Project 2 - FYS-STK3155

November 13, 2020

1 Abstract

In this second project our goal has been to study gradient descent methods, logistic regression, and neural networks. With these methods we have explored classification and regression problems looking at the same Franke data we saw in project one as well as a new dataset consisting of handwritten digits. We have explored how gradient descent methods are a useful and general solution for finding the minimum of our cost function and as such optimize a model. In coding the solutions we have seen first hand the similarities, and differences, of logistic regression and regression with a feed forward neural network.

We have seen how our neural network code is a versatile tool for solving many different problems, from regression to multinomial classification, and how we can modify the architecture of such a network to fit it to the problem at hand.

We have found our neural network code to outperform our linear regression methods from project 1, ordinary least squares, ridge and lasso when fitting the models to the noisy Franke data, and we have seen the same model (with a different architecture and various choices for the hyper-parameters) outperform logistic regression classifying a series of handwritten digits.

In fitting our models we have seen that it is necessary to put a lot of work into selecting the optimal hyper-parameters in order to achieve the optimal performance. We have seen that some models are very sensitive to the choice of certain parameters, like the learning rate, and selecting one outside of the optimal range can give very poor results.

Especially for neural networks, where the number of hyper-parameters to tune can be astronomical, finding good choices can be very challenging.

2 Introduction

In machine learning, once you know your data, the key question is what model to use to best represent these data. This is no trivial task, and while the advances in the field since its infancy in the 1960's or so have given rise to a plethora of useful and amazing models and techniques have given us many great tools for the task, finding the optimal choice can be a challenge.

Familiarity with the data to model is key in making this decision, and likely to give you a good first selection in possible models and techniques to employ. Once this is done a natural choice is to try out the most promising subset of models on the test data to see how they perform. With the increasing complexity of many models, particularly neural networks, testing a model on the data is not necessarily a trivial task however. The models able to solve the most difficult tasks and model the most complex data tend to have a very large set of parameters to tune, including making choices for things like the network architecture (for neural networks), the choice of cost functions, etc.

With sub-optimal parameters what woul actually be a great model choice can look like a a complete miss when applying it to the data at hand, and finding good parameters for every model type considered is rarely feasible. This situation highlights one of the key issues when employing machine learning in your field of choice, and while it is not a trivial one, putting in the work may yield impressive results.

Another issue when adjusting large amounts of parameters in a complex machine learning model is that understanding the model and how it draws insight from the input data becomes very difficult, and sometimes even impossible. This is a clear drawback of overly complex model. In a scientific setting when trying to understand some system our suggestion is therefore to prioritize simpler models whenever they produce satisfactory results. If however the goal is simply predictory, for instance for a machine scanning post addresses containing handwritten numbers te see where mail should be sent, prioritizing performance is natural.

In this project we look at a few models, namely our set of linear models from project one, logistic regression, and a feed forward neural network. We fit the different hyper-parameters for the models and study their performance. What we find is that while adjusting hyper-parameters is not trivial, and probably not considered the most exciting task, it does pay dividends in our results.

3 Data

3.1 Franke Function and Terrain Data

We will revisit these datasets from project one. For further description refer to the report for project 1 [1]

3.2 MNIST Handwritten Numbers

For our classification problem we will be using the MNIST database of handwritten numbers, or more specifically a subset of this available as the digits dataset in `scikit-learn`[2].

This dataset contains pixel images of handwritten numbers. In total the digits dataset contains 1797 8x8 images. Meaning we have a dataset consisting of 1797 data points, each with 64 descriptors or inputs, and one output, the label.

For a plot of one such data point, see figure 1.

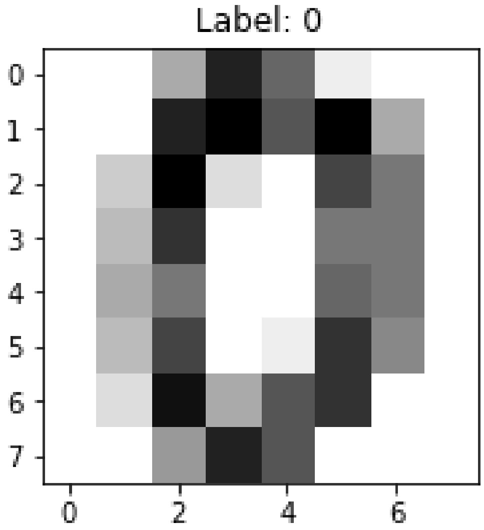

Figure 1: Plot of one of the entries in the digits dataset showing a number 0.

We will use this dataset to fit and test our classification models. To prepare the data we split it into a training set and a test set. I use `scikit-learn`'s `train_test_split`() for this and an 80-20 split.

We want to make sure that the different classes, i.e. the labels are similarly distributed among the test set and the training test. If for instance no occurrences of the label '3' is found in the training set, models trained on this set are expected to perform poorly on identifying such digits in the test set. To avoid such issues we set the `stratify` parameter in `train_test_split()` to the target variable. The result is seen in figure 5, and we can see that the results are satisfactory; the labels appear to be very evenly distributed.

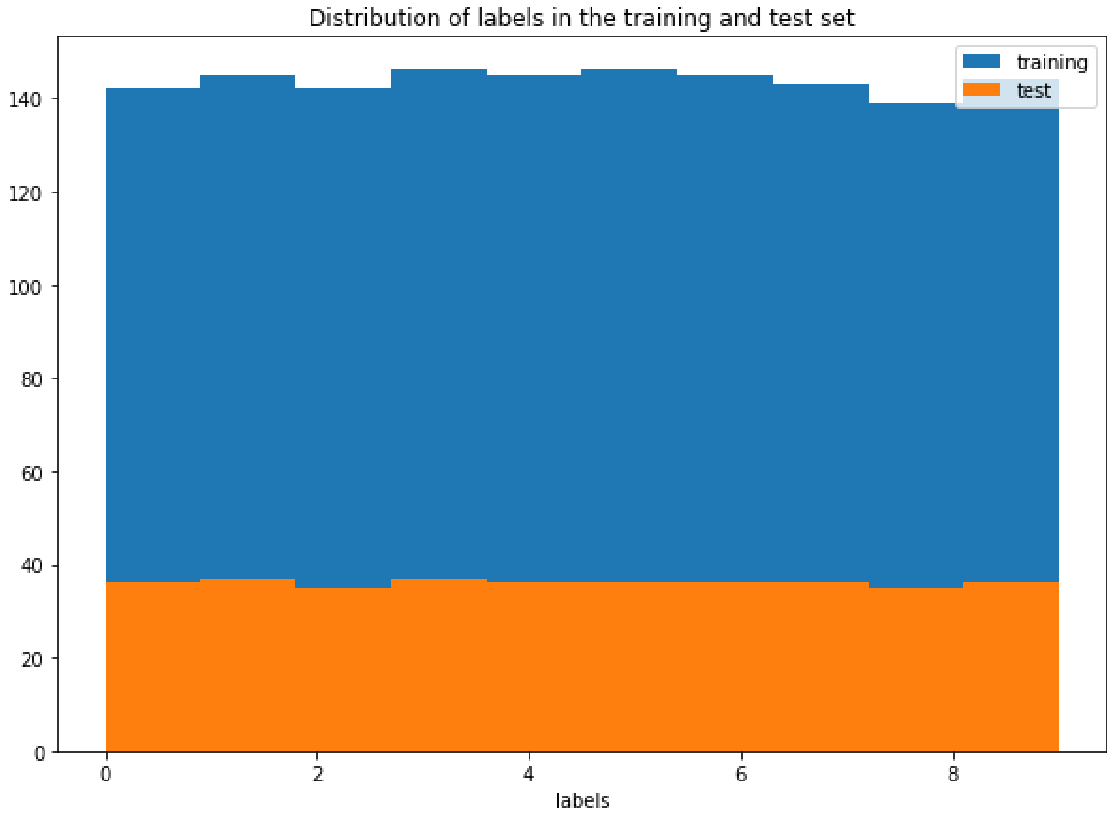

Figure 2: Histogram of label distribution in training and test set.

Figure 2 shows the distribution of the different labels int the test and training sets. We see that the labels seem to be very evenly distributed.

4 Methods

For descriptions of ordinary least squares and ridge regression, please refer to my report for project 1[1]. ## Performance Measures For a description of the mean square error and r^2 please again consult the report of project 1[1].

4.0.1 Accuracy

We use the accuracy score to measure the performance of our classification methods. This measure is given by the number of correctly guessed targets t_i divided by the total number of targets, that is

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n},$$

where I is the indicator function, which takes the value 1 if $t_i = y_i$ and 0 otherwise (for a binary classification problem). Here t_i represents the target and y_i is the prediction. The number of targets is given by n.

4.1 Regression Methods

For description of regular linear regression see Methods section in project 1 [1].

4.1.1 Logistic Regression

Logistic regression is a machine learning algorithm typically used for classification problems. It is most typically applied in binary classification problems, e.g. true/false, yes/no, positive/negative, but it can be extended to problems with multiple classes, called multinomial logistic regression.

I logistic regression a logistic function is applied to model the dependent variable. Such a logistic function takes in any input value, and will always return a value between zero and one. This is often viewed as transforming the input and outputting a probability value. In classification we are interested in discrete output. Some functions, like the step function (see figure number) produces a binary zero or one, while we for other functions, like the sigmoid translate the output values according to a cutoff, e.g. output above or equal 0.5 gets coded as a one.

The classic function used to transform the input in logistic regression is the sigmoid, see the section on activation functions and figure 4.

The general form of logistic regression is

$$\hat{y_k} = g(\beta X_k),$$

where $g(z)$ is the sigmoid and β are weights fit to the inputs in training the model.

While we in linear regression used the mean square error as the cost function, this cost function will we non-convex for the logistic case. This makes finding the minimum difficult, and for this reason we cannot use it as our cost function for logistic regression. To find a cost function we can use the maximum likelihood estimator. In the binary case this has the form,

$$P((x_i, y_i)|\hat{\beta}) = \prod_{i=1}^{n} \left[p(y_i = 1|x_i, \hat{\beta})\right]^{y_i} \left[1 - p(y_i = 1|x_i, \hat{\beta}))\right]^{1-y_i}$$

which by taking the logarithm and reordering leads to the cross-entropy

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

The cross-entropy will be a convex function of the weights $\hat{\beta}$. To train our model we want to minimixe the derivative of the cost function with respect to the weights β.

Multinomial Logistic Regression[9] In multinomial logistic regression, also known as softmax regression we use the same general procedure as in binomial logistic regression, but now the output of the model must be coded into discrete values using several ranges. If for example we are interested in classifying our input to one of four classes A, B, C and D we can code an output between 0 and 0.25 to A, an output between 0.25 and 0.5 to B and so forth.

The *Softmax* function is the preferred output or activation function for multinomial logistic regression. For a classification problem with K classes, the softmax gives the probability for each class.

Using maximum likelihood and a set of K classes k_i we get

$$\prod_{i=1}^{N}\prod_{k=1}^{K} P(y_i = k|x_i, \beta)$$

with the probabilities given by the softmax function

$$P(y_i = k|x_i, \beta) = \frac{\exp(\beta^{(k)} x_i)}{\sum_{j=1}^{K} \exp(\beta^{(j)} x_i)}$$

for each training data point i. Here $\beta^{(k)}$ are the weights of the model. We see from the form of the softmax that it is the probability of class k_i divided by the probability of all the other classes.

We want to maximize this, or equivalently, minimize its logarithm

$$C(\beta) = -\sum_{i=1}^{N}\sum_{k=1}^{K} 1\{y_i = k\} \log \frac{\exp(\beta^{(k)T} x_i)}{\sum_{j=1}^{K} \exp(\beta^{(j)T} x_i)}$$

which is our cost function. Here $\{$`statement`$\}$ is 1 if `statement` is `True` and 0 otherwise.

To train our model we want to minimize the cost function. To do this we need the gradient, which is given by

$$\nabla_{\beta^k} C(\beta) = -\sum_{i=1}^{N} x_i \left(1\{y_i = k\} - P(y_i = k|x_i, \beta) \right).$$

A method like stochastic gradient descent or newton's method can be used together with this to train the model and find optimal weights β.

4.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a variation of the gradient descent method. There are many different variations of gradient descent but the common denominator is that they work to find the minima of a function (typically the cost function) by iteratively moving in the direction of steepest descent.

In other words, if we want to find the minimum of a function $F(\mathbf{x})$, we should move in the direction of the negative gradient $-\nabla F(\mathbf{x})$.

For a (typically small) stepsize $\gamma_k > 0$ known as the step size, or learning rate we then have

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k),$$

An initial guess is made at the first step.

For a convex cost function and a sufficiently small stepsize γ_k this method can be used to find the global minimum of said cost function.

One drawback of this method is that computing the gradient for large datasets can be very computationally expensive. This is where SGD and its variations comes in. Instead of calculating the gradiant for all datapoints, SGD chooses a subset of the data to calculate the gradient on. The method divides the dataset into a set of N/M so-called mini-batches, and for each step the gradient is calculated on one of these mini-batches. These mini-batches are denoted by B_k where $k = 1, \cdots, N/M$. N is here the size of the training data, and M is the size of each mini-batch.

Rewriting the gradient descent method in terms of a cost function $C(\mathbf{fi})$, we get

$$\mathbf{fi}_{k+1} = \mathbf{fi}_k - \gamma_k \nabla_\beta C(\mathbf{x}_k, \mathbf{fi}_k),$$

Taking the gradient with respect to one mini-batch per step we get

$$\beta_{k+1} = \beta_k - \gamma_k \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \mathbf{fi}_k)$$

which is the stochastic gradient descent method.

In linear regression the cost function is the mean square error. For the gradient of the cost function we then have

$$\nabla_\beta C(\mathbf{fi}) = \frac{2}{n}\Big(X^T(X\beta - y) + \lambda\beta\Big)$$

here I have added an L_2 regularization parameter λ which will be zero for ordinary least squares, but take some non-zero value for ridge. The number of datapoints is given by n.

