by the choice of cost function which in order for backpropagation to work (as well as stochastic gradient descent), must be written on the form

$$C = \sum_{\mathbf{x}} C_{\mathbf{x}} \left(\mathbf{a}^L, \mathbf{y} \right),$$

i.e as a sum over the data set $\{\mathbf{x}, \mathbf{y}\}$. The superscript L denotes a parameter of the output layer. We use stochastic gradient descent to update the weights and the biases in such a way that the cost function is minimized. To do this we need to find the partial derivatives of $C_{\mathbf{x}}$ with respect to each of the weights and the biases, which is what the backpropagation algorithm will give us.

Since $C_{\mathbf{x}}$ is a direct function of the outputs a_i^L we start by finding the derivatives with respect to the weights w_{ij}^L of the output layer.

$$\frac{\partial C_{\mathbf{x}}}{\partial w_{ij}^L} = \frac{\partial C_{\mathbf{x}}}{\partial a_i^L} \frac{\partial a_i^L}{\partial z_i^L} \frac{\partial z_i^L}{\partial w_{ij}^L}$$

Here $\partial a_i^L / \partial z_i^L = f'\left(z_i^L\right)$ and $\partial z_i^L / \partial w_{ij}^L = a_j^{L-1}$. Thus

$$\frac{\partial C_{\mathbf{x}}}{\partial w_{ij}^L} = \frac{\partial C_{\mathbf{x}}}{\partial a_i^L} f'\left(z_i^L\right) a_j^{L-1}.$$

The first couple of factors will appear a lot, so we define

$$\delta_i^L \equiv \frac{\partial C_{\mathbf{x}}}{\partial a_i^L} f'\left(z_i^L\right)$$

to write

$$\frac{\partial C_{\mathbf{x}}}{\partial w_{ij}^L} = \delta_i^L a_j^{L-1}.$$

The corresponding expression in matrix-vector notation is

$$\frac{\partial C_{\mathbf{x}}}{\partial \mathbf{W}^L} = \boldsymbol{\delta}^L \left(\mathbf{a}^{L-1}\right)^T,$$

where $\partial C_{\mathbf{x}} / \partial \mathbf{W}^L$ is a matrix of the same shape as $\mathbf{W}^L$ and $\boldsymbol{\delta}^L$ is given by

$$\boldsymbol{\delta}^L \equiv \frac{\partial C}{\partial \mathbf{a}^L} \odot f'\left(\mathbf{z}^L\right),$$

where $\odot$ denotes the Hadamard product of two vectors (or matrices) of the same shape. Now we find the derivatives of $C_{\mathbf{x}}$ with respect to the biases b_i^L

$$\frac{\partial C_{\mathbf{x}}}{\partial b_i^L} = \frac{\partial C_{\mathbf{x}}}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_i^L}\frac{\partial z_i^L}{\partial b_i^L}.$$

Here $\partial a_i^L/\partial z_i^L = f'\left(z_i^L\right)$ and $\partial z_i^L/\partial b_i^L = 1$, so

$$\frac{\partial C_{\mathbf{x}}}{\partial b_i^L} = \frac{\partial C_{\mathbf{x}}}{\partial a_i^L} f'\left(z_i^L\right) = \delta_i^L.$$

The corresponding expression in matrix-vector notation is

$$\frac{\partial C_{\mathbf{x}}}{\partial \mathbf{b}^L} = \boldsymbol{\delta}^L.$$

Now even though $C_{\mathbf{x}}$ is a direct function of the activations a_i^L of the last layer, we could just as well (if we wanted to) write $C_{\mathbf{x}}$ as a direct function of the activities z_i^L of the last layer as well. The derivatives of $C_{\mathbf{x}}$ with respect to the activities of the last layer are

$$\frac{\partial C_{\mathbf{x}}}{\partial z_i^L} = \frac{\partial C_{\mathbf{x}}}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_i^L} = \frac{\partial C_{\mathbf{x}}}{\partial a_i^L} f'\left(z_i^L\right) = \delta_i^L.$$

We can use these to find an expression for the total derivative of C_k, which will help us to find the derivatives of C_k with respect to the weights and biases of the next-to-last layer.

$$dC_{\mathbf{x}} = \sum_i \frac{\partial C_{\mathbf{x}}}{\partial z_i^L} dz_i^L = \sum_i \delta_i^L dz_i^L$$

Now, since

$$\frac{\partial z_i^L}{\partial w_{jn}^{L-1}} = \frac{\partial z_i^L}{\partial a_j^{L-1}}\frac{\partial a_j^{L-1}}{\partial z_j^{L-1}}\frac{\partial z_j^{L-1}}{\partial w_{jn}^{L-1}} = w_{ij}^L \sigma'\left(z_j^{L-1}\right) a_n^{L-2},$$

and

$$\frac{\partial z_i^L}{\partial b_j^{L-1}} = \frac{\partial z_i^L}{\partial a_j^{L-1}}\frac{\partial a_j^{L-1}}{\partial z_j^{L-1}}\frac{\partial z_j^{L-1}}{\partial b_j^{L-1}} = w_{ij}^L \sigma'\left(z_j^{L-1}\right),$$

the derivatives of $C_{\mathbf{x}}$ with respect to the weights and biases of the next-to-last layer are

$$\frac{\partial C_{\mathbf{x}}}{\partial w_{jn}^{L-1}} = \sum_i \delta_i^L \frac{\partial z_i^L}{\partial w_{jn}^{L-1}} = \sum_i \delta_i^L w_{ij}^L \sigma' \left(z_j^{L-1} \right) a_n^{L-2}$$

and

$$\frac{\partial C_{\mathbf{x}}}{\partial b_j^{L-1}} = \sum_i \delta_i^L \frac{\partial z_i^L}{\partial b_j^{L-1}} = \sum_i \delta_i^L w_{ij}^L \sigma' \left(z_j^{L-1} \right).$$

Defining

$$\delta_j^{L-1} \equiv \sum_i \delta_i^L w_{ij}^L \sigma' \left(z_j^{L-1} \right)$$

with the corresponding expression in matrix-vector notation given by

$$\boldsymbol{\delta}^{L-1} = \left[\left(\mathbf{W}^L \right)^T \boldsymbol{\delta}^L \right] \odot \sigma' \left(\mathbf{z}^{L-1} \right),$$

we can write

$$\frac{\partial C_{\mathbf{x}}}{\partial w_{jn}^{L-1}} = \delta_j^{L-1} a_n^{L-2}$$

and

$$\frac{\partial C_{\mathbf{x}}}{\partial b_j^{L-1}} = \delta_j^{L-1}.$$

So the derivatives of the cost function with respect to the weights and biases of the next-to-last layer can be written on the same form as the derivatives with respect to the weights and biases of the outer layer. In fact, if we write the total differential of $C_{\mathbf{x}}$ in terms of derivatives of $C_{\mathbf{x}}$ with respect to the activities z_i^{L-1} of the next-to-last layer, we can find the derivatives of $C_{\mathbf{x}}$ with respect to the weights w_{jn}^{L-2} and the biases b_j^{L-2}, and they are on the same form as well. Repeating this process, with a general δ_j^{l-1} given by

$$\delta_j^{l-1} \equiv \sum_i \delta_i^l w_{ij}^l \sigma' \left(z_j^{l-1} \right)$$

we can find the derivatives of $C_{\mathbf{x}}$ with respect to *all* the weights and biases in the neural network. Therefore, we can use the following algorithm:

```
$\boldsymbol{\delta}^L = \frac{\partial C_{\mathbf{x}}}{\partial \mathbf{a}^L} \odot f'\left(\mathbf{z}^L\right)$
**for** $l = L,\ L-1,\ \ldots,\ 2$ **do**
    $\frac{\partial C_{\mathbf{x}}}{\partial \mathbf{W}^l} = \boldsymbol{\delta}^l \left(\mathbf{a}^{l-1}\right)^T$
    $\frac{\partial C_{\mathbf{x}}}{\partial \mathbf{b}^l} = \boldsymbol{\delta}^l$
    $\boldsymbol{\delta}^{l-1} = \left[\left(\mathbf{W}^l\right)^T \boldsymbol{\delta}^l\right] \odot \sigma'\left(\mathbf{z}^{l-1}\right)$
**end for**
```

3 Method

3.1 Linear Regression - Gradient Descent

We perform linear regression using Ordinary Least Squares (OLS) and Ridge regression methods on data generated using the Franke's function. Gradient Descent (GD) methods are essential for neural networks in finding the optimal weights and biases for the neurons in the network until convergence. We therefore implement Batch Gradient Descent (BGD), Stochastic Gradient Descent (SGD) and Mini-Batch Gradient Descent (MBGD) to assess their performance. Further, we study these GD approximation methods for finding minima when the derivative is 0 with Mean-Squared Error (MSE) with K-fold Cross Validation (CV) resampling. We are going to observe the effect of momentum, the importance of mini-batch size, hyperparameter λ for Ridge and learning rates for Ridge and OLS. With these observations, we are able to find parameters that are suitable. For comparative purposes, sklearn's SGDRegressor and inversion matrix (as a baseline, analytical solution) will also be implemented.

