A computational analysis of a dense feed forward neural network for regression and classification type problems in comparison to regression methods

Sakarias Frette, Mikkel Metzsch Jensen, William Hirst

(Dated: November 14, 2021)

We implement our own neural network code using SGD, for which we analyze the performance using three different data sets: Franke's function (regression), Breast cancer data (classification) and MNIST digits (classification). By performing a multiple of grid searches for the best hyperparameters we compare the MSE and accuracy score of the neural network to traditional methods such as ordinary least square (OLS), ridge and logistic regression. In addition we compare our neural network to existing software like Scikit-Learn and TensorFlow. We found that the neural network outperformed all other alternative methods both on the regression problem with a MSE score on 0.032 vs 0.049 for OLS, on the breast cancer data with an accuracy of 99.12% vs 95% for Scikit-learn and finally on the MNIST data set with an accuracy of 98% vs 93% for logistic regression. From an analysis of activation functions we found that Sigmoid was generally better than RELU and leaky RELU when giving enough epochs (roughly more than 30). We found a great benefit of introducing momentum to SGD as well as some small benefit of adaptive learning when being limited to a fewer numbers of epochs.

CONTENTS

I. INTRODUCTION

The use of neural networks have exploded in academia the last 10 years, and continue to impress with its capabilities in complex problem solving. For instance, DeepMind made its AlphaGo which beat the reigning champion 4/5 times in the game Go[4], facial recognition is so proficient that it can be utilized as ID verification on the subways in China[3], and cars can now drive them selves on the road with higher than 99% success rate [5].

Neural networks have proven to be quite effective for a variety of classification problems, but also outperforms more traditional methods using linear regression on more complex regression problems. In fact, logistic regression and a neural network with no hidden layers are practically the same thing. Neural networks can however scale much better in dimensionality, to the extent that it can solve complex problems such as quantum many body problems, image recognition and complex differential equations.

In this report we will be comparing our own neural network to methods such as OLS-, Ridge- and Logistic regression on both regression and classification problems as well as validating results with those produced by already existing software like Scikit-Learn and TensorFlow. We also test our network by calculating predictions for simple sets of data, such as logic gates. In addition we study the effect of different hyperparameters such as learning rate, regularization constant, network architecture, and their proficiency on different sets of data.

The rapport is structured in the sections theory, implementation, results and discussion and conclusion. The theory introduces the mathematics and ideas behind stochastic gradient decent and neural networks. In the implementation section we discuss how we have chosen to implement the different theories into our code as well as discussing how the theory relates to each problem we wish to study. The results and discussion section is where we display our results and is divided into the subsections; Validation, Franke's function, Breast cancer classification and MNIST classification. Finally we summarize our findings in the conclusion.

II. THEORY

Machine learning is in a sense just a cost minimization problem with respect to some parameters. For neural networks we denote these as weights and the biases. The main goal is to choose these parameters such that the network is able to make satisfactory predictions for a given problem.

A. Gradient descent

We consider a general parameter $\boldsymbol{\theta} = \{\theta_1, \theta_2, \dots \theta_n\}$ for a n-dimensional problem. The goal is to choose $\boldsymbol{\theta}$ such that we minimize a given cost function $C(\boldsymbol{\theta})$ with respect to a set of data points given by the design matrix $\mathbf{X}$ and corresponding target values $\mathbf{t}$. One common approach to such a problem is the Ordinary Least Squares method (OLS) for which we find the optimal $\boldsymbol{\theta}_{opt}$ that minimize the cost function as

$$\boldsymbol{\theta}_{opt} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}.$$

This is the main workhorse behind many regression problems, but as the number of data points increases the matrix inversion becomes computationally expensive, and the inverse $(\mathbf{X}^T\mathbf{X})^{-1}$ might not exist for all $\mathbf{X}$. The concept of gradient descent provides a computationally efficient alternative to this problem.

1. *Regular Gradient descent*

For a given cost function $C(\boldsymbol{\theta})$ we can approach the minimum by calculating the gradient $\nabla_\theta C$ with respect to the unknown parameters $\boldsymbol{\theta}$. When evaluating the gradient in a specific point $\boldsymbol{\theta}_i$ in the parameter space, the negative of the gradient will correspond to the direction for which a small change $d\boldsymbol{\theta}$ in the parameter space will result in the biggest decrease in the cost function. By choosing a step size η, better known as the learning rate, we can iterate forward in the parameter space as

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta\nabla_\theta C(\boldsymbol{\theta}_i). \tag{1}$$

In order to get convergence towards a minimum we must choose the learning rate η to be sufficiently small such that it does not "step over" the minimum point. The usual approach to choosing η is simply to define it as a hyperparameter and run the gradient descent scheme for different η's and choose the one that yields the best result.