For logistic regression the cost function is the cross-entropy, which we went over in the section on logistic regression.

4.2.1 Decaying Learning Rate

Gradient methods can be sensitive to the choice of the step size or learning rate γ, and both too small and too large values can give very poor results. One technique to limit this issue is to gradually decrease the learning rate as we move through the epochs. The idea is that we use large steps in the beginning when we are likely to be far away from the solution, and smaller steps as we approach the minimum.

The let the learning rate γ_j *decay* according to the function

$$\gamma_j(t; t_0, t_1) = \frac{t_0}{t + t_1}$$

where $t_0, t_1 > 0$ are fixed parameters, and $t = i \cdot m + b$ with i denoting the current epoch number, m the number of mini-batches, and b the current batch number.

4.3 Neural Network

Neural networks are a group of models originally inspired by biological neuron used in supervised and unsupervised learning as well as specialized tasks such as image processing. They are non-linear models and can be considered powerful extensions of supervised learning methods like linear and logistic regression We will be focusing on a type of neuroal networks called feed-forward neural networks (FFNN).

Figure 3: A schematic showing a fully connected FFNN with one hidden layer.

4.3.1 Feed-Forward Neural Network

A FFNN is a type of artifical neural network (ANN) made up of layers of connected neurons, also called nodes. They consist of an input layer, followed by one or more so-called hidden layers, and finally an output layer. In each layer there is a certain number of nodes, and this number can vary between the layers. The nodes of one layer are connected with the nodes of the next with an associated weight variable. In addition there may be a bias in each layer. Training the model amounts to finding optimal values for these weights and biases.

In FFNN the information flows only forward, from one layer to the next. If all nodes in each layer are connected to all the nodes in the next we have a fully connected network. According to the *universal approximation theorem*[4], a FFNN with just a single hidden layer containing a finite number of neurons can approximate a continuous multidimensional function to arbitrary accuracy. This result assumes that the activation function for the hidden layer is a non-constant, bounded and monotonically-increasing continuous function.[5]

For a fully-connected model each input node sends its input x_j to every node in the first hidden layer. The input to node i of the first hidden layer becomes:

$$z_i^1 = \sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1 \tag{1}$$

Each input is weighted bu w_{ij} and in addition to the sum over the weighted inputs the node receives a bias contribution b_i^1. This bias is to assure we don't end up with zero activation in a layer, as this would stop the flow of information from the input to the output, and give us no output. The ouput from node i in the first hidden layer is

$$y_i^1 = f(z_i^1) = f\left(\sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1\right) \tag{2}$$

where $f(z)$ is the activation function for the hidden layer. The output from the nodes in the hidden layer are given as weighted inputs to all the nodes in the next hidden layer in the same way as described here for this first layer, with a (different or equal) activation function giving the ouput of that next layer. This continues until the output layer. The output from this final layer is the model layer. Typically the nodes in the hidden layers all have the same activation function, while the ouput layer has a different one.

Generalizing we get the output from a model with l hidden layers as:

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^{l+1} f^l \left(\sum_{k=1}^{N_{l-1}} w_{jk}^l \left(\dots f^1\left(\sum_{m=1}^{M} w_{nm}^1 x_m + b_n^1\right)\dots\right) + b_j^l\right) + b_i^{l+1}\right] \tag{3}$$

which is a nested sum of weighted activation functions.

With the biases and activations as $N_l \times 1$ column vectors $\hat{b}_l$ and $\hat{y}_l$, where the i-th element of each vector is the bias b_i^l and activation y_i^l of node i in layer l respectively, and the weights as an $N_{l-1} \times N_l$ matrix, W_l we can write the sum as a matrix-vector multiplication. Looking at hidden layer 2 for simplicity we can write this in matrix notation as

$$\hat{y}_2 = f_2\left(\left[\begin{array}{cccc} w_{11}^2 & w_{12}^2 & \cdots & w_{1N_l}^2 \\ w_{21}^2 & w_{22}^2 & \cdots & \vdots \\ \vdots & \vdots & & \vdots \\ w_{N_{l-1}1}^2 & w_{N_{l-1}2}^2 & \cdots & w_{N_{l-1}N_l}^2 \end{array}\right] \cdot \left[\begin{array}{c} y_1^1 \\ y_2^1 \\ \vdots \\ y_{N_l}^1 \end{array}\right] + \left[\begin{array}{c} b_1^2 \\ b_2^2 \\ \vdots \\ b_{N_l}^2 \end{array}\right]\right). \tag{4}$$

In general we have an expression for z_i^l, the activation of node i of the l-th layer as

$$z_i^l = \sum_{j=1}^{N_{l-1}} w_{ij}^l a_j + b_i^l.$$

Here b_i^l is the bias into node i in layer l, w_{ij}^l is the weight from node j in layer $l-1$ on the input to node i in layer l, and a_j the output from node j in layer $l-1$.

The output from node i in layer l becomes

$$a_i^l = f^l(z_i^l) = \frac{1}{1+\exp-(z_i^l)}.$$

Using the sigmoid as activation function in layer l.

4.4 Regularization

We will be adding an L_2 regularization parameter λ to our logistic regression as well as our neural network code. See description of ridge regression in project 1[1] for more info on L_2 regularization.

4.4.1 L2 Regularization FFNN

when adding L2 regularixation to a feed forward neural network with backpropagation like we have, this amounts to adding a regularization term to the cost function minimizing the size of the individual weights in the model.

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

where C_0 is the original cost function without regularization and λ is the regularization parameter. The regularization does not have a dependency to the biases, so these are not affected, but the derivative of the cost function with respect to the weights becomes

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w.$$

Using this the weights are updated according to

$$w- > w\left(1 - \frac{\gamma\lambda}{n}\right) - \gamma \frac{\partial C_0}{\partial w}$$

4.4.2 Activation Function

A choice must be made for the activation function to be usen in the nodes of the hidden layer as well as the nodes of the ouput layer. I will be using the sigmoid function as well ass the ReLU and Leaky ReLU in the hidden layers, and the softmax for the output layer when classifying handwritten numbers. For regression with FFNN I will use ReLU as the activation function for the output layer.

Sigmoid Function This activation function is popularly used in the ouput layer of binary classification problems. This function is at risk for the so-called vanishing gradient problem; that the gradient becomes too small for effective model training. the function has the form

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

A plot is seen in figure 4.

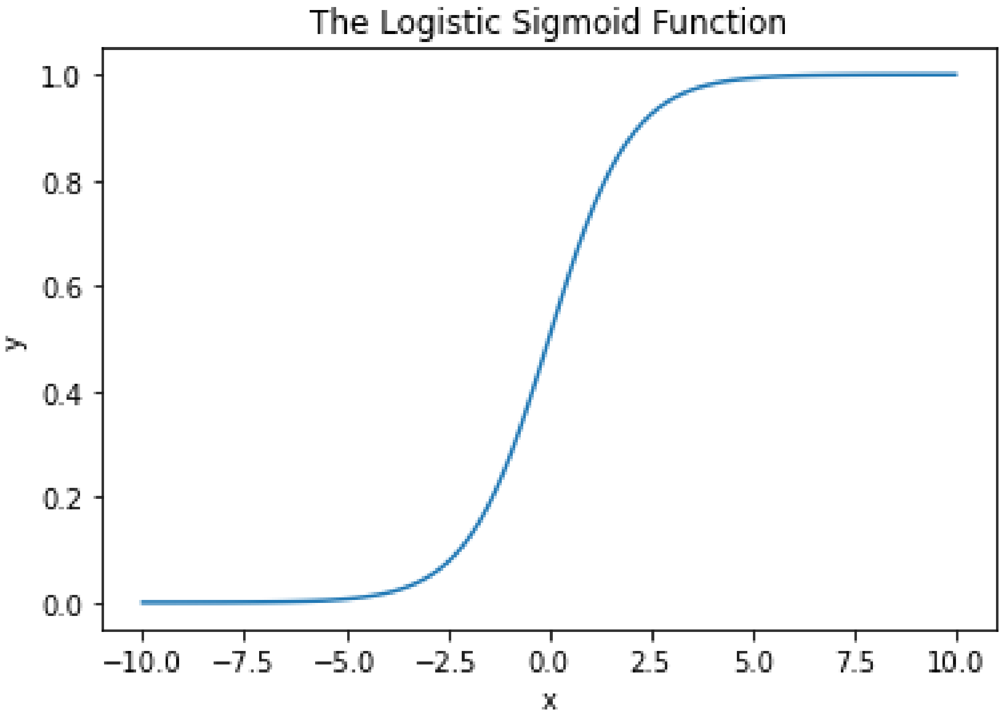

Figure 4: Plot of the logistic sigmoid function

Hyperbolic Tangent Function A mathematically shifted version of the sigmoid that generally performs better than the sigmoid. It has the form

$$f(x) = \tanh(x),$$

and a plot is seen in figure 5.

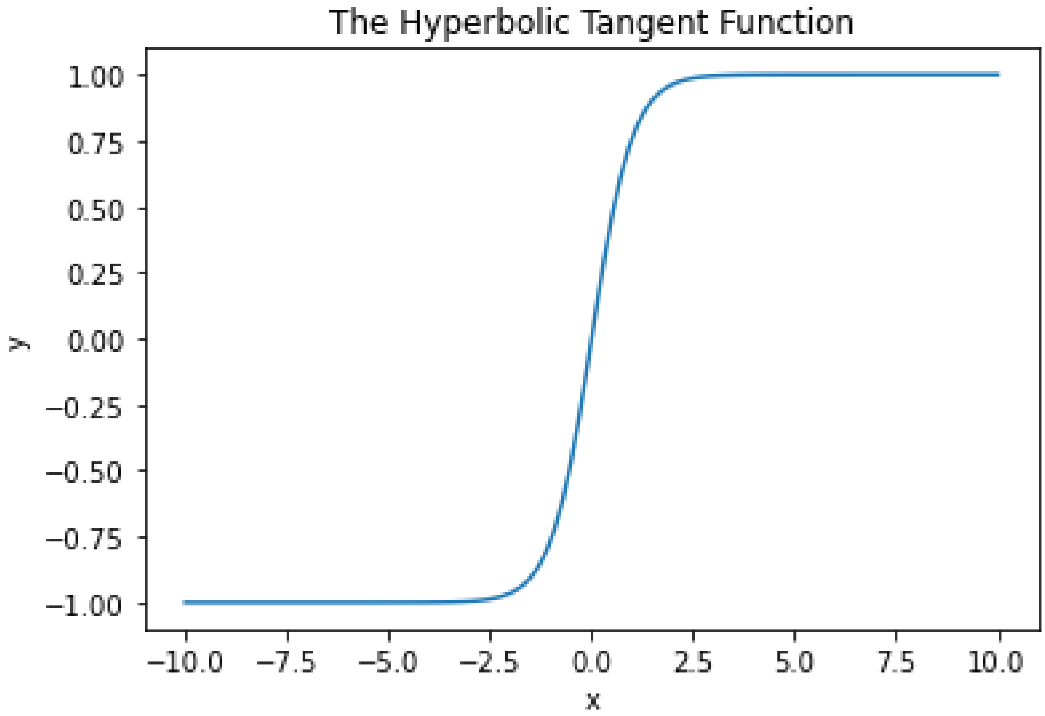

Figure 5: Plot of the hyperbolic tangent function

Binary Step Function Also called the heaviside step function, or the unit step function. It is an "all or nothing" approach and jumps from 0 to 1 at $x = 0$.

$$f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

A plot is seen in figure 6.

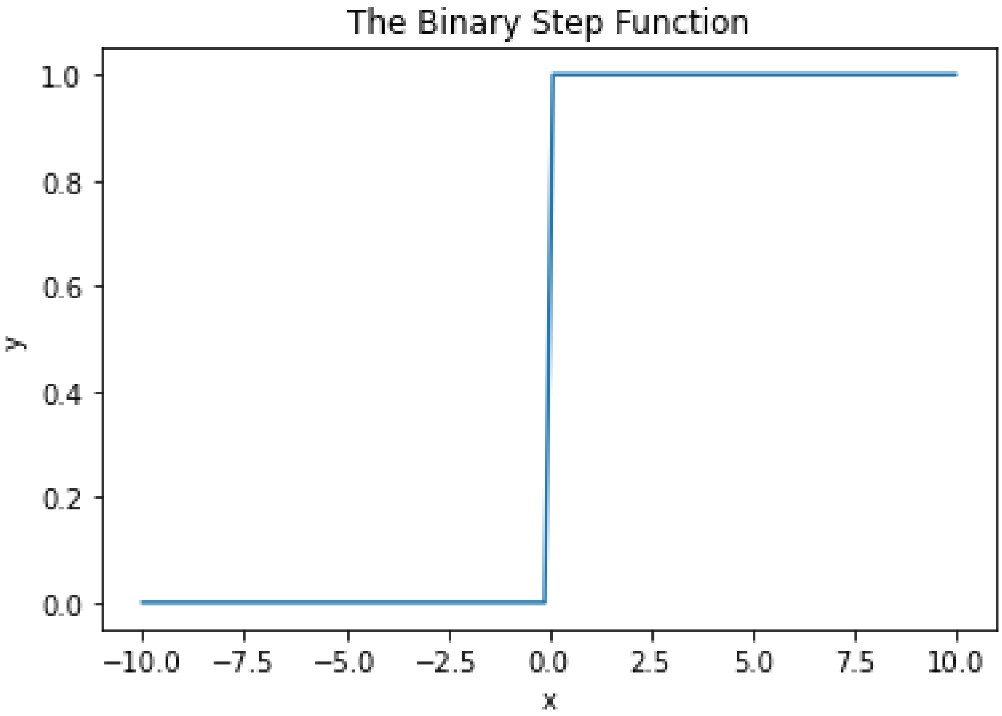

Figure 6: Plot of the binary step function

ReLU Function ReLU, or Rectified Linear Unit is a popular and fairly general activation function. It is a common choice for the activation function of the hidden layers. It is computationally cheap and therefore efficient. One issue is that all negative values are mapped to zero, which can lead to what is called *dying ReLU problem*. This is an issue that can occur if the weighted sum of a neuron's inputs is negative. The neuron will then produce 0 as its output. Because the gradient of the ReLU function is 0 when its input is negative this is unlikely to change and the neuron is essentially "dead".