3.2 Logistic Regression

In this project we are preforming logistic regression on the digits dataset from scikit learn. This dataset contains handwritten digits with labels corresponding to their value. To perform the logistic regression on this data set we start by splitting it in test and train data. The test data is what we will use to train the classification function and the train data is what we will use to find the accuracy of the method.

We will find the θ vector needed for the score by using the gradient of the cost function,

$$\theta^{n+1} = \theta^n + \eta \nabla c$$

the η is the learning rate which determines the step size. By using this we will update the θ for each step. The gradient of the cost function is given by,

$$\nabla c = (y - p)^T X - 2\lambda\theta$$

where λ is the regularization rate and p is the probability function. In our case the probability function for the multi-class classification is the softmax function, thus p is,

$$p = softmax(X\theta^T)$$

We use stochastic gradient decent for the logistic regression which means that we will loop over the epochs and the minibatches. Each time we loop we update θ and get new values for the probabilities. This means that the more times we loop the more accurate the probabilities will be in theory. Now we have a probability distribution for the different classes, then we will find which of the classes the instance will be predicted to be by finding the class that has the highest probability. The class that has the highest probability will be the class our classifier puts the digit in. In this specific case it means that we get a probability distribution with probabilities for that image being the different digits from 0 to 9. Then the digit with the largest probability is what it will be classified as.

To find how well the logistic regression predicts the numbers we will test the accuracy using,

$$accuracy = \frac{\sum_1^i I(t_i = y_i)}{n}$$

thus calculating the mean of how many instances the function got right. This accuracy score will give us an insight on how good this method is for predicting this data.

We perform the logistic regression with the l_2 regularization parameter λ being 0 first and then we add the λ parameters with different values to see how this improves our predictions accuracy. In addition to changing the regularization parameter we can change the learning rate, number of epochs and the number of mini-batches to see how these changes will affect the accuracy of our method. We will expect that a larger number of epochs and a larger number of batches will increase the accuracy as it will take smaller steps when determining the θ.

We will also use the scikit learn functionally for multi dimensional classes to see how our own logistic regression compares to that of scikit learn. We will do this because we want to see if our method can be close in accuracy to what can be achieved by implementing pre-existing packages. When looking at the scikit learn results we also look at scaled and un-scaled design matrix to see if the scaling improves the accuracy.

The method we have used here to make a multi dimensional classifier can also be used on a dimensional case by using a different logistic function.

3.3 Neural Network

To assess the performance of our implementation of a neural network, we're using it to solve both a regression problem and a classification problem. In the regression problem we're using the network to model the Franke function. A natural choice of cost function is the mean squared error

$$\text{MSE} = \frac{1}{n}\sum_{\mathbf{x}}^{n}\left(a^L - f(\mathbf{x})\right)^2 .$$

The choice of activation function in the output layer is a simple identity function $f(x) = x$ so as not to restrict the range of output values of the network. Since the Franke function is a scalar function of two variables, the input layer will contain two nodes while the output layer will contain one node in the regression problem.

For the classification problem we're using the MNIST data set, which is a collection of 28×28 pixel images of handwritten digits, each with a label signifying what digit the image is supposed to represent. The input layer will thus consist of $28 \times 28 = 784$ nodes, while the output layer will consist of 10 nodes, each corresponding to the respective digits. The output of the nodes in the output layers will be interpreted as probabilities. A natural choice of activation function in the output layer is thus the softmax function, which ensures that the outputs of the nodes in the output layer will sum up to 1 and be in the range $[0,\ 1]$. As cost function we choose the cross-entropy

$$C = \frac{1}{n}\sum_{\mathbf{x}}^{n}\sum_{j}^{K} y_j \ln a_j^L + (1 - y_j)\ln\left(1 - a_j^L\right),$$

where K are the number of classification classes, and for assessing the performance we compute the accuracy. For both regression and classification we are also going to use a regularization parameter, by adding a term to the respective cost functions:

$$C \to C + \frac{\lambda}{2n} \sum_{jk} w_{jk}^2$$

Here the sum goes over all the weights in the neural network. Now, the best and worst thing about neural networks are the endless amount of parameters that can be tweaked in order to modify the result. These are parameters like the number of hidden layers, the number of nodes in various layers, initialization of weights and biases, the choice of activation function in the hidden layers, the number of epochs, the learning rate, the regularization parameter, and so on. Since we unfortunately live in no more than three dimensions we can only visualize tweaking a few parameters at a time. We can not guarantee that even better results can't be achieved using another set of parameters, because you can always ask ”Ok, but what if you change literally everything?”. We have therefore settled on a ”strategy”, which is completely open to criticism, for achieving the best possible result. The strategy goes as follows:

First of all we're plotting the MSE/Accuracy as a function of the number of hidden layers. There will be 30 nodes in each hidden layer and the choice of activation function in the hidden layers is the sigmoid function.

Second, when the optimal number of hidden layers is found, we plot the MSE/Accuracy as a function of the number of nodes, which will be the same across all hidden layers. Here we will also use the ReLU and Leaky ReLU functions as activation functions in the hidden layers to see which activation function gives the best results.

Third, when the activation function that gives the best results is found, as well as when the optimal number of nodes across all hidden layers is found, we plot the MSE/Accuracy as a function of the learning rate η and the regularization parameter λ.

4 Results and Discussion

4.1 Gradient Descent and Hyperparameters

The GD methods were implemented for OLS and Ridge regression in order to assess their performances on data generated with the Franke function. The parameters learning rate (η), batch size, momentum (γ) and λ were studied first with Mini-Batch GD in OLS for finding the optimal parameters with lowest MSE. The parameters were studied for OLS since it does not take regularization into consideration, and with Mini-Batch GD since it introduces stochasticity for the number of mini-batches and thus makes it easier to escape local minima. With the most optimal parameters, the analytical method inversion matrix and approximate methods BGD, SGD, MBGD and Sklearn's SGDRegressor are plotted for OLS and Ridge regression. OLS is used as benchmark for the results found in previous work [1] where Franke function was studied for OLS, Ridge and Lasso regression.

First, Figure 4.1 shows MSE against number of epochs with different learning rates 0.01, 0.001 and 0.0001 for OLS.