For some simpler cases, where we are able to calculate the so-called Hessian matrix, we can compute an approximation for the optimal learning rate. For a convex problem the gradient is uniquely equal to zero at the global minimum for which the optimal parameter must satisfy

$$g(\boldsymbol{\theta}_{opt}) = \nabla_\theta C(\boldsymbol{\theta}_{opt}) = 0.$$

By using Newton-Raphson's method we can iterate towards the root for the gradient by following the tangent line (1. order approximation) as

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \frac{g(\boldsymbol{\theta}_i)}{g'(\boldsymbol{\theta}_i)},$$

where

$$g'(\boldsymbol{\theta}_i) = \frac{\partial g(\boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i} = \nabla^2_\theta C(\boldsymbol{\theta}_i) = \mathbf{H}.$$

The final iteration scheme reads

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \mathbf{H}^{-1}\nabla_\theta C(\boldsymbol{\theta}_i).$$

The above scheme is a quite effective strategy for gradient descent as the matrix multiplication of the hessian inverse will change the gradient according to the second order approximation of the cost function. [6]

2. *Stochastic gradient descent*

A variation of the regular gradient descent is the Stochastic Gradient Descent (SGD) for which we introduce the concept of mini batches. For each step, or epoch we divide the data randomly into M mini batches of size m. For each batch we compute the gradient and update the unknown parameter using equation 1. Thus we will update the $\boldsymbol{\theta}_{i+1}$ M-times for each step. The main benefit of SGD over regular GD is that

1. we can reduce computation time.
2. we can reduce the risk of getting stuck in local minima.

The first benefit comes into play when we large data sets. Since the computation of the gradient involves matrix operations, the calculation of the full gradient can be more computationally expensive than the calculation of M gradients involving only the data from each batch. The second benefit is due to the fact that SGD introduces some randomness to the movement through parameter space. It is as Grant Sanderson from 3blue1brown put it, it may look as a drunk man stumbling towards the minimum[1]. Thus for small batch sizes we increase the likelihood of escaping local minima that governs only small domains of the parameter space.

3. *Gradient descent with momentum*

As a last alternative to optimize the gradient descent, we introduce momentum, also known as inertia. The key idea is that we keep track of the previous gradient step, such that we can calculate the velocity through parameter space. When increasing the momentum parameter γ the movement through parameter space becomes more steady and less effected by small fluctuations in the cost function. We can think of this as a heavy ball rolling down a rugged landscape. the heavier the ball the less prone to sudden changes in direction it will be. The concept of momentum is a way to further reduce the chance of getting stuck in local minima when travelling through parameters space.

Formally we define this as:

$$\begin{aligned} \mathbf{v}_t &= \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta E(\boldsymbol{\theta}_t), \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \mathbf{v}_t, \end{aligned} \tag{2}$$

where $\mathbf{v}_t$ is the velocity term for time step t. Notice that when choosing $\gamma = 0$ we have zero momentum and equation 2 becomes equivalent with the regular gradient descent method described in equation 1.
In addition there exist a whole variety of different momentum-based methods, for which some of well known might be the "RMS prop" or "ADAM optimizer", but we will limit ourselves to the most simple version described above.

B. Logistic regression

We define the logistic function $p(t)$ as

$$p(t) = \frac{1}{1+e^{-t}} = \frac{e^t}{1+e^t}.$$

We study the binary case where $y_i = 0$ or $y_i = 1$. In addition we introduce a polynomial model of order n as

$$\hat{y}_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2, \ldots, \beta_n x_i^n.$$

We define the probabilities of getting $y_i = 0 \vee 1$ given a input point x_i and beta values $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_n)$ as

$$\begin{aligned} p(y_i = 1|x_i, \boldsymbol{\beta}) &= \frac{e^{\hat{y}_i}}{1+e^{\hat{y}_i}}, \\ p(y_i = 0|x_i, \boldsymbol{\beta}) &= 1 - p(y_i = 1|x_i, \boldsymbol{\beta}). \end{aligned}$$