The rectifier is defined as the positive part of its argument

$$f(x) = x^+ = \max(0, x).$$

A plot is seen in figure 7.

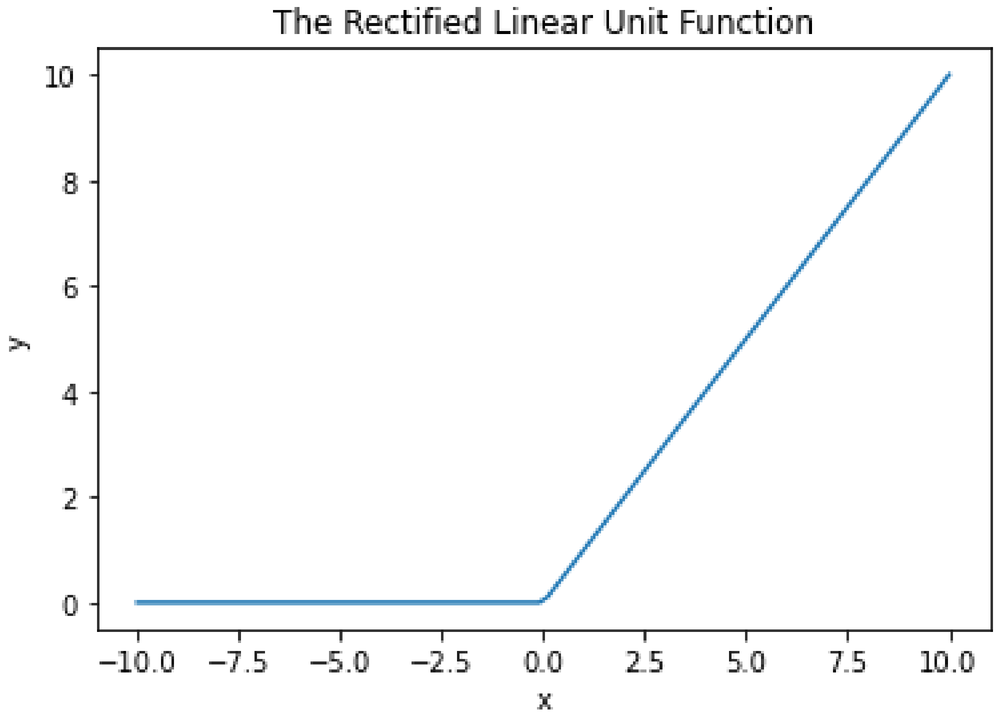

Figure 7: Plot of the ReLU function

Leaky ReLU Function Leaky ReLU solves the dying ReLU problem described in the last subsection by assigning a small positive slope for $x < 0$. This does however reduce the performance/computational cost.

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01x & \text{otherwise} \end{cases}$$

See figure 8 for a plot of this function.

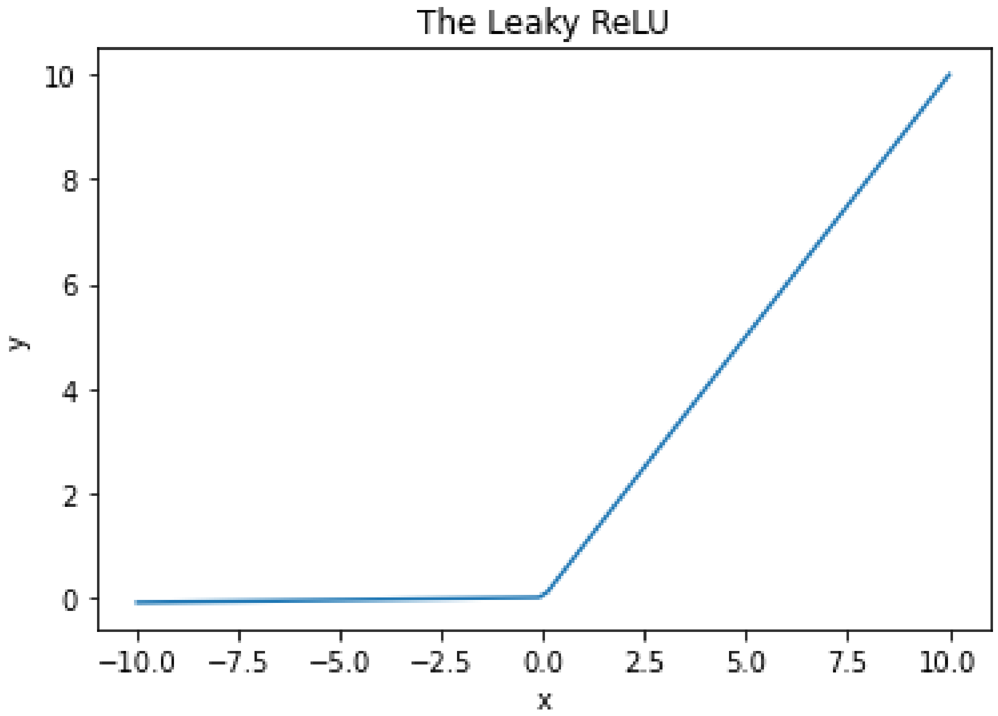

Figure 8: Plot of the Leaky ReLU function. It is a bit hard to spot, but the slope from x=-10 to x=0 is positive.

ELU Function The ELU or exponential linear unit function is another function in the ReLU family proposed to avoid the *dying ReLU problem*. It does however add another parameter to the model α.

$$f(x) = \begin{cases} \alpha(\exp(x) - 1 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

A plot is seen in figure 9.

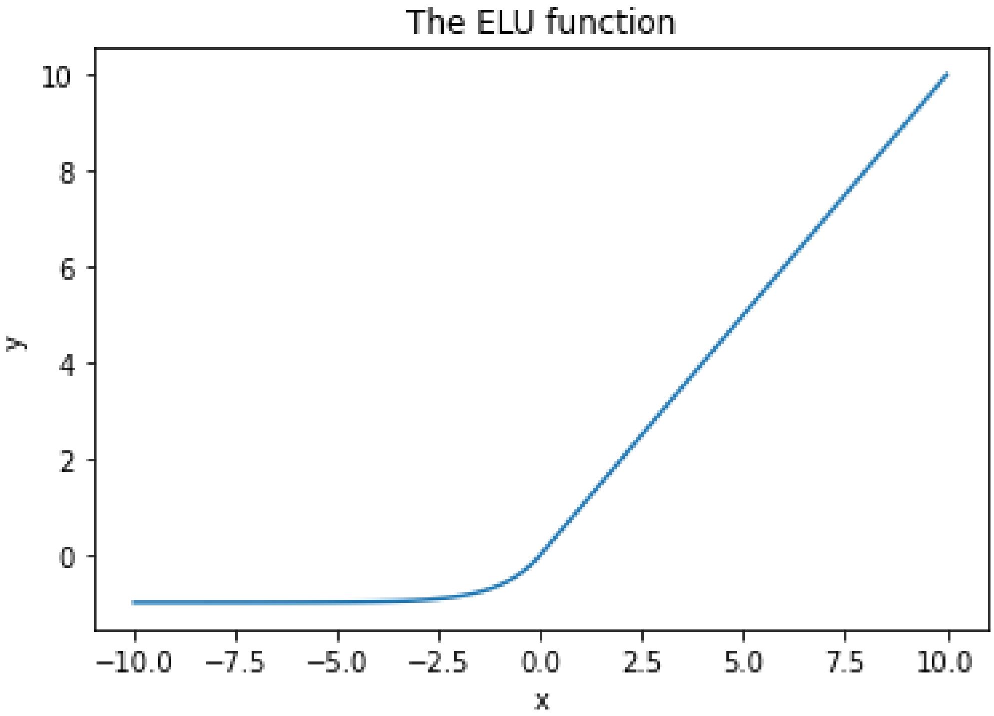

Figure 9: Plot of the ELU function. I have set the paramater alpha equal to 1.

Softmax Function A generalization of the logistic function often used in multinomial logistic regression.

$$f(x) = \frac{e^x}{\sum_{j=1}^{K} e^x},$$

The function takes as input a vector x of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers. See figure 10 for a plot.

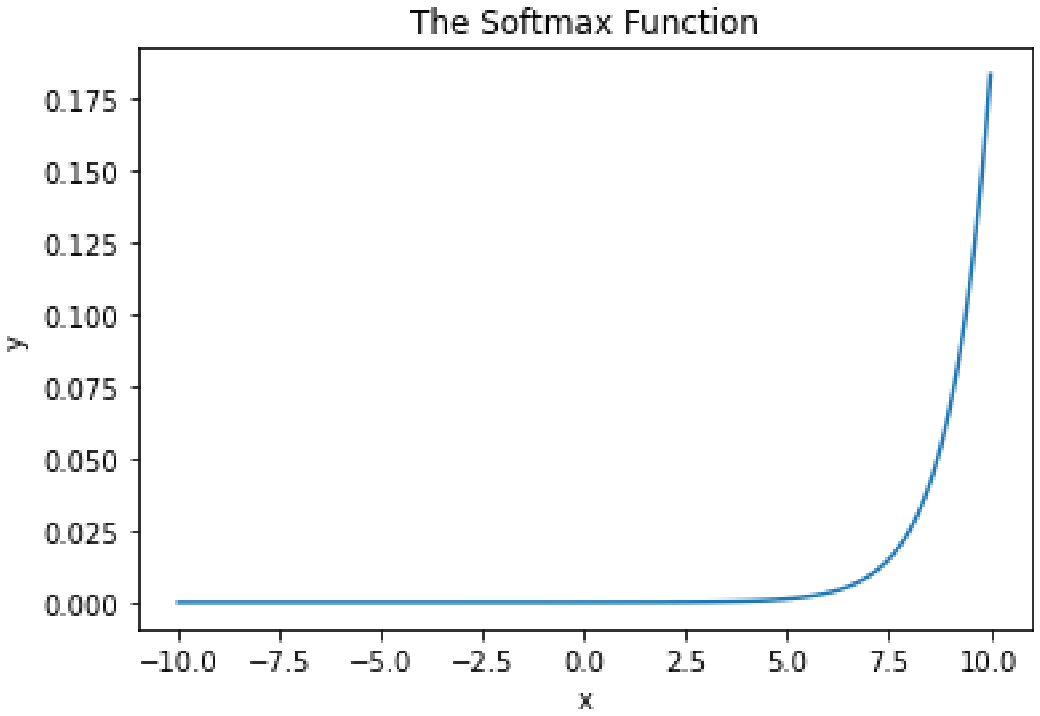

Figure 10: Plot of the softmax function.

5 Results and Discussion

5.1 Linear Regression with Gradient Descent

In project one we looked modeled the Franke function using ordinary least squares (OLS) and ridge regression. Now we want to compare these methods to methods using stochastic gradient descent to obtain the fit, i.e the parameters or weights β.

The comparison is done on data generated with the Franke function, with added stochastic noise with $\sigma^2 = 0.1$. See figure 11 for a 3d plot of these data.

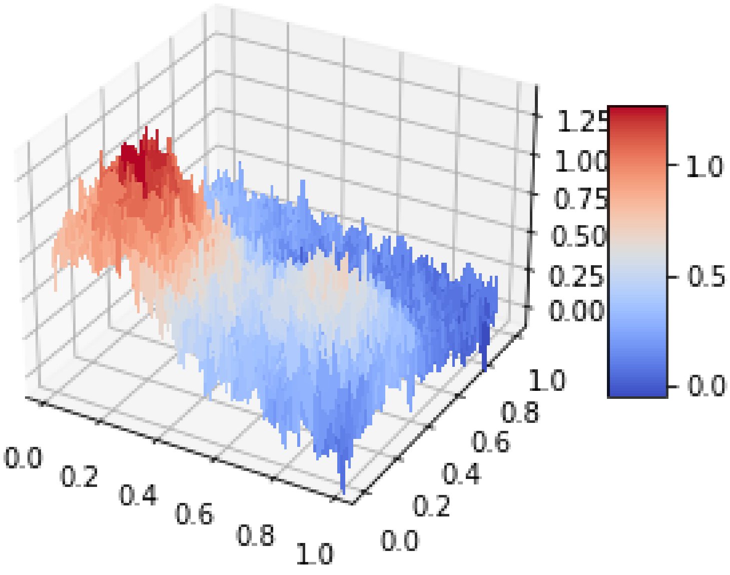

Figure 11: 3D plot of the Franke function with added noise.

```
** Comparing my ols implementation with that of scikit-learn: **
The arrays are about the same:  True
The mean squared error between my result and that of scikit-learn is:
2.0601779122756744e-22
```

We fit our regression models to this data. For OLS there are no parameters to tune. We find the mean square error and the r^2 score of our OLS method, see table 1.

Table 1: Error measures fitting OLS method to noisy Franke data.

```
*** Error measures for OLS method: ***
Train:   MSE:  0.012086801248348334   r2:  0.8644398531166579
Test:    MSE:  0.012501297568858788   r2:  0.8583664210243516
```

For linear regression with gradient descent we have several parameters, the learning rate, the batch size, and the number of epochs. We explore how the error is affected by various choices for these parameters. The results for learning rate can be seen in figure 12. We can see that the error improves steadily, although somewhat erraticaly as the learning rate increases before exploding when we reach a too large learning rate. Too large is seen to be about $\gamma > 0.07$.