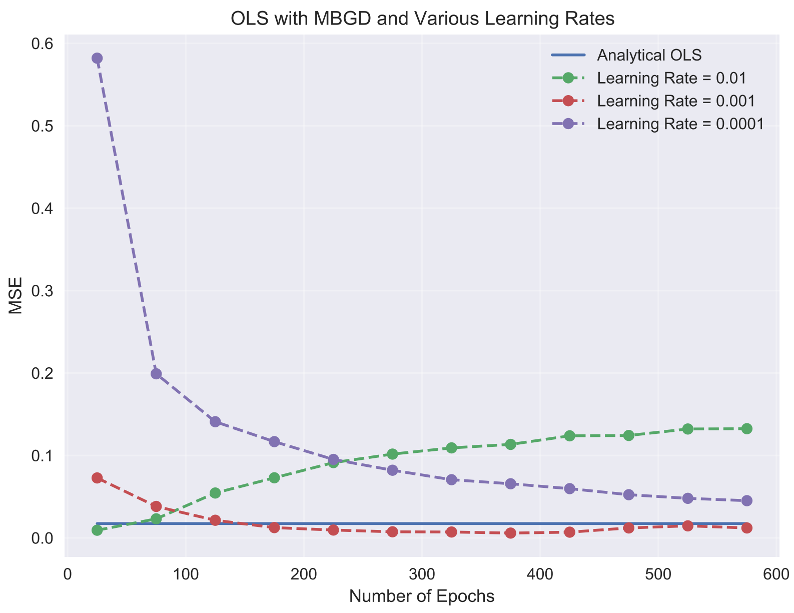


Figure 4.1: *Franke function using OLS optimized with MBGD with momentum for various learning rates of 0.01, 0.001 and 0.0001. The number of total epochs is 600 with step 25. The analytical OLS is also plotted as a baseline.*

From Figure 4.1, the learning rate 0.01 increases slightly with the number of iterations in epochs. This could be due to the step size of 25 being too large. Going a bit higher, a learning rate of 0.001 goes below the analytical OLS after approximately 150 epochs and is stable there. This appears to be a good learning rate. While even smaller learning rate 0.0001 is high for fewer epochs, it also decreases substantially after 600 epochs, but does not reach as low MSE as learning rate 0.001 after 600 epochs.

In Figure 4.2 the MSE is plotted against number of epochs with various batch sizes for Mini-Batch GD for OLS. The batch sizes are 2, 8, 16 and 32.

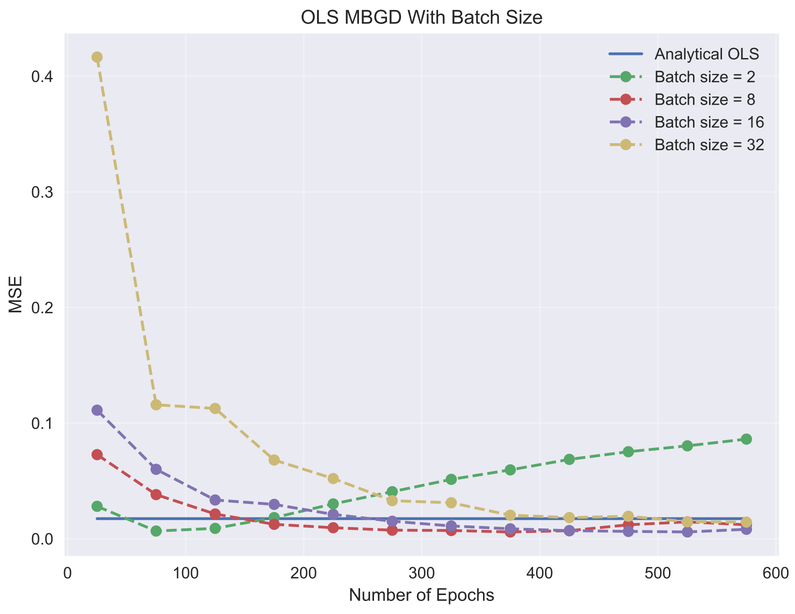


Figure 4.2: *Franke function using OLS optimized with MBGD with momentum for various batch sizes of 2, 8, 16 and 32. The number of total epochs is 600 with step 25. The analytical OLS is also plotted as a baseline.*

Each epoch iterates over the set number of minibatches which is defined as n datapoints divided by M mini batch size. In Figure 4.2, the batch size 2 might increase due to the step size being too large. Higher batch sizes of 8 and 16 progress faster and level out below analytical OLS with MSE of ∼0.001. A batch size of 32 also progresses faster and requires more epochs in order to converge around the same MSE as batch sizes 8 and 16. A batch size of 8 is better preferred due to its generally low MSE.

Moving on to Figure 4.3, the various momentum parameters (γ) are plotted with MSE against number of epochs for OLS with Mini-Batch GD.

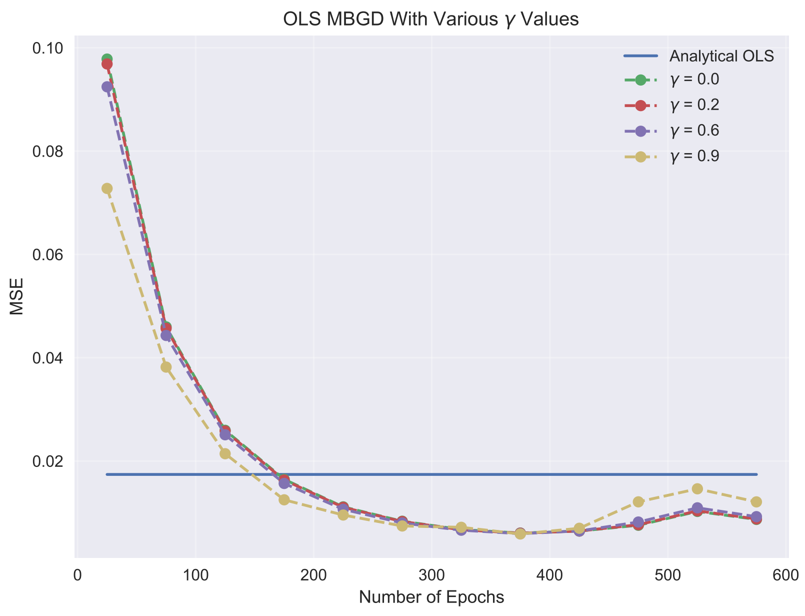


Figure 4.3: *Franke function using OLS optimized with MBGD for various momentum parameters γ of 0.0, 0.2, 0.6 and 0.9. The number of total epochs is 600 with step 25. The analytical OLS is also plotted as a baseline.*

It can be observed in Figure 4.3 that $\gamma = 0$ is highly similar to $\gamma = 0.2$ and $\gamma = 0.6$, implying the small effect of adding small momentum to the Mini-Batch GD optimization. However, by increasing the momentum parameter to 0.9, the convergence is faster, reaching a lower MSE sooner than the lower γ-values. The general effect is therefore an increase in convergence rate, especially if the momentum parameter is closer to 1. The penalty parameters all increase a bit after 400 epochs, especially $\gamma = 0.9$, which could be caused by overfitting. Overfitting implies that our model stop learning the data, and memorizes them instead. For NN, it could therefore be wise to stop the training when MSE on test data is at its minimum.

Regarding penalty parameter λ, regularization is introduced in Ridge and $\lambda = 0$ is without regularization. The effect of adding penalty is observed in Figure 4.4.

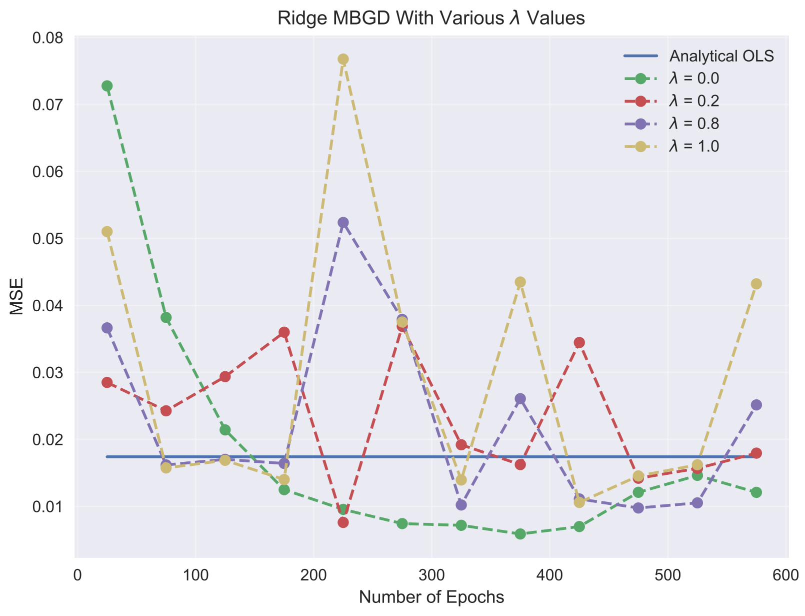


Figure 4.4: *Franke function using Ridge (OLS for $\gamma = 0$) optimized with MBGD for various penalty parameters of 0.2, 0.8 and 1.0. The number of total epochs is 600 with step 25. The analytical OLS is also plotted as a baseline.*

In Figure 4.4, OLS with $\gamma = 0$ gradually converges and is subjected to overfitting after ∼400 epochs as observed in Figure 4.3 where the curve corresponds to $\gamma = 0.9$. For regularization with Ridge, an occurring trend is more fluctuations for higher λ. MSE quickly converges before destabilizing with more oscillations. These fluctuations could be due to the increasing penalty for the higher λ values. This in turn affects the update rule of GD. The l2-penalty punishes the weights behaving poorly (poor explanatory variables). On the other hand, for "well-behaved" explanatory variables, a constant term is subtracted. This penalization prevents over-fitting, but could also lead to less precision in Figure 4.4.