We want to choose the parameters $\boldsymbol{\beta}$ such that we maximize the conditional likelihood of having the observed data $\mathcal{D} \in \{x_i, y_i\}$ given β, that is the Maximum Likelihood Principle (MLE). This yields

$$P(\mathcal{D}|\boldsymbol{\beta}) = \prod_{i=1}^{n} \Big(p(y_i = 1|x_i\boldsymbol{\beta})\Big)^{y_i} \Big(1 - p(y_i = 1|x_i, \boldsymbol{\beta})\Big)^{1-y_i}$$

In order to minimize computations we can define the log-likelihood as

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

In addition we can rewrite the cost function as a minimization problem by defining the cost function as the negative log-likelihood

$$C(\boldsymbol{\beta}) = -\log P(\mathcal{D}|\boldsymbol{\beta}). \tag{3}$$

We find the gradients as

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\sum_{i=1}^{n} \begin{pmatrix} y_i - p(\hat{y}_i) \\ x_i y_i - x_i p(\hat{y}_i) \\ \vdots \\ x_i^n y_i - x_i^n p(\hat{y}_i) \end{pmatrix} = -X^T\Big(\mathbf{y} - P(X\boldsymbol{\beta})\Big)$$

Furthermore, we add the L2-norm to the cost function

$$C(\boldsymbol{\beta}) = -\log P(\mathcal{D}|\boldsymbol{\beta}) + \lambda||\boldsymbol{\beta}||_2^2, \qquad \lambda > 0$$

where

$$\lambda||\boldsymbol{\beta}||_2^2 = \lambda \sum_{i=1}^{n} \beta_i^2 \quad \implies \quad \frac{\partial \lambda||\boldsymbol{\beta}||_2^2}{\partial \boldsymbol{\beta}} = 2\lambda\boldsymbol{\beta}.$$

This gives the full gradient expression with L2 regularization as

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -X^T\Big(\mathbf{y} - P(X\boldsymbol{\beta})\Big) + 2\lambda\boldsymbol{\beta}.$$

C. Neural networks

We now introduce the basic concepts of the neural network. Specifically we are going to address a dense feed forward neural network, meaning the information is only moving forward trough the network and each node in a given layer is connected to each and every node in the following layer. The general structure of the network is sketched in figure 1. In the beginning we have an input layer containing one node for each feature of the data. The following layers are the so-called hidden layers which can have a custom chosen number of nodes for each layer. Finally we have the output layers with one node per target category. For a simple True and False type of problem a single output node which takes values between 0 and 1 is sufficient for the prediction. For classification between multiple classes we would require an output node for each class.

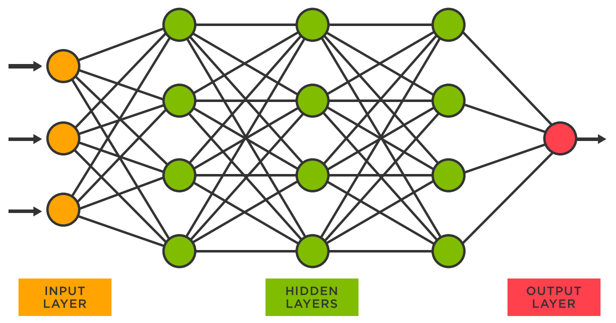


Figure 1: Conceptual illustration of the dense feed forward neural network. Reference (last accessed: 14.11.21)

The connection between the nodes is controlled by the introduction of the weights w and biases b. The weights and biases serve as the unknown parameter. For an already trained network we can collect the predictions by performing the so-called feed forward mechanism which is explained in details in the following section. However, before diving into the details behind the neural network and the key algorithms for performing training and prediction, we want to put down some notation. We aim to use some of the most common notation, but it is nonetheless easy to get lost. Thus we provide a table (I) for which you can always go back to clarify the meaning of variables and indexes.