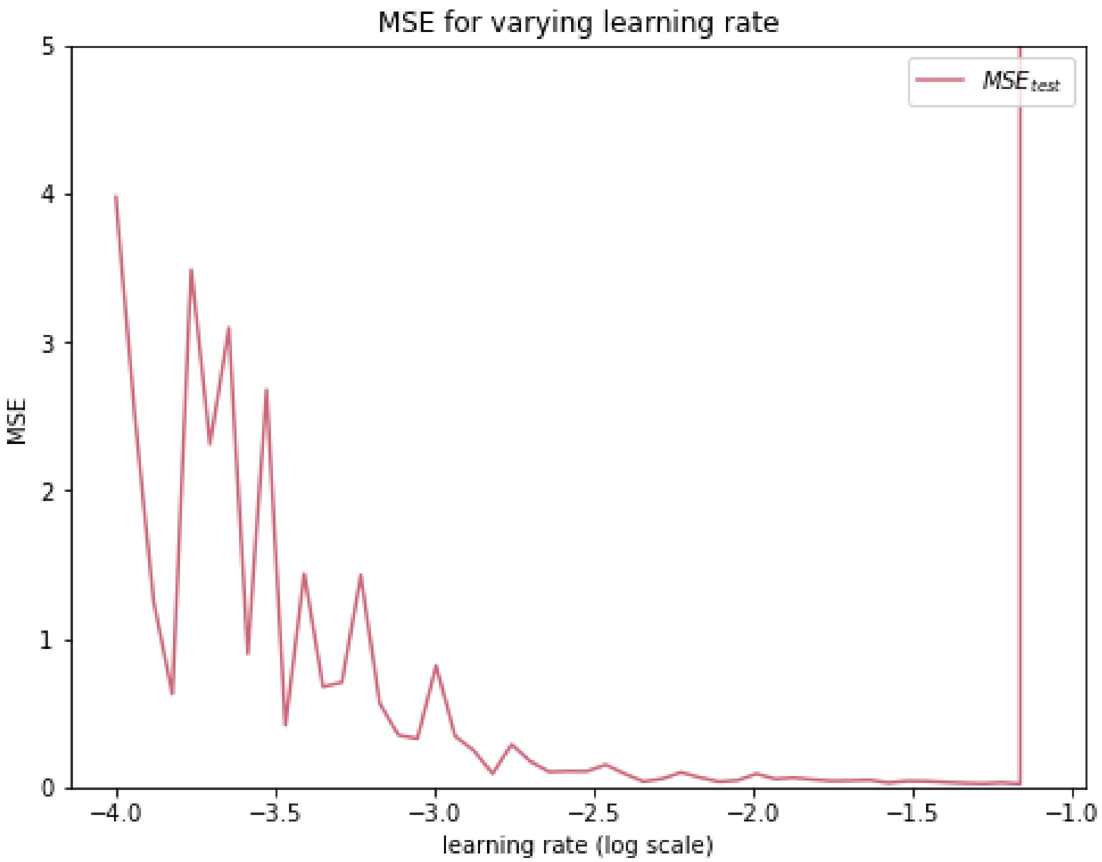

Figure 12: The mean square error on the test set as the learning rate of the SGD method is increased.
No regularization.

The increased error for very small learning rate can be explained by the SGD method taking such small steps down the slope it does not reach a minimum, whereas with a too large learning rate the minimum is skipped altogether.

We can let the learning rate vary according to a learning schedule as described in the methods section on stochastic gradient descent. The results are plotted in figure 13.

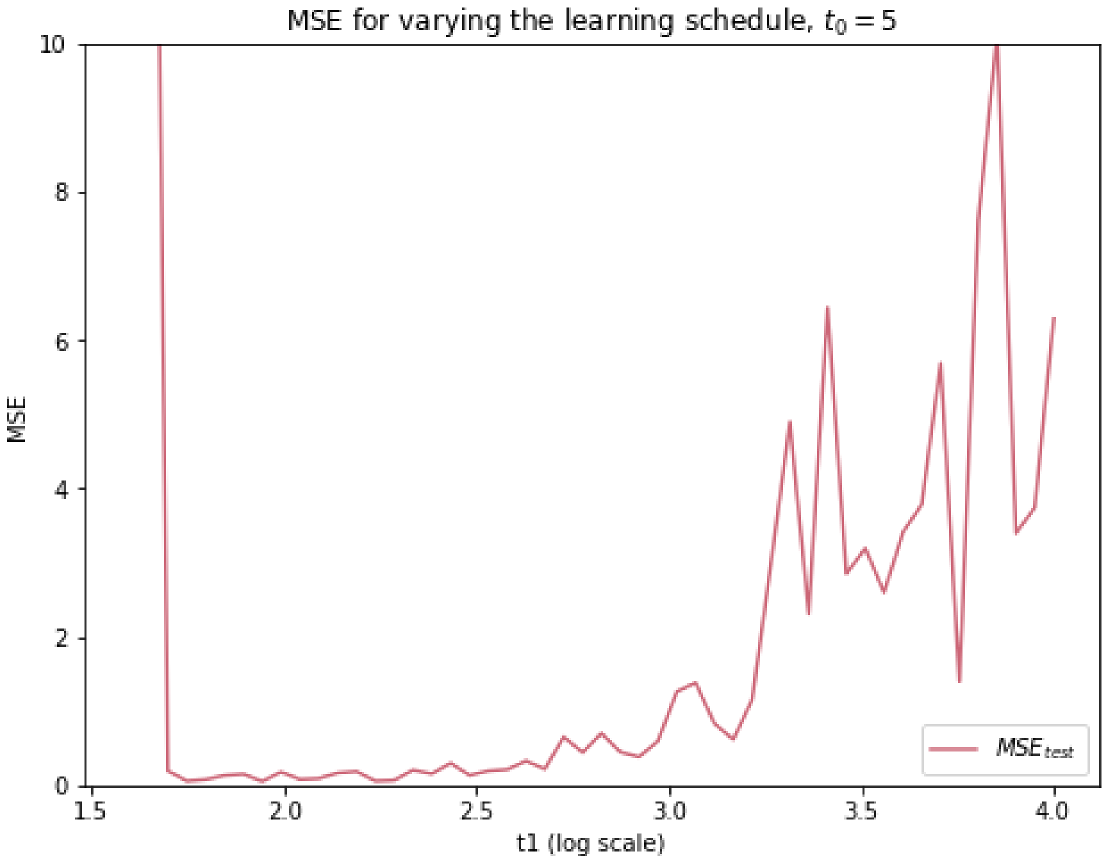

Figure 13: The mean square error for varying learning schedule.

We see that a too small value for `t1` leads to the error exploding, but the error also startes rising for large t1. The sweet spot appears to be around $50 < \texttt{t1} < 200$.

We also want to explore how the error is affected by the number of epochs used. See figure 14 for a plot. As expected the error decreases as the number of epochs increases. At first there is a steep improvement, but as we reach a few hundred epochs the rate of improvement has slowed down markedly and we gain very little relative to the increase in computational cost. Although there is some benefit to increasing to 1000+ epochs, whether that is worth it or not will depend on computation cost and precision requirements.

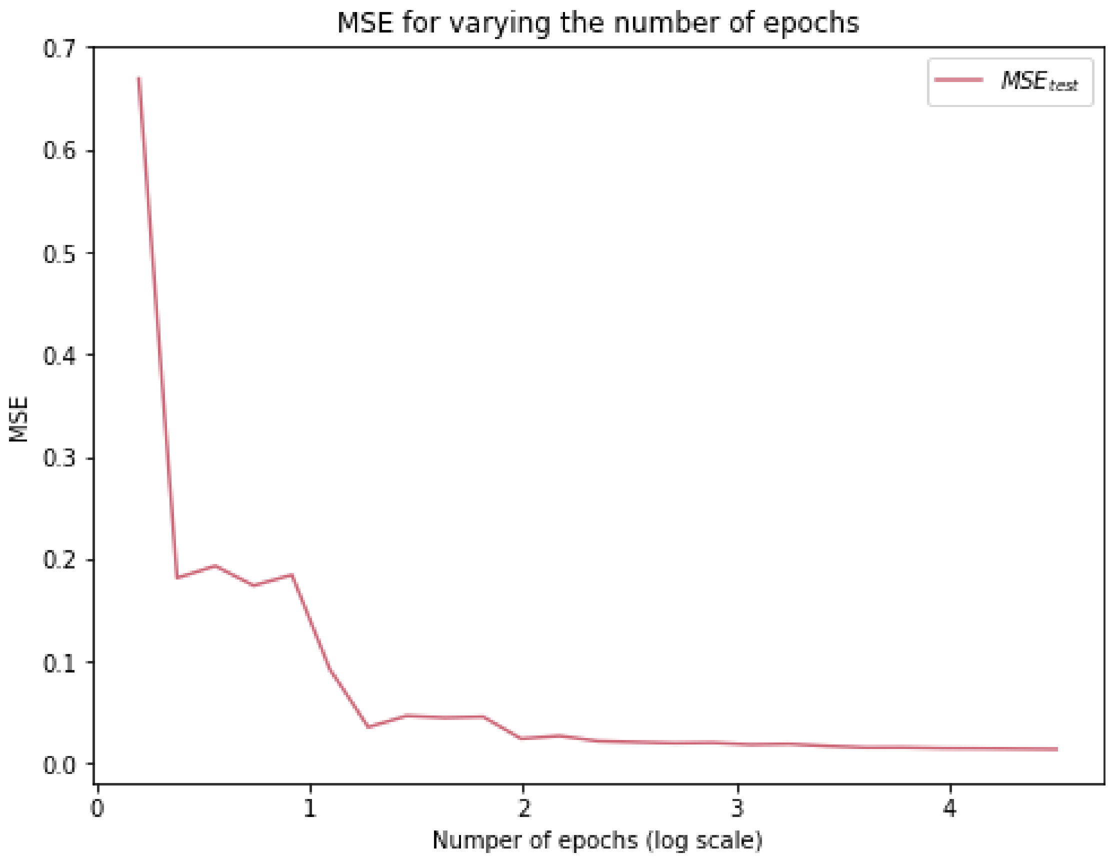

Figure 14: The mean square error for increasing number of epochs.

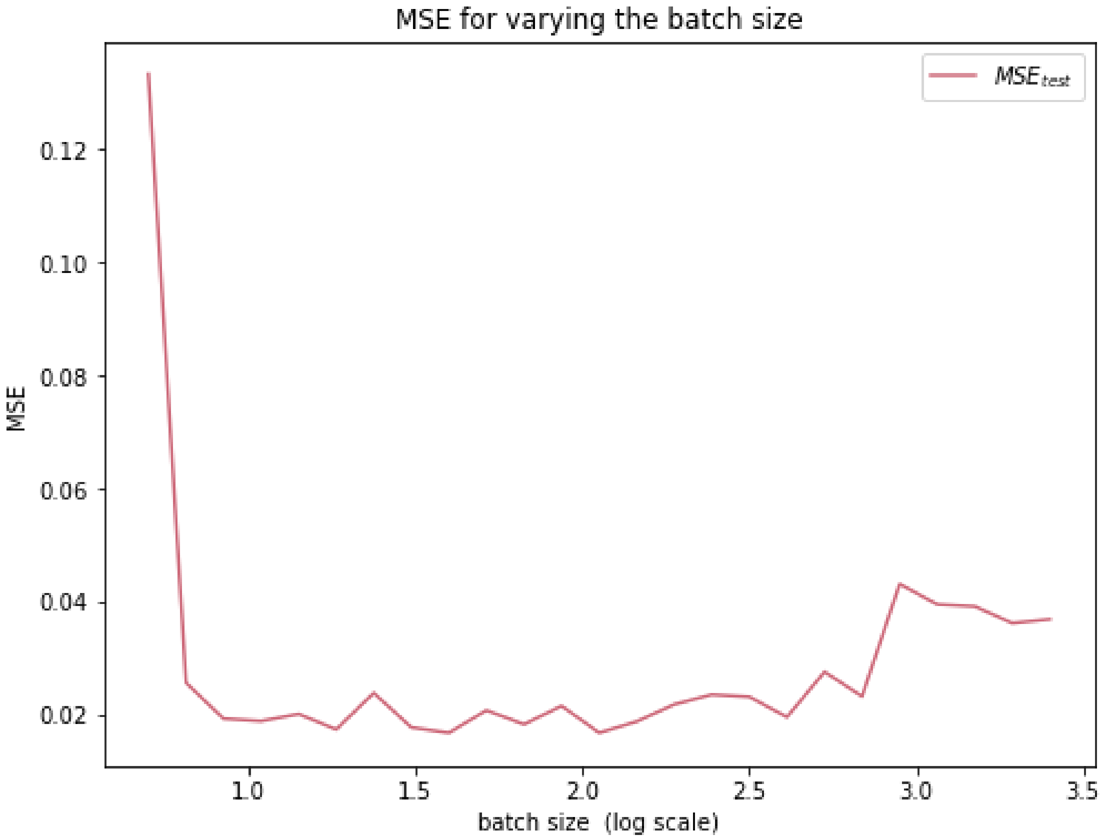

Figure 15: The mean square error for increasing the batch size.

A similar plot for the choice of batchsize can be seen in figure 15. As we see the error explodes for small batch size, but for any batchsize > 7 or so the performance is fairly stable at a minimum. As the batchsize grows very big the error starts to climb up, but even at about 1/3 of the size of the training set the error is small. The reason the error starts increasing here is that the number of epochs is constant, so with very large batchsize we perform fewer steps per epoch and the increased precision of the gradient calculation does not make up for this reduction in steps. For very small batchsizes the estimate for the gradient is simply too inaccurate, and we are not guaranteed to be moving down the gradient slope. We see that the error increase is slight for the batchsized explored, but one should be careful to choose a very big batchsize.

We move on to look at ridge regression with gradient descent. I will be using k-fold cross validation to explore choices for the regression parameter λ and how this choice affects model performance. Figure 16 shows a heatmap of the results.