Based on the figures above, optimal parameters for weights and biases for neurons in a neural network might include; $\eta = 0.001$, batch size $= 8$, $\gamma = 0.9$ and $\lambda = 1$ (Ridge). Even though the parameters will also need manual tuning and adjusting for the weights and biases, they can still provide an impression of their performance. In order to study the behavior and effect of the parameters further, we could have run for more epochs which is more computationally expensive. A grid search could also have been applied in order to search through a subset of the hyperparameter space.

The parameters found here are set as "default" values for the GD methods we will now discuss. The GD methods Batch GD, Stochastic GD, Mini-Batch GD and Sklearn's SGDRegressor are implemented and plotted for MSE against number for epochs for OLS (Figure 4.5) and Ridge (Figure 4.6). The analytical inversion matrix is also implemented.

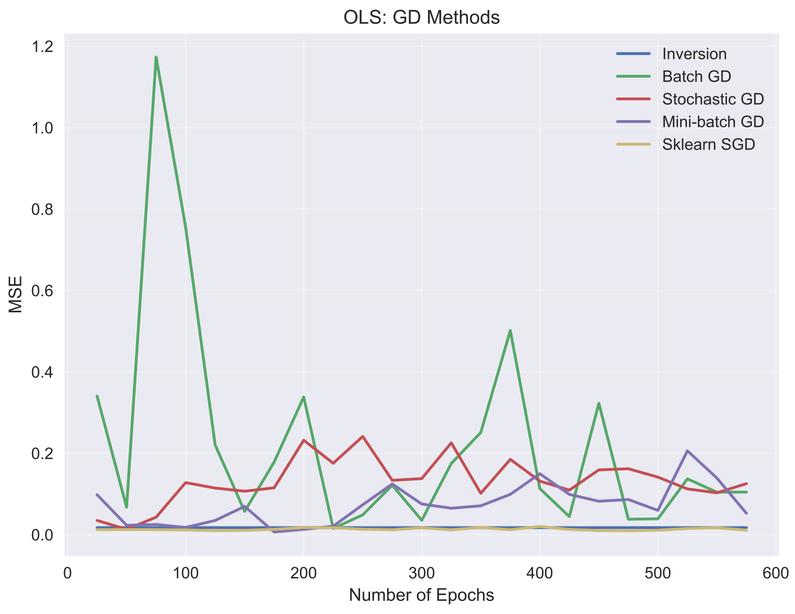


Figure 4.5: *Franke function using OLS for various optimization GD methods. The number of total epochs is 600 with step size 25.*

As expected, MBGD moves closer to the minimum than SGD and BGD in Figure 4.5 due to its stochasticity ultimately making it easier to escape local minima. For BGD, the MSE at 1.2 is high after around 100 epochs. This could imply that it did not reach as far as it should and might be stuck in a minimum and generalizes poorly. After running for more epochs, it seems to fluctuate around a minimum. The fluctuations in SGD also imply that it is fluctuating around a minimum, but around a lower MSE than BGD.

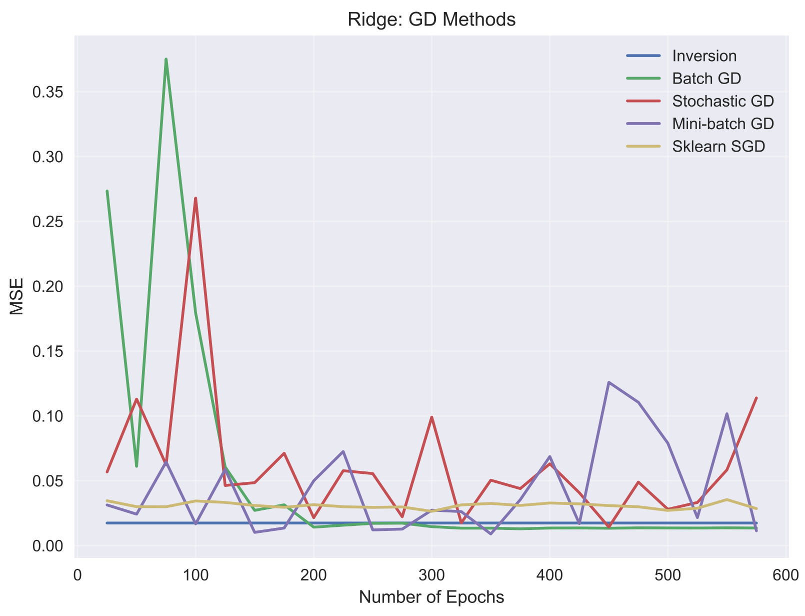


Figure 4.6: *Franke function using Ridge for various optimization GD methods. The number of total epochs is 600 with step size 25.*

In general, the MSE is lower for Ridge than OLS as observed in Figure 4.6. This is due to the penalization introduced in Ridge that prevents it from over-fitting, and thus provides a more stable MSE compared to OLS in Figure 4.5. BGD is high after 100 epochs and converges slower since it uses all data points for each iteration. However, once it converges, the MSE is lower and more stable than for SGD and MBGD. SGD converges faster than BGD since the parameters are being updated for each iteration (one training example in each iteration). MBGD generally has low MSE, but after 400 epochs it increases which could be due to overfitting. Therefore, for NN, it could be wise consider stopping the training after 400 epochs, or increase the step size.

4.2 Logistic Regression

When using logistic regression to classify the different digits in the dataset we have used the stochastic gradient decent method. This method is a function of the number of epochs. We thus want to see how the accuracy of the logistic regression changes with the change in number of epochs. Here we have used from 10 to 1000 epochs to see how this would impact the accuracy.

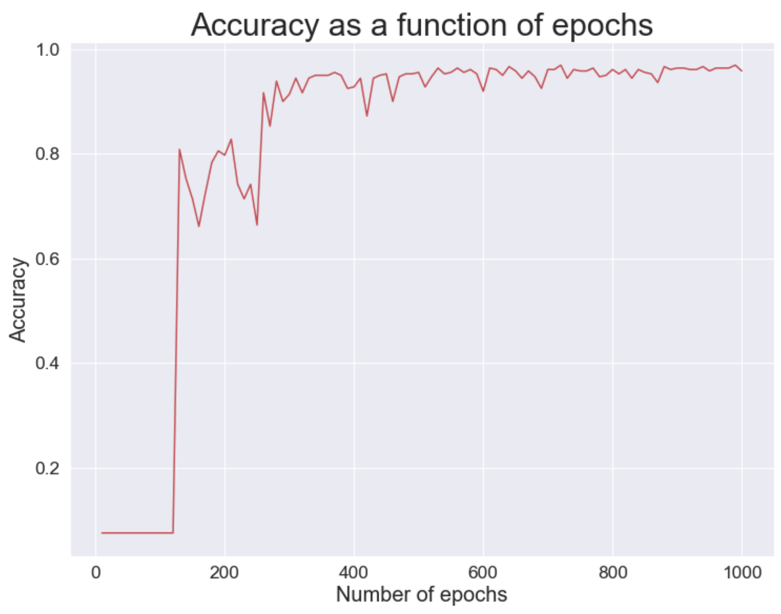


Figure 4.7: *Accuracy as a function of the number of epochs used*

From figure 4.7 we see the graph of the accuracy as a function of epochs. Here we have made a new fit for the logistic regression for each change in the number of epochs. We can see from the graph that the accuracy is low where we have a low number of epochs. This is expected as a low number of epochs means that we loop over fewer times and thus take larger increments when finding the θ value. However we see that when the number of epochs surpasses 500 there is no longer a large increase in accuracy. At this point the accuracy seems to be fluctuating a little bit, but not getting any better. This might be because at some point the ideal amount of epochs is reached and having more epochs will not be beneficial.