Table I: Notation

Matrices and vectors		
Notation	Description	Type
X	Design Matrix (input data).	$\mathbb{R}^{N\times\#\text{features}}$
t	Target values.	$\mathbb{R}^{N\times\#\text{categories}}$
y	Model output, the prediction from our network.	$\mathbb{R}^{N\times\#\text{categories}}$
W^l	The weight matrix associated with layer l which handles the connections between layer $l-1$ and l .	$\mathbb{R}^{n_{l-1}\times n_l}$
B^l	The bias vector associated with layer l which handles the biases for all nodes in layer l.	$\mathbb{R}^{n_l\times 1}$
Elements		
w_{ij}^l	The weight connecting node i in layer $l-1$ to node k in layer l.	$\mathbb{R}$
b_j^l	Bias acting on node j in layer l.	$\mathbb{R}$
z_j^l	Node output before activation on node j on layer l.	
a_j^l	Activated node output on node j on layer l.	$\mathbb{R}$
Functions		
C	Cost function	
σ^l	Activation function associated with layer l.	
Quantities		
n_l	The number of nodes in layer l.	
L	Number of layers in total with $L-2$ hidden layers.	
N	Total number of data points.	
All indexing starts from 1: $i, j, k, l = 1, 2, \ldots$		

1. *Feeding Forward*

The forward mechanism is the main mechanism behind the usage of the neural network. For a single data point the data flow becomes as outlined in the following.

1. The input nodes receive the input data for each feature.

2. Each input node sends the data value into each node of the following hidden layer with a scaling according to the associated weight of every single connection.

3. Each node in the hidden layer sums up the weighted contributions from the input layer and adds a bias value associated to that given node. We denote this raw node output by z.

4. The unactivated value z is immediately sent through an activation function σ associated with the layer to produce the activated value $a = \sigma(z)$.

5. Each node in the hidden layer then forwards the activated value into the next layer using the same procedure. Notice that the number of nodes in the hidden layers no longer corresponds to any given features and one can only speculate on how the complex network creates its own sub-features and interprets the underlying correlations in the data.

6. Finally the stream of forwarded data enters the output layer where the activation values on each node corresponds to the network predictions for each category.

The feed forward mechanism for a simple four layer network is shown in figure 2.

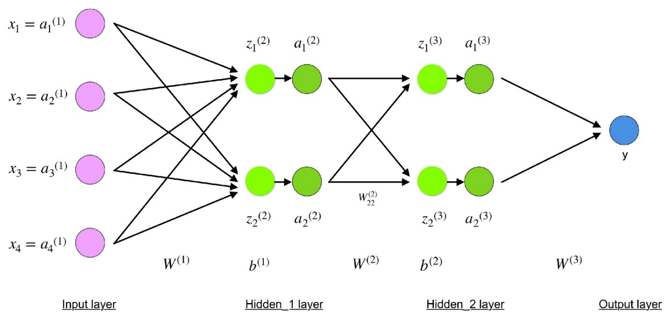


Figure 2: Illustration of a simple 4-layer neural network. Reference (last accessed: 14.11.21)

The feed forward step to obtain the activation value on a given layer $l \neq 1$ (not the input layer) is given as

$$z_j^l = \sum_{i=1}^{n_{l-1}} w_{ij}^l a_i^{l-1} + b_j^l, \quad a_j^l = \sigma^l(z_j^l) \tag{4}$$

where z_j^l is the input value to node j in layer l, w_{ij}^l is the weight connecting node i in layer $l-1$ to node j in layer l, a_i^{l-1} is the activation value from node i in layer $l-1$ and b_j^l is the bias associated with node j in layer l. We can obtain a matrix-variant of equation 4 by defining

$$W^l = \begin{bmatrix} w_{11}^l & w_{21}^l & \cdots & w_{n_{l-1}1}^l \\ w_{12}^l & w_{22}^l & \cdots & w_{n_{l-1}2}^l \\ \vdots & \vdots & & \vdots \\ w_{1n_l}^l & w_{2n_l}^l & \cdots & w_{n_l n_{l-1}}^l \end{bmatrix},$$

$$(W^l)^T = \begin{bmatrix} w_{11}^l & w_{12}^l & \cdots & w_{1n_l}^l \\ w_{21}^l & w_{22}^l & \cdots & w_{2n_l}^l \\ \vdots & \vdots & & \vdots \\ w_{n_{l-1}1}^l & w_{n_{l-1}2}^l & \cdots & w_{n_{l-1}n_l}^l \end{bmatrix},$$

$$A^l = \begin{bmatrix} a_1^l \\ a_2^l \\ \vdots \\ a_{n_l}^l \end{bmatrix}, \quad B^l = \begin{bmatrix} b_1^l \\ b_2^l \\ \vdots \\ b_{n_l}^l \end{bmatrix}, \quad Z^l = \begin{bmatrix} z_1^l \\ z_2^l \\ \vdots \\ z_{n_l}^l \end{bmatrix},$$

such that we get

$$Z^l = (W^l)^T A^{l-1} + B^l, \qquad A^l = \sigma(Z^l). \tag{5}$$

The complete feed forward algorithm is simply done by using equation 5 for layer $l = 2, \ldots, L$, for which we produce the neural network prediction $\mathbf{y}$.