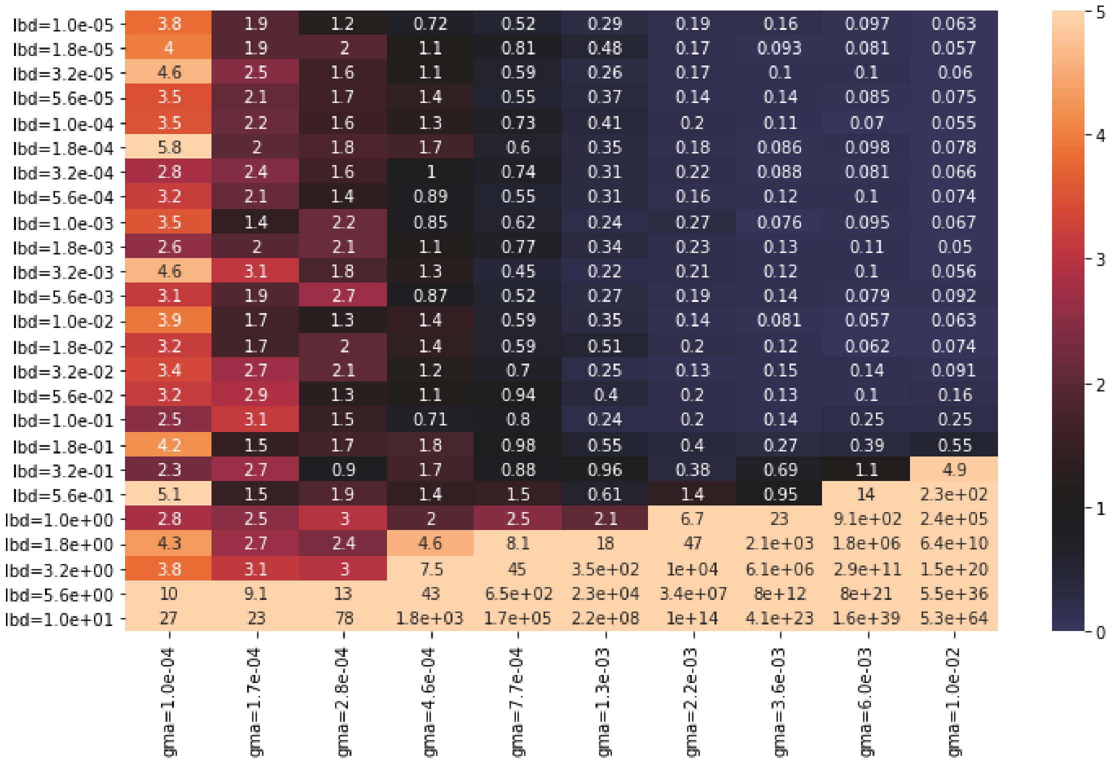

Figure 16: A heatmap to illustrate how the mean square error of the ridge regression model with stochastic gradient descent depends on the choice of ridge parameter lambda and learning rate gamma.

We see from figure 16 that for a small enough regression parameter λ, the MSE is fairly robust to the choice of the learning rate γ. As λ grows however, the range of γ-values giving a small MSE narrows.

Now that we have found some optimal parameters for SGD on this particular problem, let us compare our model performance with that of scikit-learn's `SGDRegressor`. A table of the results is seen in table 2.

Table 2: Comparing my own ridge regression model with SGD with `SGDRegressor` from `scikit-learn`.

```
SGDRegressor: mse=0.0308    r2=0.6509
Own ridge SGD: mse=0.0164    r2=0.8146
```

I see the error is actually better for my implementation using these parameter, although this method is noticeably slower. The improved error and at least part of the speed improvement is likely due to `SGDRegressor` calculating the gradient on only one datapoint at a time, instead of a batch[3]. This gives a noisier/more erratic movement towards the minimum.

5.1.1 Neural Network - Regression

The second of the overarching goals in this project was to use a feed forward neural network with backpropagation (FFNN) to model the Franke function. We have used the sigmoid function as activation function for the hidden layers, while the output layer has no activation function as I want the actual, continuous values, i.e. not translated into a binary yes/no value or a set of classes. Cross-entropy is used as cost function.

The network is set up with a number of input nodes matching the number of terms in the design matrix `X`, and just one output node as we only have one output, z.

I try out a few different configurations for the hidden layer(s). Figure 17 shows a plot of the resulting r^2 scores, all using 100 epochs and a batch size of 50. The configurations correspond to the following set up:

- config1 = [21,1]
- config2 = [21,5,1]
- config3 =[21,10,1]
- config4 = [21,5,2,1]
- config5 = [21,5,5,1]
- config6 = [21,10,5,1]
- config7 = [21,10,10,1]
- config8 = [21,15,10,5,1]

which shows a list of the number of nodes per layer from input layer, via all the hidden layers, ending with the output layer. The model is tested on a set of learning rates, but the learning rate where the error explodes is different for the various configurations. I have plotted only the sensible values, so when the lines in figure 17 stop that means their error exploded for the next value of the learning rate γ.

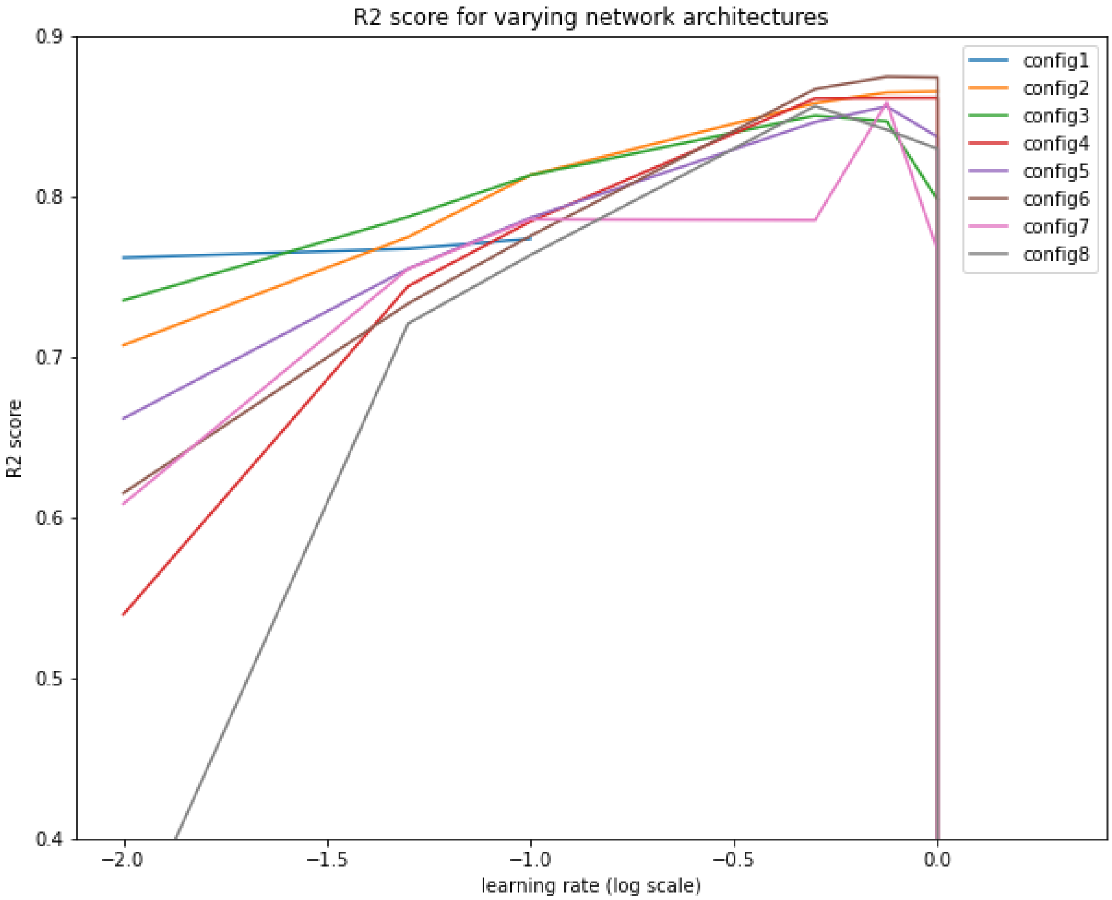

Figure 17: R^2 score on test set for a selection of architectures for the neural network. For all models the logistic sigmoid is used in the hidden layers, cross entropy is the cost function. The models are fitted over 100 epochs with a batchsize of 50.

Of the network architectures tried, the optimal seems to be configuration number 6 with two hidden layers, one with 10 nodes and one with 5. We will use this configuration from here on.

We do, however, see that performance is fairly stable for the majority of the tested configurations. Configuration 1 (no hidden layers) seems to be particularly sensitive to the choice of learning rate, but apart from this they are all in agreement on the optimal range for the learning rate. In addition the optimal choice of the learning parameter gives fairly similar values for the r^2 score. It is worth noting that the parameters such as number of epochs and batch size has not been exhaustively explored for all the configurations shown. As such some are likely to have even better 'optimal' r^2 scores for different hyper-parameters.

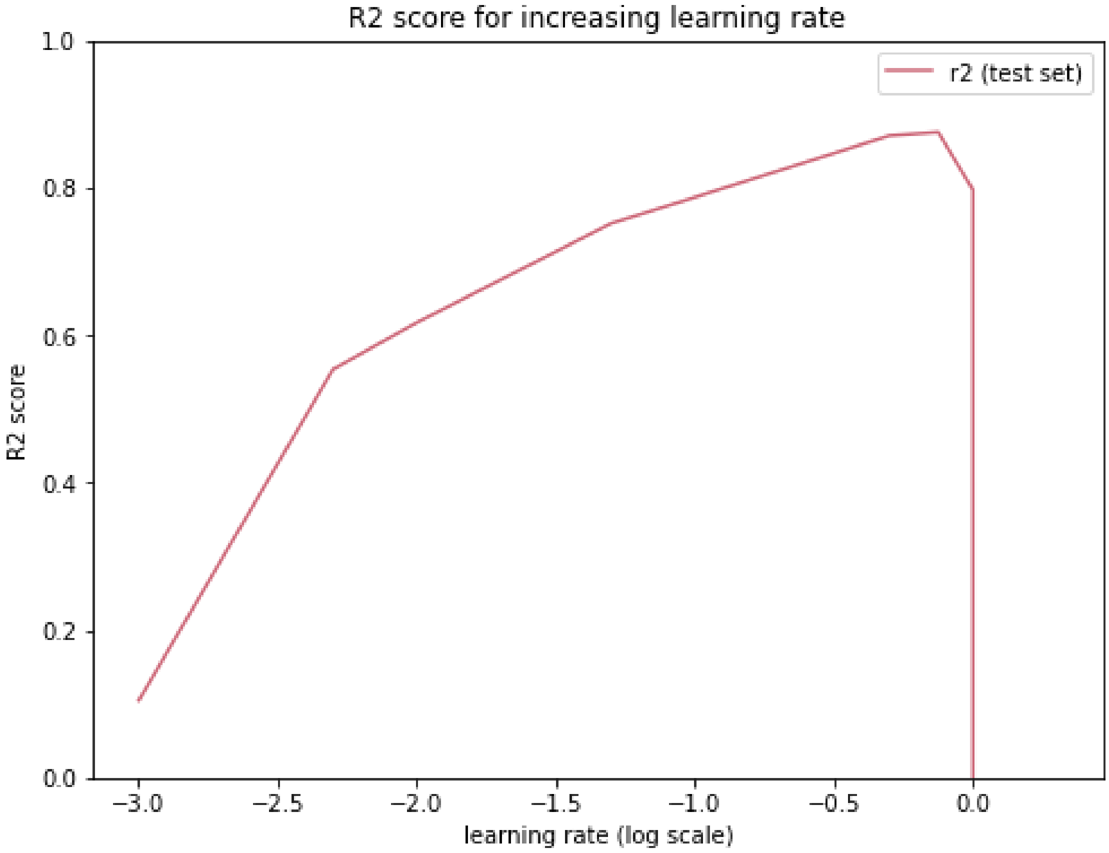

Figure 18: How the performance of the FFNN regression model on the franke data changes with the learning rate. One hidden layer with 10 nodes, and one with 5. The batchsize is 50 and the number of epochs is 100. The hidden layer uses the sigmoid activation function and the output layer uses no activation function.

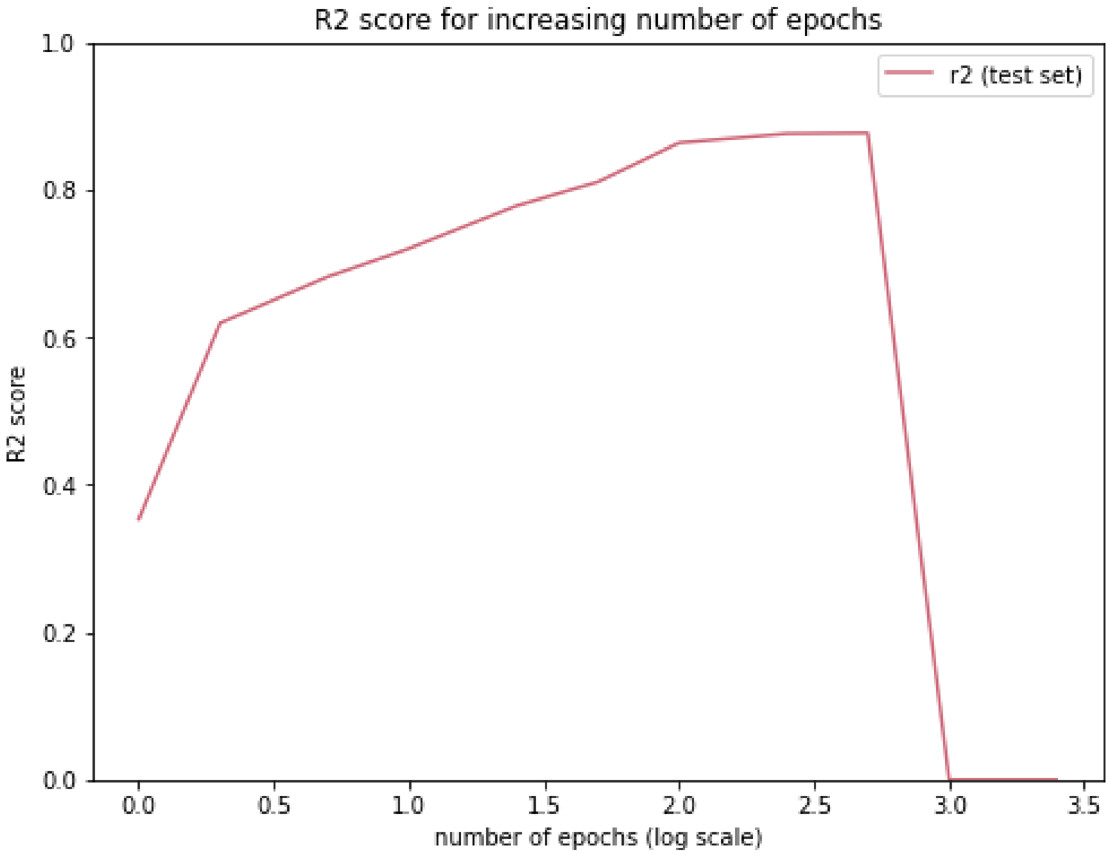

Figure 18: How the performance of the FFNN regression model on the franke data changes with the number of epochs. One hidden layer with 10 nodes, one with 5. The batchsize is 100 and the learning rate is 0.8. The hidden layer uses the sigmoid activation function and the output layer uses no activation function.