Another of the parameters we can change when fitting the logistic regression to our classification is the l2 parameter λ. This is the parameter we can change in the gradient expression. This parameter changes the value for the gradient and thus changes the fitting function for the logistic regression. We have thus chosen to again plot the accuracy as a function of this parameter to see which l2 regularization parameter will give the best fit.

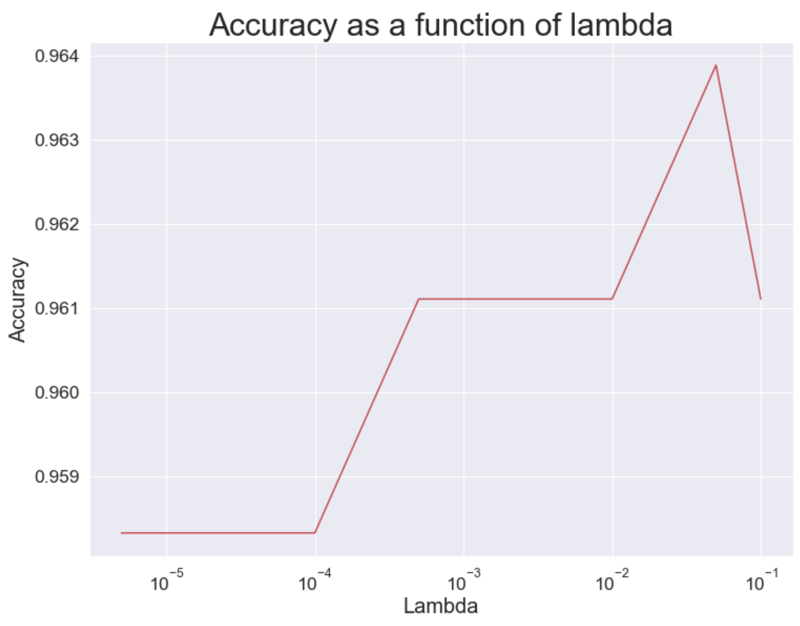


Figure 4.8: *Accuracy as a function of l2 regularization parameter λ*

From figure 4.8 we see that the accuracy changes as we change the λ value. We have here calculated the accuracy for 10 different λ value ranging from 10^{-5} to 10^{-1}. The plot shows that from these values for the parameter the best accuracy is achieved with of about $\lambda = 5 \times 10^{-2}$. The two parameters we have looked at individually can also influence each other, thus we also want to look at a heapmap of accuracy with both the number of epochs and the λ value.

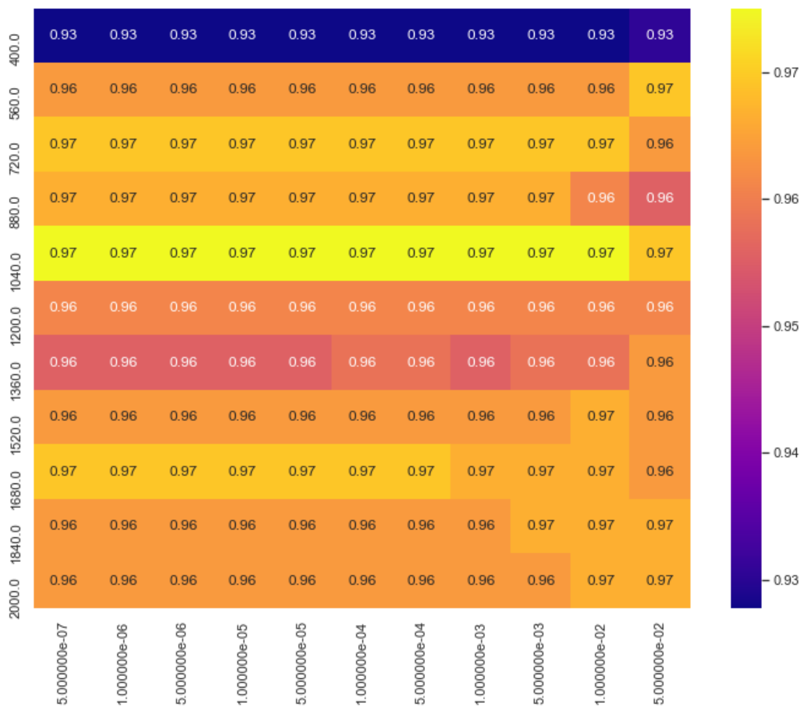


Figure 4.9: *The heatmap shows the accuracy when changing both the number of epochs and the λ value.*

From this heatmap we see that the greatest accuracy is at around 1000 epochs with a low λ value. However, we also see from this map that there are multiple combinations of the number of epochs and the λ values that give higher accuracy, and that the accuracy thus depends on both.

Now we have found the accuracy we can get on this data set by using our own logistic regression functionality. We can compare this to what we get when using scikit learn, we have used the LogisticRegression method for multiclasses in scikit learn.

Table 4.1: *Accuracy scores for the different logistic regression methods*

sklearn no scaling	sklearn with scaling	logistic regression $\lambda = 0$	logistic regression $\lambda = 0.005$
0.95556	0.96	0.95833	0.96388

We see here that the logistic regression method we use with SGD and the softmax function is gives a better accuracy than we get when using scikit learn. The accuracy when using the logistic regression method in scikit learn is better when we use scaled values for the design matrix. Our own logistic regression method is better when we use the optimal value for the regularization parameter than without one.

4.3 Neural Networks

4.3.1 Regression with Neural Networks

To analyse the Neural Network we have made on a regression case we use the Franke Function that we have also used in project 1 [1]. We want to find out how our Neural Network will give the lowest MSE by trying to change the number of hidden layers, the number of nodes in the hidden layers, the learning rate and the regularization parameter. To do this we thus fit the Franke Function with the Neural Network while chaning the parameter and calculating the MSE of each fit. We start by looking at the number of hidden layers that will give the lowest MSE, when doing this we hold the other parameters constant and only change the number of hidden layers, from zero to four. When changing the number of hidden layers we will keep the learning rate $\eta = 0.3$ the regularization parameter $\lambda = 1 \times 10^{-6}$, the number of epochs to 30 and the number of mini-batches to 10. Then we calculate the MSE as a function of the number of hidden layers using Sigmoid, ReLu and leaky ReLu as activation functions.