2. *Back Propagation*

The back propagation algorithm is the essential algorithm behind the training of the neural network. This serves the purpose of tuning the weights and biases according to a given cost function. A common choice for regression type problems is the mean squared error

$$C(\mathbf{y}) = ||\mathbf{y} - \mathbf{t}||_2^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - t_i)^2, \tag{6}$$

where N is the number of data points. However, one can choose from a wide selection of cost functions in general. For classification cases it is normal to use Cross entropy as the cost function. In that case it is defined as

$$C(\mathbf{y}) = -\sum_{i=1}^{n}\Big[y_i \log(t_i) + (1 - y_i)\log(1 - t_i)\Big]. \tag{7}$$

By calculating the gradient gradient $\nabla_{w,b} C$ with respect to the weights and biases, we can use gradient descent (see section II A) to minimize the cost function. Thus we need to evaluate $\partial C / \partial w_{ij}^l$ and $\partial C / \partial b_j^l$. We begin by looking at the last layer $l = L$. By using the chain rule we get

$$\frac{\partial C}{\partial w_{ij}^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{ij}^L}.$$

We remember that

$$a_j^L = \sigma(z_j^L), \qquad z_j^L = \sum_{i=1}^{n_{L-1}} w_{ij}^L a_i^{L-1} + b_j^L,$$

such that we get

$$\frac{\partial C}{\partial w_{ij}^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)a_i^{L-1}.$$

Notice that the remaining derivatives can easily be calculated when deciding on a specific cost and activation function. For the bias we simply get

$$\frac{\partial C}{\partial b_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial b_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)\cdot 1.$$

We use this to motivate the definition of the local gradient

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l},$$

such that

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L).$$

This yields the more compact expressions

$$\frac{\partial C}{\partial w_{ij}^L} = \delta_j^L a_i^{L-1}, \qquad \frac{\partial C}{\partial b_j^L} = \delta_j^L.$$

The local gradient δ_j^l is also commonly called the "error" since it reflects how big of an influence the j$^{\text{th}}$ node in layer l have on the change of the cost function. We let δ^l denote the vector containing all the local gradients associated with layer l and we can write it out as a matrix equation for the last layer

$$\delta^L = \nabla_a C \odot \frac{\partial \sigma}{\partial z^L}, \quad \nabla_a C = \left[\frac{\partial C}{\partial a_1^L}, \frac{\partial C}{\partial a_2^L}, \ldots, \frac{\partial C}{\partial a_{n_L}^L}\right]^T,$$

where $\odot$ is the Hadamard product (element-wise product). We can then define the gradient δ_j^l for the j$^{\text{th}}$ node on a general layer l in terms of δ_k^{l+1} for the k$^{\text{th}}$ node on the following layer $l+1$ by using the chain rule

$$\begin{aligned}\delta_j^l &\equiv \frac{\partial C}{\partial z_j^l}\\ &= \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l}\\ &= \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l}\delta_k^{l+1}. \qquad (8)\end{aligned}$$

We remember

$$z_k^{l+1} = \sum_{j=1}^{n_l} w_{jk}^{l+1}a_j^l + b_k^{l+1} = \sum_{j=1}^{n_l} w_{jk}^{l+1}\sigma(z_j^l) + b_k^{l+1},$$

and by taking the derivative we obtain

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{jk}^{l+1}\sigma'(z_j^l). \qquad (9)$$

We substitute back 9 into 8 to find

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{ij}^{l+1}\sigma'(z_j^l).$$

The complete back propagation algorithm then becomes

- Compute
$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L).$$
- For $l = L-1, L-2, \ldots, 1$ compute:
$$\delta_j^l = \sum_k \delta_k^{l+1} w_{jk}^{l+1}\sigma'(z_j^l).$$
- For all layers l update weights and biases:
$$\begin{aligned} w_{ij}^l &\leftarrow w_{ij}^l - \eta\delta_j^l a_i^{l-1},\\ b_j^l &\leftarrow b_j^l - \eta\delta_j^l,\end{aligned}$$
where η is the learning rate.