Figure 18 shows the r^2 score as a function of the number of epochs using configuration 6. We see the r^2 score improves with increasing the number, as expected, but from 100 epochs and on it flattens out and no further improvement is seen, meaning increasing the number above this will increase the computational cost at minimal to no additional gain in performance.

Next up is the batch size, as seen in the plot in figure 19. Even for small batch sizes the performance here is good, we don't see the same sensitivity we saw using linear regression with SGD, but we get a similar dip for very large batch sizes. Note that the training set has 7500 entries, so a batch size of 500 (the first data point where we see a reduction in the r2 score) is about 7% of the total training set.

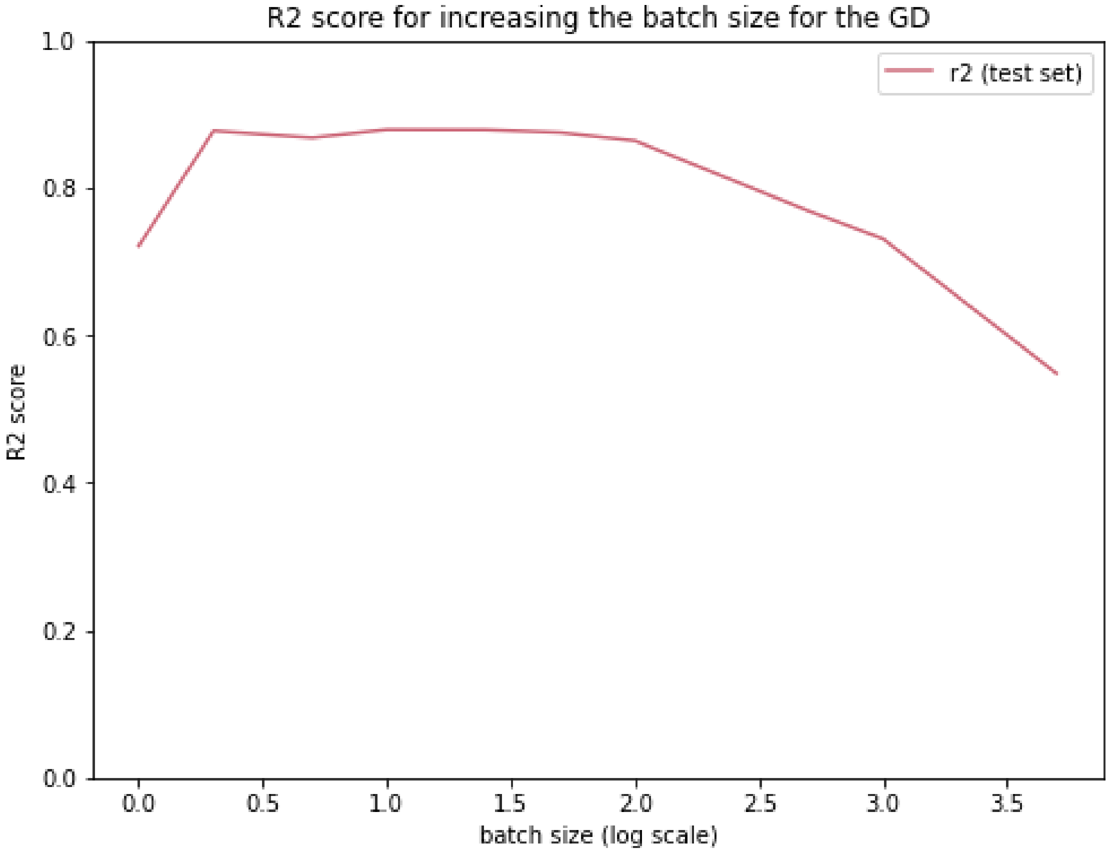

Figure 19: How the performance of the FFNN regression model on the franke data changes with the size of the batches used in the stochastic gradient descent. One hidden layer with 10 nodes, one with 5. The number of epochs is 100 and the learning rate is 0.8. The hidden layer uses the sigmoid activation function and the output layer uses no activation function.

Adding L2 Regularization We now explore adding regularization to the method, namely l2 regularization. First we look at the choice of the regularization parameter λ searching for an optimized value. Figure 21 shows a plot.

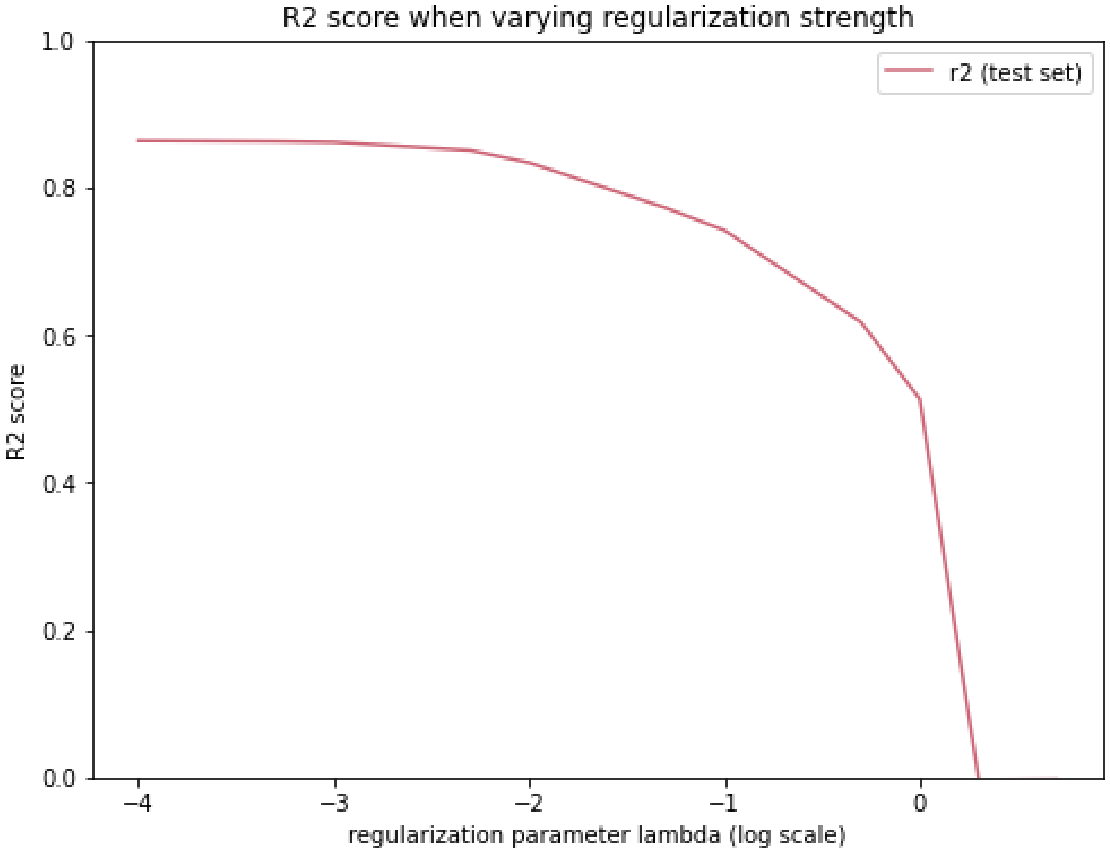

Figure 21: How the performance of the FFNN regression model on the franke data changes with the strength/weight of the l2 regularization. Model with one hidden layer with 10 nodes, one with 5. The number of epochs is 100 and the learning rate is 0.8. The hidden layer uses the sigmoid activation function and the output layer uses none.

Different Activation Functions There are several available options for the activation function of the nodes in the hidden layers. Figure 22 shows a plot of the r2 score for models when the shape of this activation function is altered. Again the error explodes at different values for the learning rate for the different activation functions, and we have avoided plotting the exploding scores, which is why the plots are over differing ranges of the learning rate. We see from figure 22 that ReLU, ELU and the sigmoid all give similar performance, while leaky ReLU seems to struggle more with the learning rate. The sigmoid gives the best r^2 score for the chosen parameters, but ReLU is seen to be somewhat more robust to the choice of learning rate.

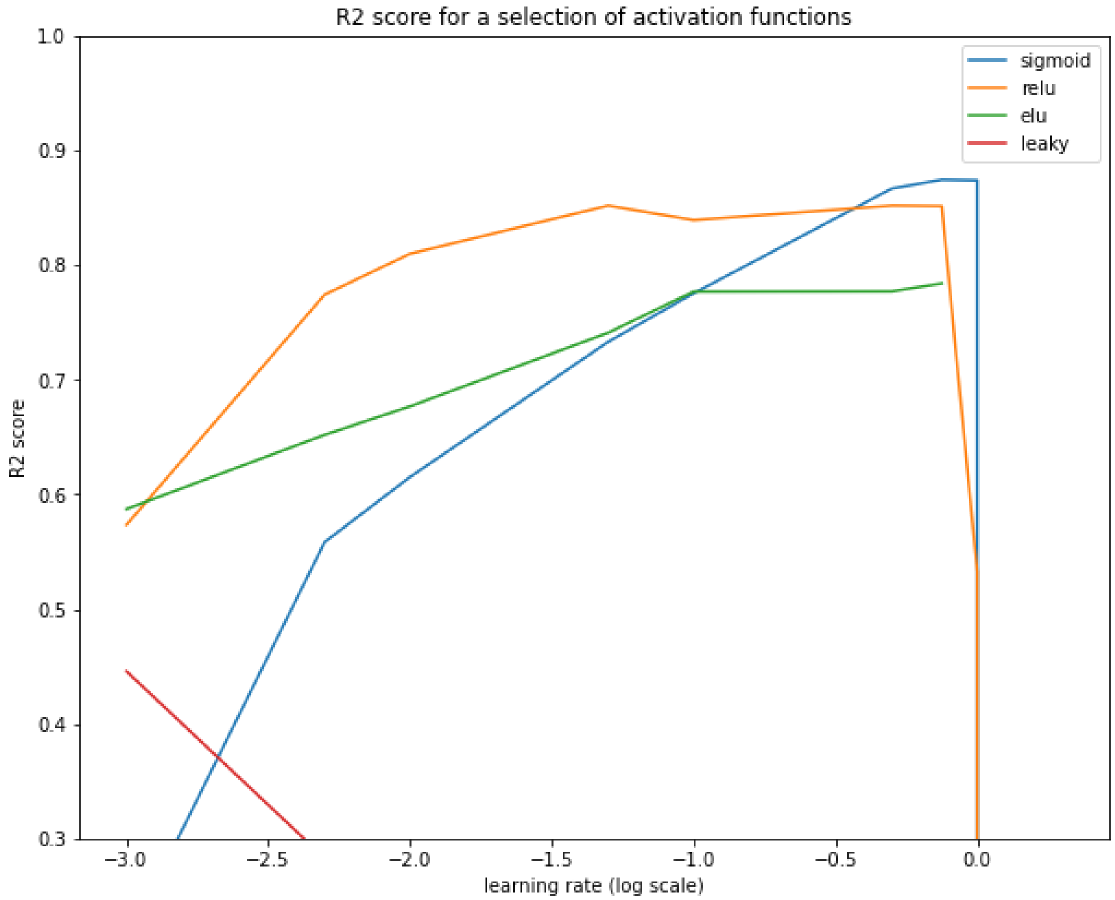

Figure 22: R2 score on test set for a selection of activation functions used in the hidden layers of the neural network. For all models cross entropy is the cost function and the network architecture is as in config6. The models are fitted over 100 epochs with a batch size of 50

Now that we have optimized our parameters to an extent, we compare the performance of this model with that of the `MLPRegressor` in `scikit-learn`. The results can be seen in table 3. For both models we used a learning rate of 0.75, a batch size of 50, and 100 epochs. We see that the resulting errors are almost identical.