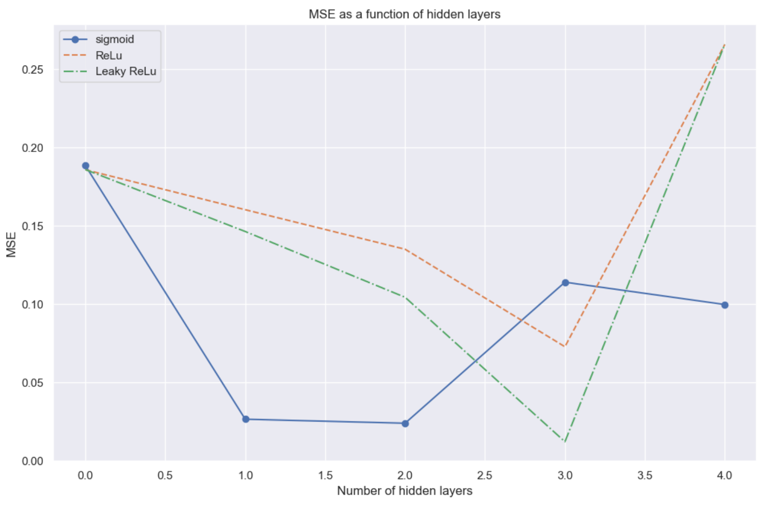


Figure 4.10: *MSE as a function of number of hidden layers using different activation functions.*

When doing the calculations for four hidden layers with ReLu and leaky ReLu we got overflow, so for four hidden layers these don't have reliable results. From this graph we see that the MSE for Sigmoid is lowest when we use one or two hidden layers. For ReLu and leaky ReLu we have the lowest MSE when we have three hidden layers. We see that for all three of the graphs zero hidden layers give a larger MSE. In addition four hidden layers also gives a larger MSE, however as mentioned we have overflow in ReLu and leaky ReLu for four hidden layers in the calculation of the Neural Network so this last point may not be accurate. We know that in the Neural Network a higher number of hidden layers will be more likely to lead to overflow, so we choose to use one hidden layer as this is also one of the lowest MSE when using the Sigmoid function.

Now that we have found the number of hidden layers that is optimal when working with the Franke Function we want to find the optimal number of nodes in this hidden layer. To do this we use the same values as above, and use only one hidden layer where we change the number of nodes in this hidden layer. We will change the number of nodes from 10 to 80 to see which will give the lowest MSE. Here we will consider the activation functions Sigmoid, ReLu and leaky ReLu.

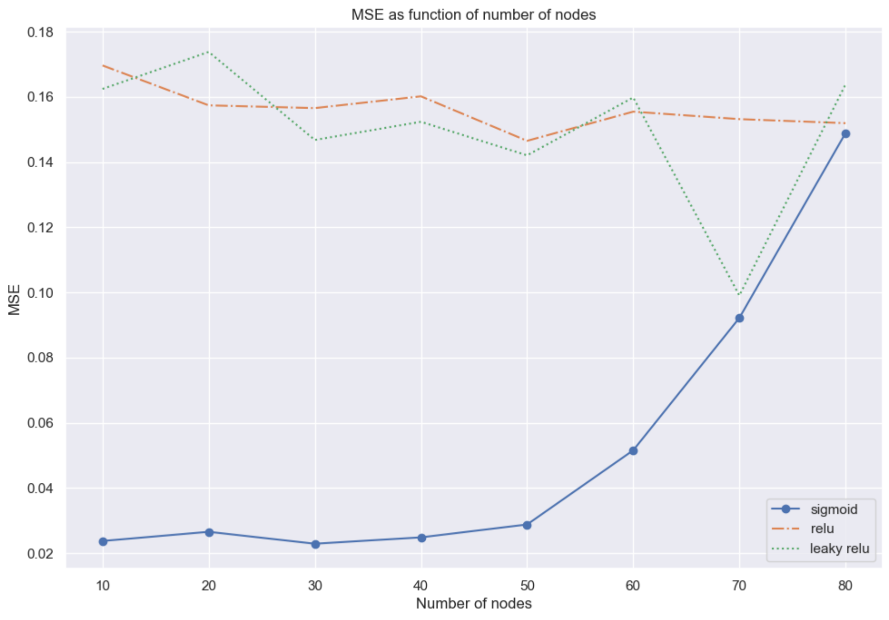
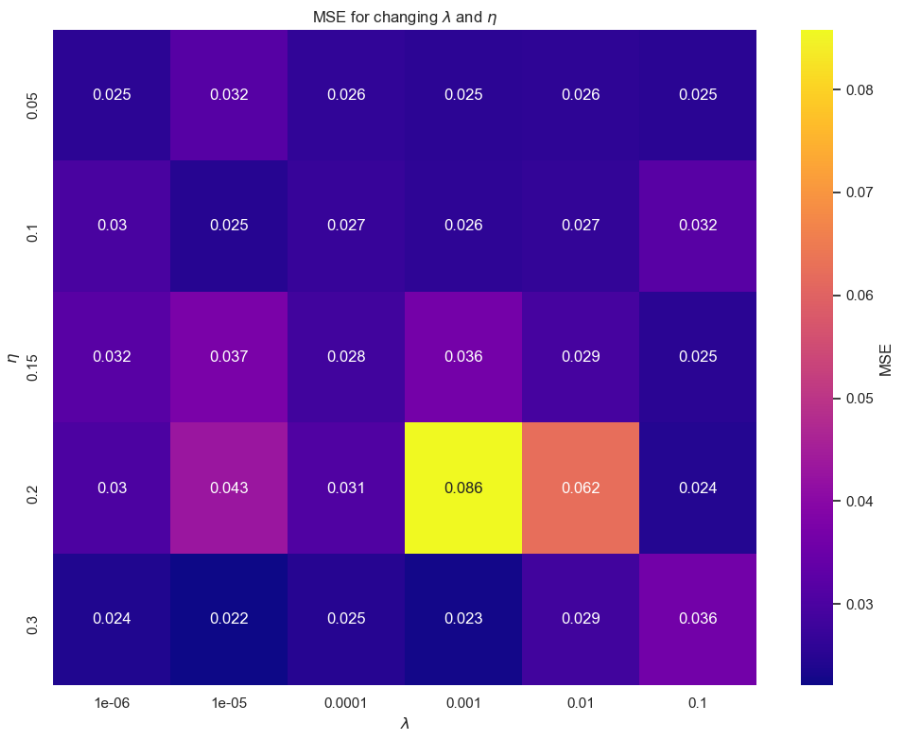


Figure 4.11: *MSE as a function of nodes in the hidden layer*

From the graphs in 4.11 we see that the MSE when using Sigmoid is lowest with 30 nodes, ReLu is lowest with 50 nodes and leaky ReLu is lowest with 70 nodes. We also see that in general the MSE is higer in ReLu and in leaky ReLu in this image, this is because we have chosen to look at one hidden layer as this was optimal when using Sigmoid as activation function and because more hidden layers would cause overflow when running the Neural Network. Thus, we can see that as long as we only want to use Sigmoid as activation function when predicting the Franke Function we have to use 30 nodes in the hidden layer. We will use this when we look at the optimal values for the learning rate and the regulatization parameter.

To find the lowest MSE we can get for the Franke Function with our Neural Network for regression we also need to find the best value for the learning rate η and the regularization parameter λ. We do this again by changing only these two parameters and making a matrix of MSE values. This gives us a grid of MSE values for the different values of the parameters. As we have done previously we will set the number of epochs to be 30, minibatches to be 10 and we will have one hidden layer with 30 nodes. We will here only be using Sigmoid activation function as we found this has the lowest MSE.



η \ λ	1e-06	1e-05	0.0001	0.001	0.01	0.1
0.05	0.025	0.032	0.026	0.025	0.026	0.025
0.1	0.03	0.025	0.027	0.026	0.027	0.032
0.15	0.032	0.037	0.028	0.036	0.029	0.025
0.2	0.03	0.043	0.031	0.086	0.062	0.024
0.3	0.024	0.022	0.025	0.023	0.029	0.036

Figure 4.12: *MSE for the different learning rates and the regularization paramets*

From figure 4.12 we see the MSE values when we change the η and the λ values. We see that there are small changes in the MSE generally, but we have two points, $\eta = 0.2$, $\lambda = 0.001, 0.01$ where the MSE is larger. We get the lowest MSE $= 0.023$ at $\eta = 0.3$ and $\lambda = 0.001$.

In total we thus have that the lowest MSE we were able to reach is 0.023, with Sigmoid activation function, 30 epochs, 10 mini-batches, one hidden layer with 30 nodes and $\eta = 0.3$ and $\lambda = 0.001$. This is a higher MSE than we were able to obtain for the Franke Function with linear regression in project 1, [1]. In general we did expect the Neural Network to give a lower MSE, however since we have made our own function and there is overflow in this function when using a large number of hidden layers this might be a contributing factor to the discrepancy in expectation and result. If we, in the future, would be able to make the Neural Network more stable for higher number of hidden layers we would likely be able to get a lower error.