In order to minimize the risk of overfitting it is common to include a L2 regularization by adding the L2 norm to the cost function as $\lambda||w||_2^2$ and $\lambda||b||_2^2$. This result in the modified expressions for the update

$$\begin{aligned} w_{ij}^l &\leftarrow w_{ij}^l - \eta(\delta_j^l a_i^{l-1} + \lambda w_{ij}^l),\\ b_j^l &\leftarrow b_j^l - \eta(\delta_j^l + \lambda b_j^l).\end{aligned}$$

III. IMPLEMENTATION

The code can be found on Github.

A. SGD algorithm

We created a class, *SGD* (found in SGD.py), which handles stochastic gradient descent. The class constructor takes the inputs

- Design matrix X
- target values t

- learning rate η
- Batch size m
- Number of epochs
- Regularization parameter λ
- Momentum parameter γ
- Gradient function (Ridge or Logistic)
- Loss function for evaluation of prediction score (Accuracy, MSE or r^2)
- Callback, option to print score history (True or False)

The class constructor calls an initialization function which initializes theta with random samples from a standard normal distribution. By simply calling the class method *SGD_train* we perform SGD using the formulas 2 described in the theory section II A 2. By varying the input for gradient function and regularization parameter λ one can choose between OLS, Ridge and logistic regression type of SGD. The general SGD algorithm is outlined in algorithm 1.

Algorithm 1: Stochastic gradient descent

```
1  Initialize $\theta$;
2  for epoch in epochs do
3      Create batches;
4      for batch in batches do
5          Fetch X, y data from batch;
6          Find gradient;
7          Update velocity;
8          Update $\theta$;
9      end
10 end
11 Return $\theta$
```

B. Neural network

Similar to SGD, we create a class to handle all neural network calculations. The class constructor takes in some similar values as the SGD class, which is data in the form of the design matrix X, and the target values t, and then hyper parameters η, λ and γ. Also we have a similar option to specify the loss calculation and callback True or False. In addition one must specify the network architecture given by

- Number of hidden layers
- Number of nodes per hidden layer
- Activation functions on all layers (Sigmoid, RELU, Leaky, RELU, Soft Max)
- Alternatively option to have a different activation function on the output layer

and finally

- Cost function (MSE or Cross Entropy)

Based on the chosen parameters for the network, the constructor calls the methods *create_layers* and *create_biases_and_weights* which setup matrices to hold unactivated values z, activated values a and all the weights W^l and biases B^l based on the definitions introduced in forward and backward propagation (see section II C). Similar to SGD we choose to initialize the weights by sampling from a standard normal distribution while all the biases is initialized with a value of 0.1. In our implementation we choose to have the same number of hidden nodes on all the hidden layers to simply the usage of the class, since we considered the additional flexibility unnecessary for the scope of our studies.

When all matrices and functions are created, the network can be trained by the call of *train_network_stochastic(epochs)* which requires a choice of number of maximum epochs. The training follows the same structure as SGD while the stopping criteria is not met (more on this in III E 2). For each epoch we perform the feed forward algorithm followed by the back propagation algorithm, updating all weights and biases as described in section II C. The algorithm for the training is outlined in algorithm 2.

Algorithm 2: Neural network training

```
1  Initialize layers, weights, biases;
2  while epoch <= epochs and stopping criteria not met
     do
3      for batch in batches do
4          Assign reduced data X[batch] to input
             layer ;
5          Feed Forward;
6          Back propagate;
7          Update Weights and Biases;
8      end
9  end
```

By calling the method *predict(X)* the network outputs a prediction based on a given design matrix X. The algorithm for the prediction is outlined in algorithm 3

Algorithm 3: Neural network prediction

```
1  Assign given design matrix $X$ to inpout layer;
2  Feed Forward;
3  Collect activation value on last layer and return as
     prediction;
```

C. Activation functions

We defined four different activation functions to use in our network, namely the Sigmoid, RELU, leaky RELU and SOFTMAX. Sigmoid is defined as