Table 3: Comparing my own FFNN regression model with `MLPRegressor` from `scikit-learn`. Both use with SGD and the logistic sigmoid function as activation function for the hidden layers.

```
Own FFNN Regressor with SGD:                mse=0.0110    r2=0.8752
MLPRegressor from scikit-learn with SGD:    mse=0.0116    r2=0.8691
```

5.1.2 Neural Network - Classification

We move on to look at using our neural network for classification. We use the FFNN model to perform classification of handwritten numbers. First we classify the numbers using `scikit-learn`'s `MLPClassifier` to have something to compare the next results to. The network is set up with 64 input nodes, corresponding to the 64 pixels in each digit image, one layer of 30 hidden nodes, and finally an output layer of 10 nodes, corresponding to the 10 output possibilities. The activation function of the output layer is the softmax, while the hidden layer(s) use the sigmoid. We are not using regularization here. Note that the `MLPClassifier` implicitly designs the input and output layers based on the provided data in `fit()` method. It uses cross-entropy as the default cost function.

```
The measured accuracy of FFNN on the test set using scikit-learn's MLPClassifier
is 0.9778
```

We get a very impressive accuracy score of ≈ 0.98 using `scikit-learn`. We compare this with our own code which gives a very similar result of ≈ 0.99 using 30 epochs and a learning rate of 0.5.

```
The measured accuracy of FFNN on the test set using our nown NeuralNetClassifier
with one hidden layer of 30 nodes, softmax activation function in the output
layer and sigmoid in the hidden layer is: 0.9889
```

Like for regression we are interested in exploring a few different configuartions or network architectures. Figure 23 shows a plot of how the resulting accuracy for a selection of configurations varies with the choice of learning rate. The tested configurations are:

- config1 = [64,10]
- config2 = [64,10,10]
- config3 = [64,30,10]
- config4 = [64,10,10,10]
- config5 = [64,30,30,10]
- config6 = [64,50,40,30,10]
- config7 = [64,50,40,30,20,10]

We see from the plot that for all the configuration the general behavior is the same. The performance improves with the increasing learning rate up to a point. The differences relate to how high the accuracy gets, and how soon it responds to the increase in learning rate. The drop off learning rate is similar for all the architectures. Even configuration 1 performs really well, despite not having any hidden layers. It is however the most sensitive to too high learning rate. In contrast it seemingly handles small learning rates better than the others. Based on figure 23 configuration 3 seems a solid choice, but any of configuration 1, 2, 3 and 5 display a satisfactory performance and stability.

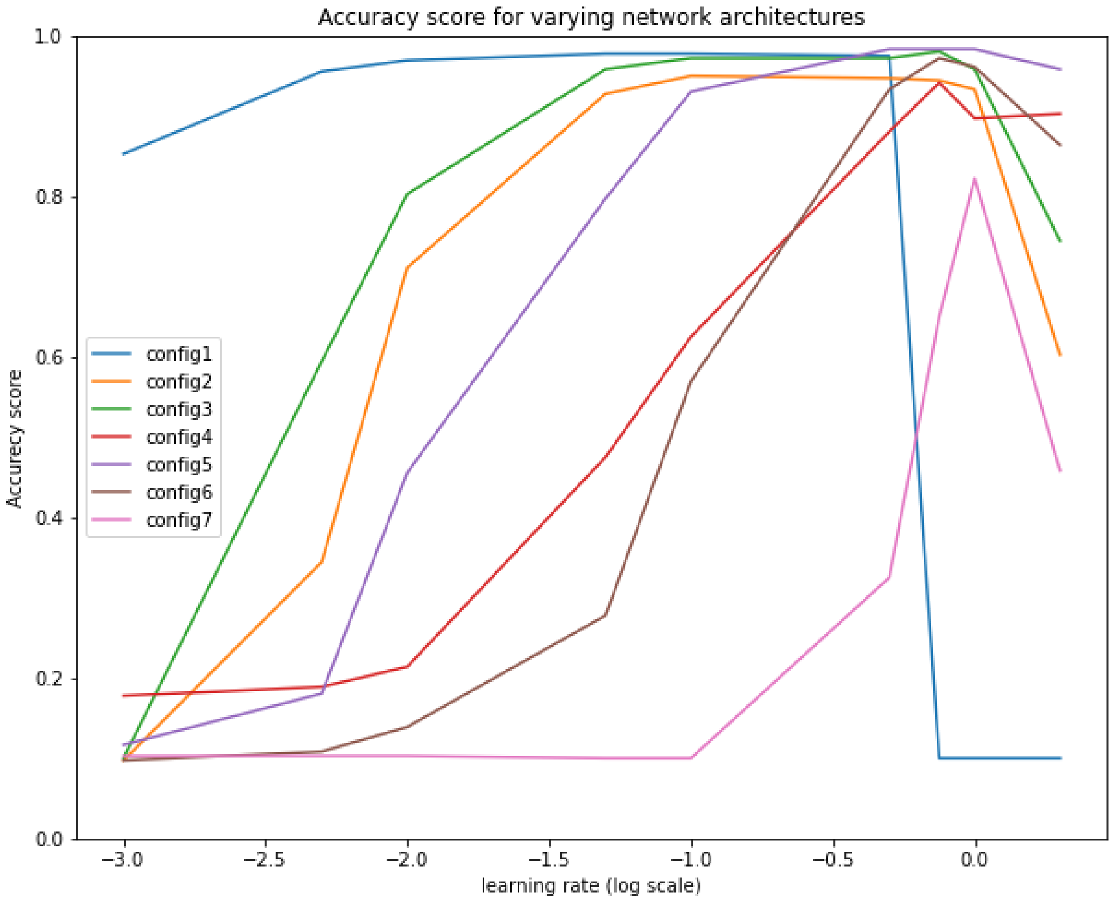

Figure 23: A plot showing how the model accuracy of our FFNN model for classification when identifying handwritten digits for a selection of network architectures. Batch size is 100 and the number of epochs is 30.

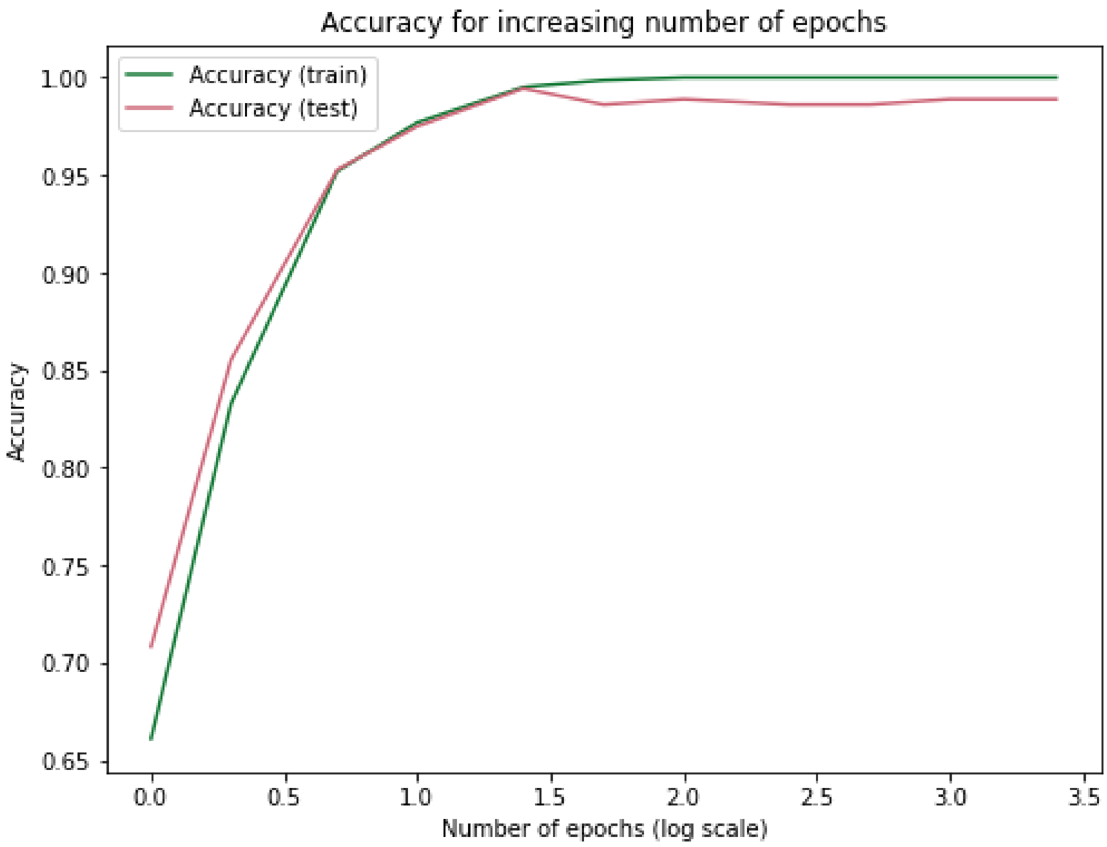

Figure 24: How accuracy of the FFNN classification model changes with the number of epochs. One hidden layer with 30 nodes, batchsize is 100 and the learning rate is 0.5. The hidden layer uses the sigmoid activation function and the output layer uses the softmax.

The max accuracy on the test set is 0.994444 after 25 epochs.

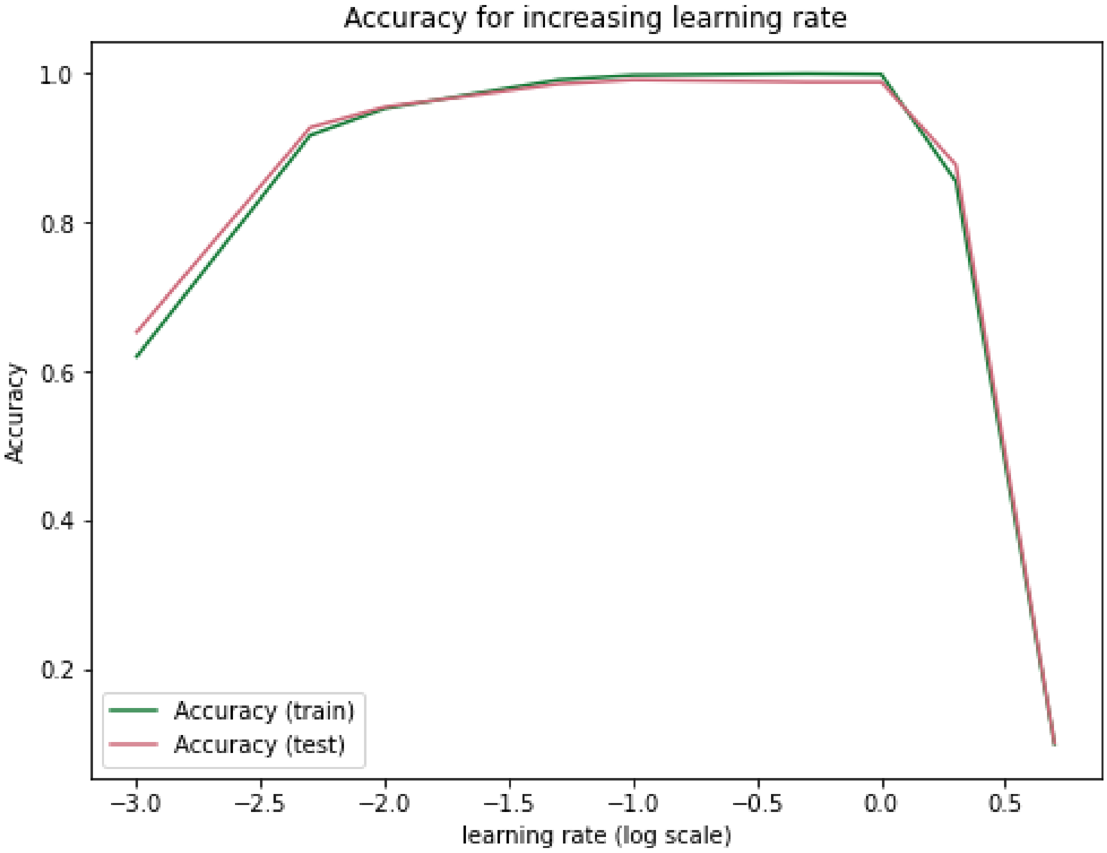

Figure 25: How accuracy of the FFNN classification model changes with the learning rate. One hidden layer with 30 nodes, batchsize is 100 and the number of epochs is 100. The hidden layer uses the sigmoid activation function and the output layer uses the softmax.

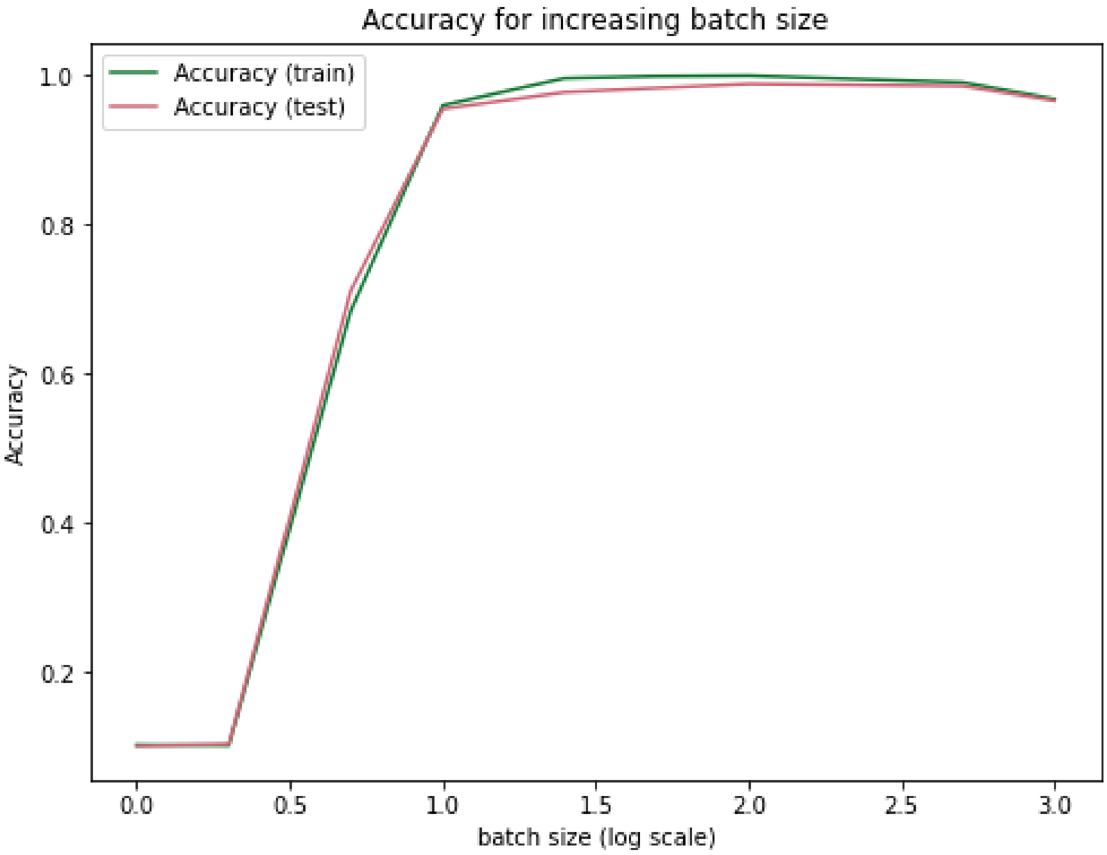

Figure 26: How accuracy of the FFNN classification model changes with the batch size. One hidden layer with 30 nodes, larning rate is 0.5 and the number of epochs is 100. The hidden layer uses the sigmoid activation function and the output layer uses the softmax.