4.3.2 Classification with Neural Networks

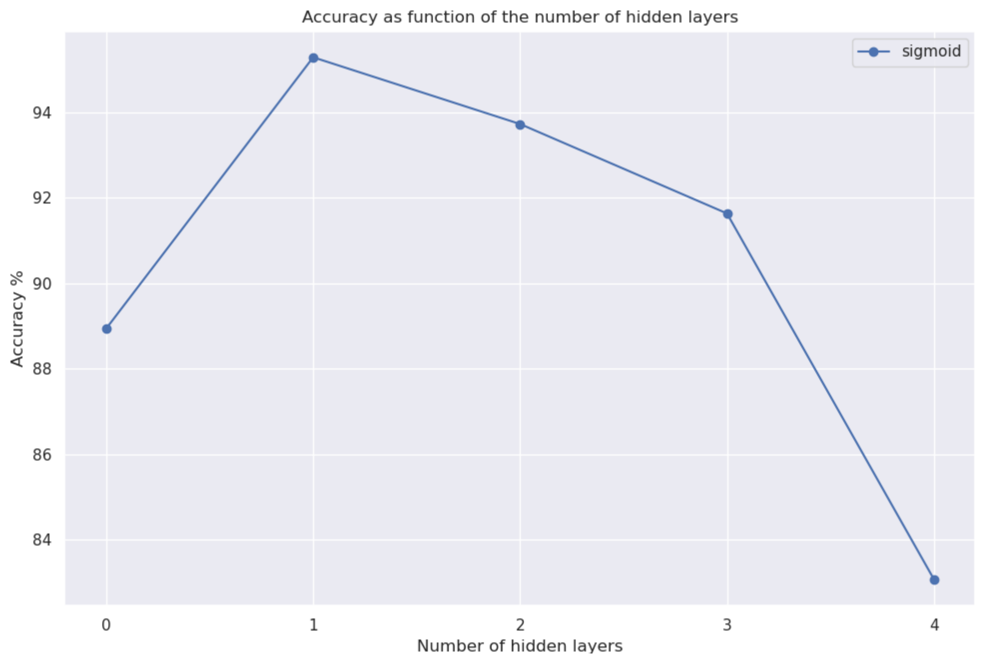


Figure 4.13: *Accuracy as a function of the number of hidden layers, each containing 30 nodes. The sigmoid function was used as activation function in the hidden layers.*

Figure 4.13 shows the accuracy as a function of the number of hidden layers. We see that a respectable accuracy is achieved, with an accuracy topping at $\sim 95\%$ when using a single hidden layer. Therefore we have used a single hidden layer in the generation of all plots that follow. We see that there is a steady decrease in the accuracy when increasing the number of hidden layers to 2 and 3, and then a significant drop in the accuracy when increasing the number of hidden layers from 3 to 4, although it is still over 80%. For this reason we did not increase the number of hidden layers even further. In the generation of these results we used a learning rate of $\eta = 3.0$ and set the regularization parameter to $\lambda = 0$. 30 epochs and a mini-batch size of 10 was used in stochastic gradient descent.

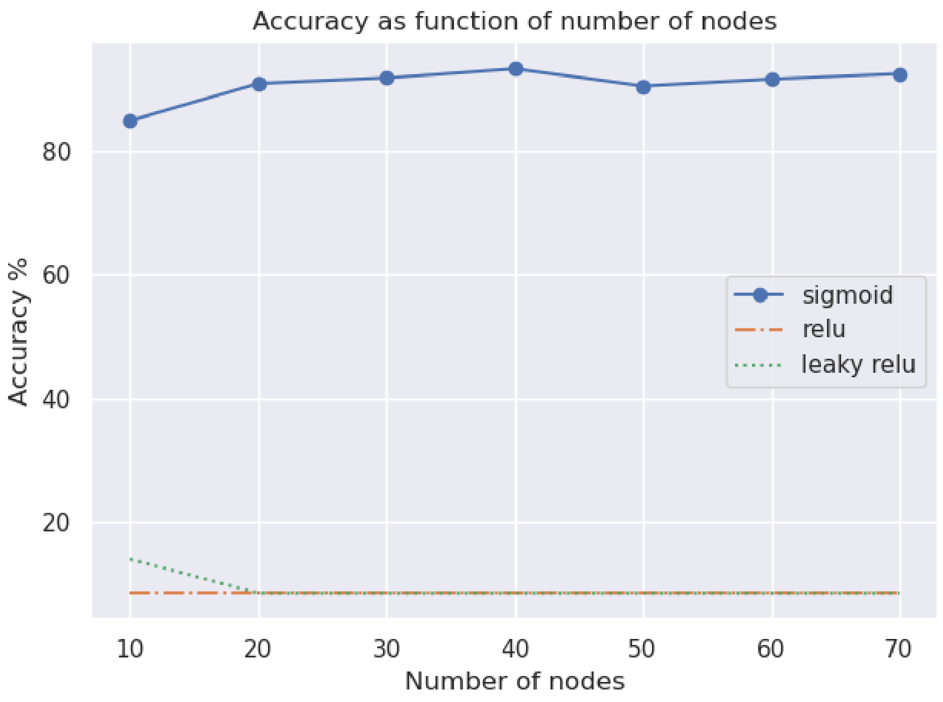


Figure 4.14: *Accuracy as a function of the number of nodes when using a single hidden layer when using sigmoid, ReLU or Leaky ReLU as activation functions.*

Figure 4.14 shows the accuracy as a function of number of nodes when using sigmoid, ReLU and Leaky ReLU as activation functions respectively. We see that there is a stark contrast in the performance of the network when changing the activation function from sigmoid to ReLU or Leaky ReLU. Now, admittedly, we struggled with overflow when using ReLU or Leaky ReLU, but we decided to plot the results nonetheless. But for this reason, and because the network seemingly performs no better than guessing digits at random when using ReLU and Leaky ReLU, these results must be taken with a large grain of salt.