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

RELU is defined as

$$f(x) = max(0,x)$$

Leaky RELU is defined as

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

SOFTMAX is defined as

$$f(x) = \frac{e^x}{\sum_j e^{x_j}}$$

D. Layers, weights and biases

E. Practical implementation

1. Adaptive learning rate

Using a constant learning rate η can sometimes lead to long computation time or overstepping a minimum. A way to correct this is to adjust the learning rate η as a function of epochs. We propose the following function to handle adaptive learning rate as

$$\eta(\xi) = \eta_0 A \cdot \sigma_s(k \cdot (t_D - \xi))$$

For an initial learning rate η_0, amplitude A, drop time t_D as a measure of a characteristic decrease point, epoch ξ, steepness parameter k and the Sigmoid activation function σ_s. The amplitude is a correction parameter to ensure that the learning rate starts at η_0 for the first epoch defined as

$$A = \frac{1}{\sigma_s(k \cdot t_D)}$$

In figure 3 we showcase how the logistic functions in $\eta(\xi)$ can be adjusted to also take shape of exponential decay and constant functions.

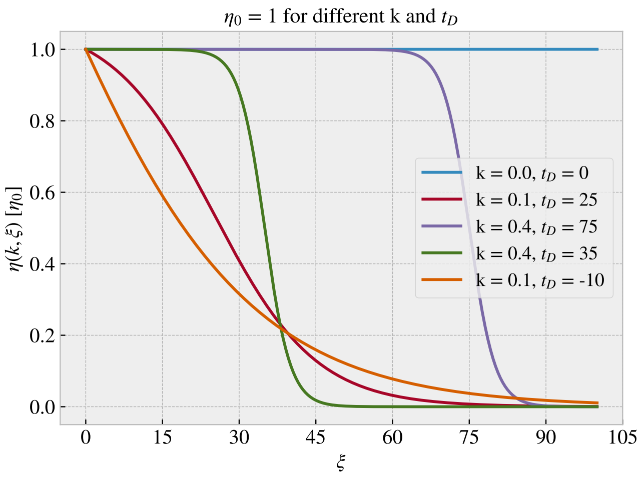


Figure 3: The adaptive learning rate function $\eta(\xi)$ η for various choices of steepness k and drop time t_D as a function of epoch ξ.

2. Handling vanishing and exploding gradients

A common problem when training the neural network is that the gradient norm either goes towards zero (vanishing) thus stopping the training or grows too large (exploding) resulting in overflow. In the first case the continuation of the learning becomes meaningless as the learning slows done extensively. In the latter case the continuation of the training will actually undo the training until that given point. Thus it is beneficial to stop the training in both cases. In practice we prevent vanishing gradient by having a lower tolerance of 10^{-8} for the norm $|\eta\delta^L|$. Exploding gradients is managed by requiring the same norm to be finite, thereby salvaging useful results that would otherwise be destroyed.

3. Cost functions

We used two cost functions: Mean squared error for regression type problems and cross entropy for classification problems.

4. RELU and leaky RELU

As will be discussed in later sections, there are certain complication when using RELU and leaky RELU as activation functions. The main problem is that they do not force the resulting values between 0 and 1. This is a problem when studying binary classification problems. Therefore we will be using the Sigmoid function on the final layer when using RELU and leaky RELU in classification, mainly in section IV B 2.

F. Data sets

1. *Franke's function*

For the regression case we will be using Franke's function. The Franke's function, f(x,y), is a analytical function given as a weighted sum of four exponentials, which we will study in the domain $x, y \in [0,1]$. The function reads as follows

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

In the case of using our neural network we define the input variables, x and y as two features of the data. In the case of SGD for OLS and RIDGE we define a design matrix, where each feature represents a polynomial degree, see [2] for more information.

Since x and y is uniformly distributed between 0 and 1 there is no need for any further scaling. The corresponding z-values in the x, y-domain is pretty much confined to the same range between 0 and 1, and thus we leave this unscaled as well.

2. *Breast cancer*

For classification we will first be using scikit learns breast cancer data. The data contains 569 data points, each with 30 features. Each feature is a specific medical attribute of a tumor, such as radius, area etc. while the output is given as binary value, malignant (dangerous = 0) or benign (not as dangerous = 1). The breast cancer data set has entries with values spanning from 0 to 4000, with varying size of magnitude for different features. Thus we apply standard scaling here, subtract by mean and divide by standard deviation, on the columns of the design matrix.