Adding Regularization To avoid over-fitting and increase the generalization of a model one can add a regularization term. As we have seen in the results so far we don't appear too troubled by over-fitting in our classification model with our current parameters. Adding regularization can also make the model more stable to changes in these parameters.

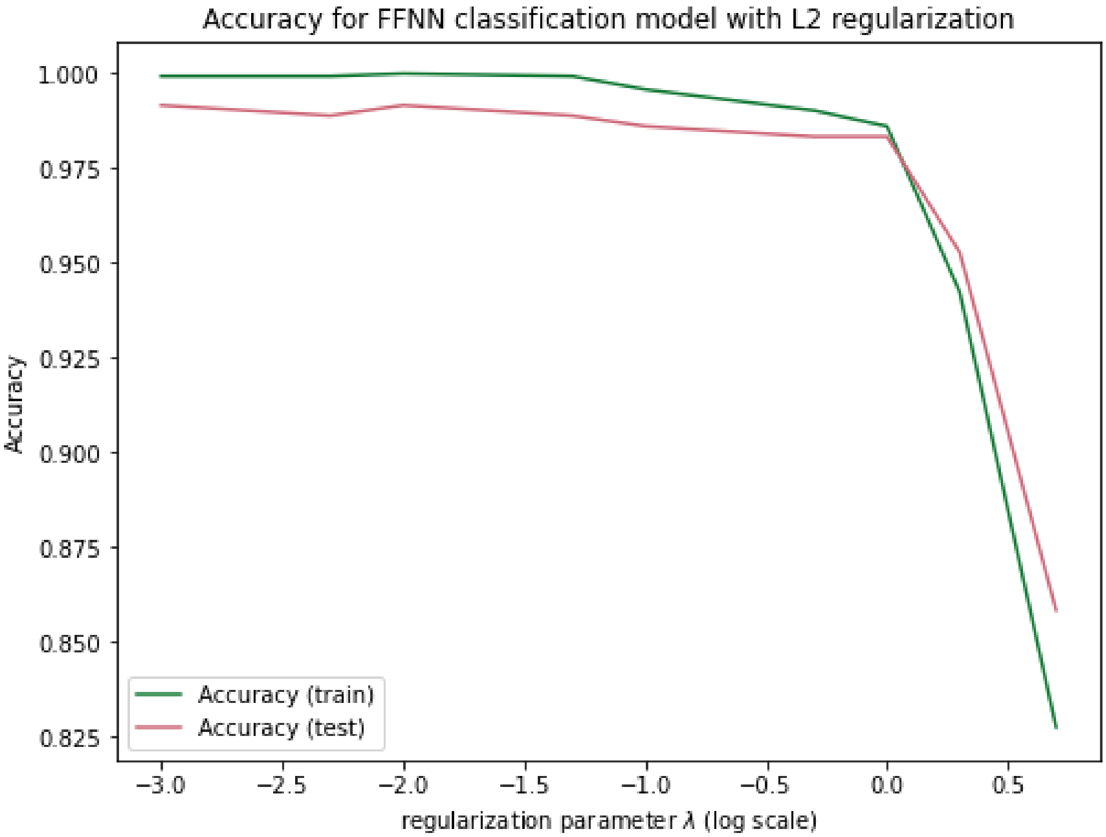

Figure 27: The FFNN classification model with added l2 regularization. One hidden layer with 30 nodes, learning rate is 0.5 and the number of epochs is 100. The hidden layer uses the sigmoid activation function and the output layer uses the softmax.

5.1.3 Logistic Regression - Classification

So far we have used a FFNN neural network for classification. We want to compare our results with a different method. The method of choice is logistic regression. As we are performing classification to multiple classes (not binary classification) we use the softmax in place of the standard sigmoid. This form of logistic regression is called multinomial logistic regression or simply softmax regression. Our cost function is again cross-entropy.

We have our own code for this as well, but first we attempt classification of our digit data with `scikit-learn`'s `SGDClassifier`. Setting the `loss` parameter of this method to 'log' gives a logistic regression for classification with SGD as training method.

When fitting the `SGDClassifier`-model to the training set of the digits data and using the fitted model to predict the digits based on the input values of the test set we get an accuracy score of 0.95. This is exactly the same score that we achieve from our own logistic regression code for multinomial regression with stochastic gradient descent. For both models the learning rate was 0.1.

```
Accuracy when using SGDClassifier with 'log'-loss on digits-data is: 0.95
```

```
Accuracy when using own code for logistic regression with SGD on digits-data is:
0.95
```

Figure 28 shows a plot of how the accuracy after on the test set varies with the chosen learning rate. We see a similar behavior as before, with very small and very large learning rates resulting in a poor accuracy score. While the performance of the two methods is very similar in the middle range of the learning rates it is interesting to see that `SGDClassifier` seems to be less sensitive to the choice of learning rate.

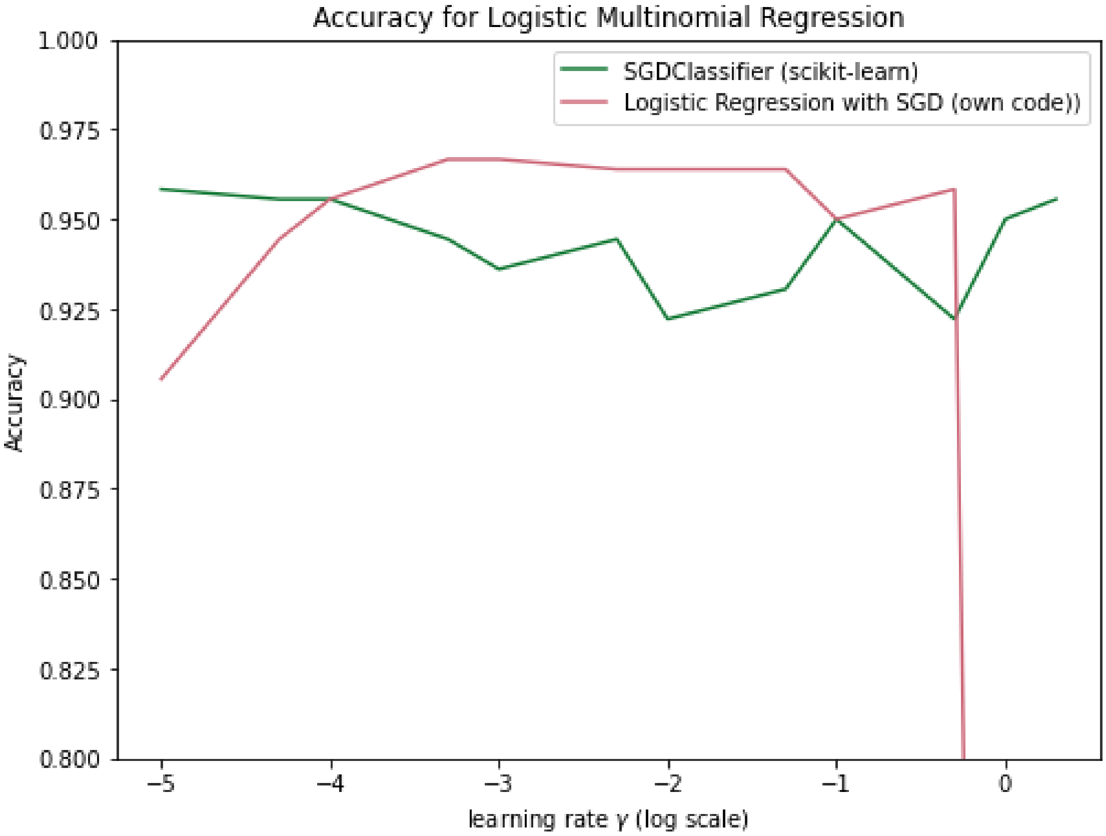

Figure 28: Accuracy on the test set of the digits dataset for varying learning rate on multinomial logistic regression with stochastic gradient descent when the methods are fitted to the corresponding training set.

Finally figure 29 gives a plot of the FFNN classification models when adding regularization in the form of an L2 regularization term. We can see that the accuracy is reduced when the rstrength of the regularization increases, as expected, but interestingly our own FFNN model seems to be affected a lot more stronly than that of scikit-learn.

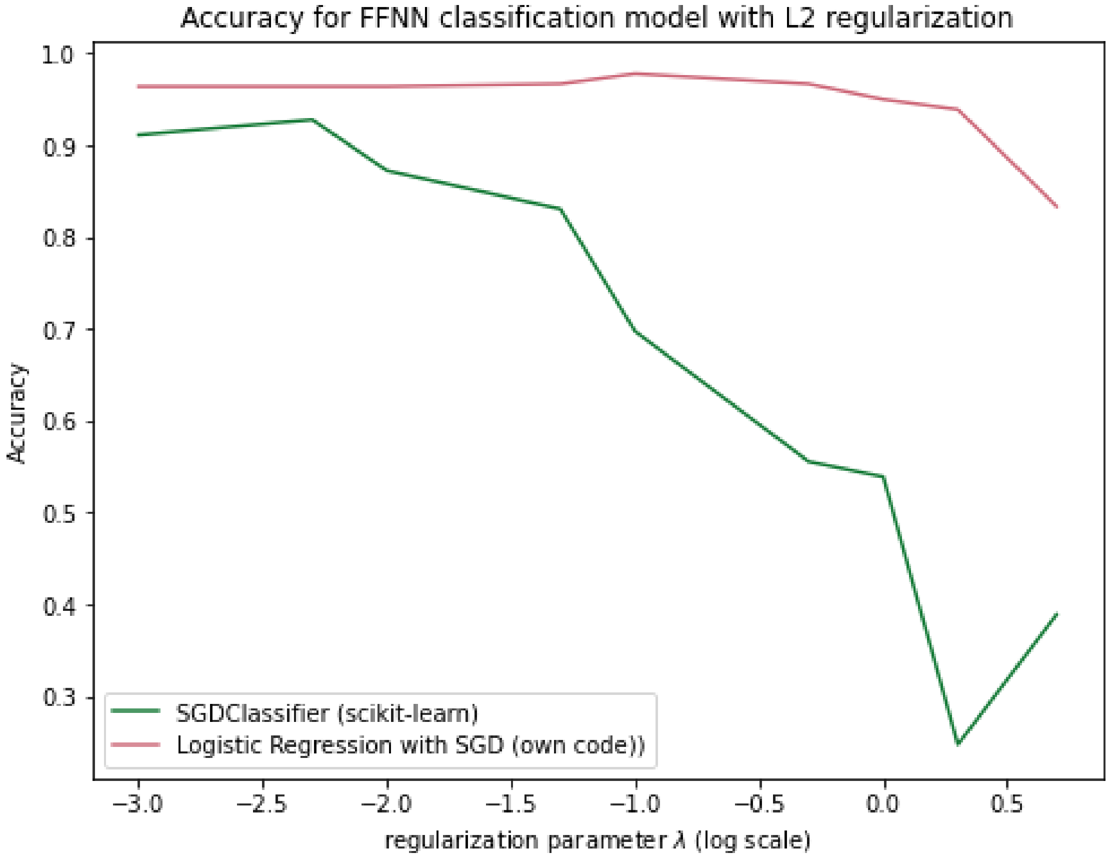

Figure 29: The FFNN classification model with added l2 regularization. One hidden layer with 30 nodes, learning rate is 0.5 and the number of epochs is 100. The hidden layer uses the sigmoid activation function and the output layer uses the softmax.

6 Conclusion

In conclusion we see that our feed forward neural network outperforms our simpler models both for regression and classification. We also see, however that fitting a neural network model requires noticeably more work in order to get optimal fits. We have looked at fairly simple, for lack of a better term, datasets and for more complex systems the difficulty of optimizing hyper-parameters is likely to also increase.

We have seen that neural networks are very versatile models that can adapt effectively to the problem at hand, and when finding a good set up for the model it can give very impressive results.

Some next steps to improve on the results from this project would be to explore other choices for the gradient method. We would also benefit from applying the methods to several more data sets, both to see how the models perform and to build our own intuition as relates to model fitting.

7 Bibliography

[1] Project 1: https://github.com/emiliefj/FYS-STK3155/blob/master/Project1/Report/Project%201%20-%20FYS-STK3155.pdf, timestamp 13.11.2020

[2] Digits dataset in scikit-learn: https://scikit-learn.org/stable/auto_examples/datasets/plot_digits_last_image timestamp 13.11.2020

[3} https://www.bogotobogo.com/python/scikit-learn/scikit-learn_batch-gradient-descent-versus-stochastic-gradient-descent.php, timestamp 13.11.2020

[4] Universal approximation theorem: https://en.wikipedia.org/wiki/Universal_approximation_theorem, timestamp 08.11.2020

[5] Slides week 40: https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40.html, timestamp 10.11.2020

[6] Softmax function: https://en.wikipedia.org/wiki/Softmax_function, timestamp 08.11.2020

[7] Neural Networks and classification of handwritten digits: http://neuralnetworksanddeeplearning.com, timestamp 11.11.2020

[8] Softmax Regression tutorial: http://saitcelebi.com/tut/output/part2.html, timestamp 11.11.2020

[9] Standford Softmax Regression tutorial: http://deeplearning.stanford.edu/tutorial/supervised/SoftmaxRegre timestamp 11.11.2020