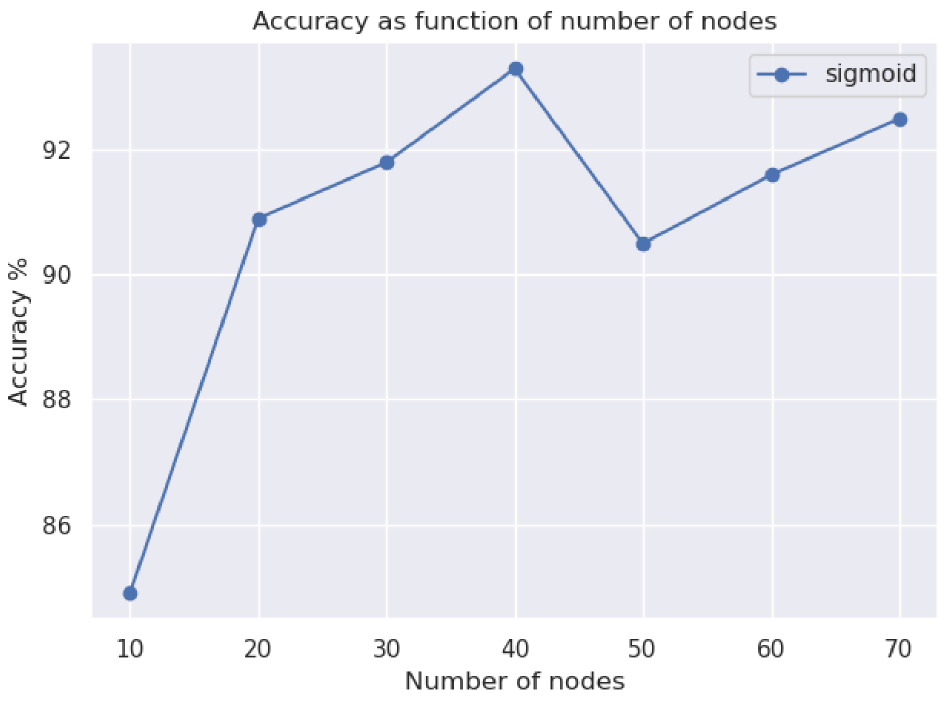
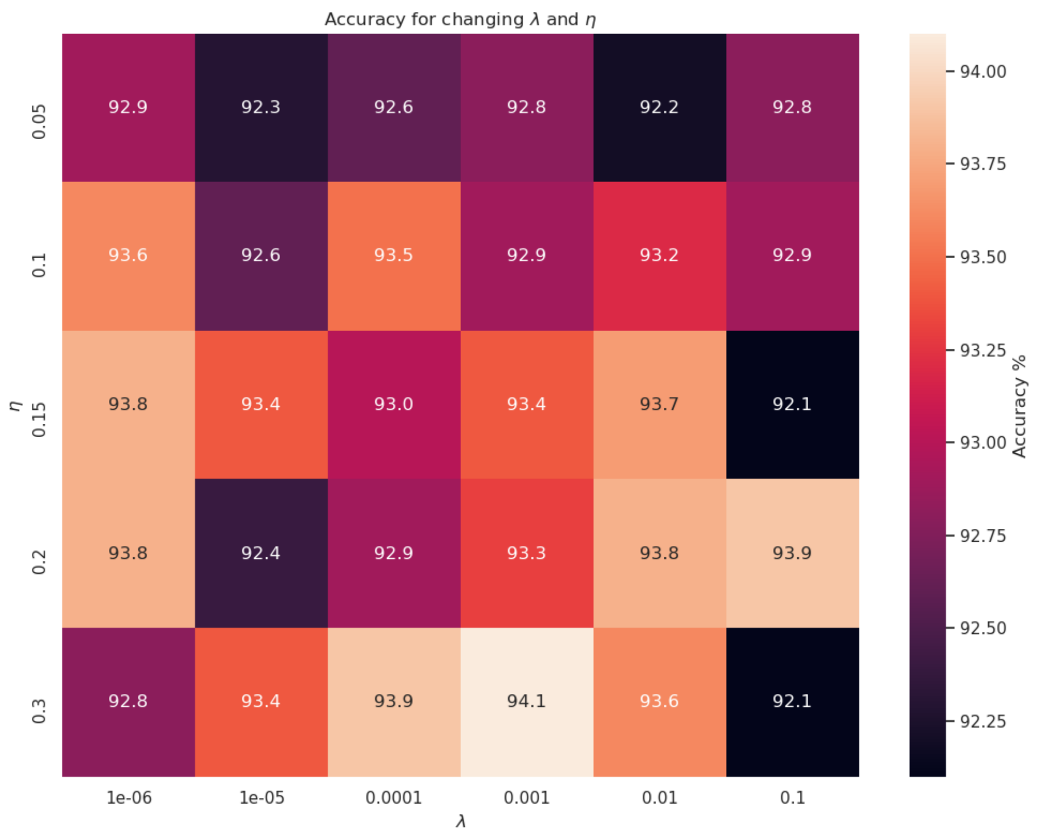


Figure 4.15: *Zoomed in version of Figure 4.14, showing the accuracy as a function of the number of nodes when using sigmoid as activation function.*

For sigmoid however, we do get results that are (at least somewhat) consistent with Figure 4.13, with a respectable accuracy over 85%, topping at $\sim 93\%$ when using 40 nodes. This is what we see in Figure 4.15. The accuracy barely changes when using anything between 20 and 70 nodes, fluctuating around 92%. This is a surprising result, considering that in Figure 4.13 there was a significant decrease in the accuracy when increasing the number of hidden layers from 1 up to 4. We cannot come up with any obvious reason why the accuracy decreases significantly when increasing the number of hidden layers, but only fluctuates when increasing the number of nodes when using a single layer. Both ways involve increasing the total number of nodes in the network one way or the other, so this is a surprising result indeed. Now, as you may have already been able to tell, there is one data point that both Figure 4.13 and Figure 4.15 share, namely the accuracy when using a single layer with 30 nodes. In Figure 4.13 we got an accuracy of $\sim 95\%$, while in Figure 4.15 we get an accuracy of slightly below 92%. This suggests that our results are not quite stable. There are two factors that bring some randomness into our results. One is the random shuffling of the training data between epochs in stochastic gradient descent. The other is the initialization of the weights, all being drawn from a normal distribution. This is something that we, quite frankly, should've taken into account, and could've taken into account by using e.g. some variant of the cross-validation resampling technique.



η \ λ	1e-06	1e-05	0.0001	0.001	0.01	0.1
0.05	92.9	92.3	92.6	92.8	92.2	92.8
0.1	93.6	92.6	93.5	92.9	93.2	92.9
0.15	93.8	93.4	93.0	93.4	93.7	92.1
0.2	93.8	92.4	92.9	93.3	93.8	93.9
0.3	92.8	93.4	93.9	94.1	93.6	92.1

Figure 4.16: *Heatmap of the accuracy as a function of the learning rate η and the regularization parameter λ.*

Figure 4.16 shows a heatmap of the accuracy as a function of the learning rate η and the regularization parameter λ. We see that we've gotten a respectable accuracy of above 92% in all cases, but there is little change in the accuracy from one datapoint to another. To see a larger variation in the accuracy we should probably have chosen a wider range of values for η and λ to get a better overview. With the plot that we've got there is no clear pattern that shows whether we should increase or decrease the respective parameters. However, the squares seem to appear somewhat more purple when decreasing the learning rate. This is to some extent expected, because a very small learning rate leads to the weights and biases barely being updated in each iteration so that the network "learns" slowly. But again, there is a very small variation in the accuracy from one square in the heatmap to another and we need to choose a wider range of learning rates and regularization parameters to get a better picture of how these affect the accuracy.

5 Conclusion

For OLS and Ridge regression methods, gradient descent optimization was implemented. We observe better performance with momentum, a learning rate of 0.01, Mini-Batch gradient descent with batch size 8 and for about 400 epochs. The parameters can indicate suitable parameters for the neurons in the neural network in finding optimal weights and biases.

We see that the accuracy for the logistic regression is high if we use a high number epochs and the optimal regularization parameter, thus indicating this is a good way to classify the data.

We implemented a neural network in order to solve both a regression problem and a classification problem. In the regression problem we were able to achieve a decently low MSE, but still higher than what we were able to achieve using more traditional regression methods like OLS and Ridge regression. We conclude from this that a neural network is suitable for solving regression problems, but with a somewhat lacking performance compared to traditional methods. In the classification problem we used a collection of images of handwritten digits and were able to reach a respectable accuracy at $\sim 95\%$. This is almost even with the accuracy that we were able to achieve using logistic regression, where we reached an accuracy at $\sim 96\%$. Due to the endless amount of parameters of a neural network, there is lots of room for future studies to see if we can get an accuracy with a neural network that surpasses that of logistic regression.For example, one could experiment with the initialization of the weights and the biases and study the accuracy as a function of epochs to see how quickly the network learns. This could potentially reduce the computational complexity significantly.

References

[1] Erik Alexander Sandvik Oda Hovet Ilse Kuperus. "FYS-STK4155 Project 1: Regression Analysis and Resampling Methods". In: (Oct. 2020). URL: https://github.com/erikasan/fys-stk-project1/blob/master/project1/report/FYS_STK_4155___PROJECT_1.pdf (visited on 11/07/2020) (cit. on pp. 1, 13, 22, 24).

[2] Morten Hjorth-Jensen. *Data Analysis and Machine Learning lectures: Optimization and Gradient Methods*. URL: https://compphysics.github.io/MachineLearning/doc/pub/week39/html/week39.html (visited on 11/02/2020) (cit. on p. 3).

[3] Aurelien Geron. *Hands-on machine learning with Scikit-Learn and TensorFlow*. First edition. Beijing: O'Reilly, 2017, pp. 111–121, 294–295. ISBN: 9781491962299 (cit. on pp. 3, 4).

[4] Morten Hjorth-Jensen. *Data Analysis and Machine Learning lectures: Optimization and Gradient Methods*. URL: https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40.html (visited on 11/03/2020) (cit. on p. 4).