3. *MNIST*

The MNIST data set contains 1797 images of handwritten numbers between 0-9. Each image has a resolution of 8×8 pixels. An example of the data is shown in figure 4.

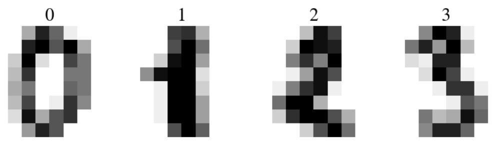


Figure 4: A set of four datapoints from sckit learns MNIST. Each point with the corresponding value written above.

The MNIST output data is structured as a list with 1797 elements, where each element is a number between 0-9. In order to use cross entropy as classification cost function we restructure the data to have a binary output. That is, for each data point we have 10 categories corresponding to all possible digits which takes values 0 (False) or 1 (True). Thus the mapping is handled as shown in the following examples.

$$3 \rightarrow \begin{bmatrix}0\\0\\0\\1\\0\\0\\0\\0\\0\\0\end{bmatrix} \quad 7 \rightarrow \begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\1\\0\\0\end{bmatrix} \quad \text{or } 5 \rightarrow \begin{bmatrix}0\\0\\0\\0\\0\\1\\0\\0\\0\\0\end{bmatrix} \tag{11}$$

The images are all represented as 8×8 matrices with values varying from 0 up to 500. Therefore, as discussed in previous section we must scale our input. We do so by the previously mentioned, standard scalar.

4. *Scaling of the weights*

Activation functions like the Sigmoid function "squeeze" the data between 0 and 1, which is well behaved under matrix multiplication, but rectifier functions like RELU and Leaky RELU have a linear dependence on the data, and thus can lead to very large values in the hidden layers during matrix multiplication. To mitigate this we added a scaling of the weights when initialization the network. We scale by dividing each element in a matrix of weights by the total number of elements in the matrix.

IV. RESULTS AND DISCUSSION

Note: In the following section when comparing results for different parameters, we chose the same seed for a fair comparison.

A. Validation

1. *SGD*

Note: Say that we choose same seed for repeated experiments for a fair comparison...

We begin by validating and testing the implementation of our SGD method. For this we use Franke's function, which we analyzed in project 1 [2]. We use a default 100×100 meshgrid (10^4 data points), a complexity of $n = 5$, referring to the highest polynomial order, and add normal distributed noise to the output with standard deviaion $\sigma = 0.2$. When comparing train and test results we use a default split of 80% training data and 20% test data.

First, we take on analyzing non-momentum OLS-gradient SGD. We investigate the convergence of the model as a function of epochs regarding the hyperparameters: Batch size m and learning rate η. We do this by keeping one parameter constant and changing the other respectively. For this we use the whole data set, and thus the MSE is for the training data. The result for varying batch size and fixed learning rate is shown in figure 5 and the results for varying learning rate and constant batch size is shown in figure 6

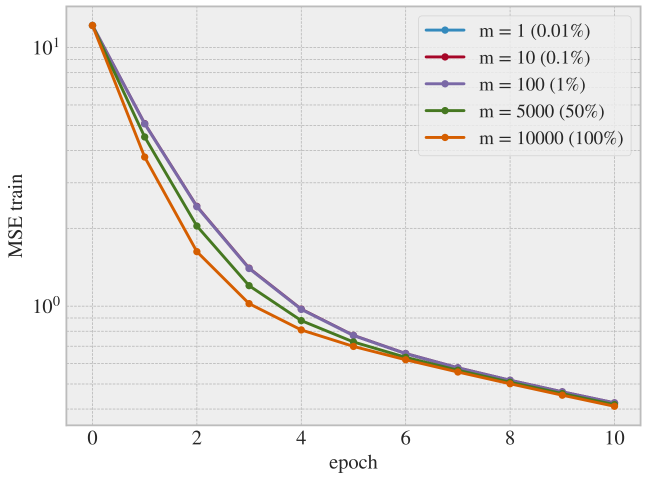


Figure 5: MSE on training data for SGD regression on Franke's function (default settings) with fixed learning rate $\eta = 0.1$ and varying batch size m. The percentage denotes the batch size relative to the number total number of data points. Notice that the curves for $m = 1$ (blue) and $m = 10$ (red) hides beneath the curve for $m = 100$ (purple).

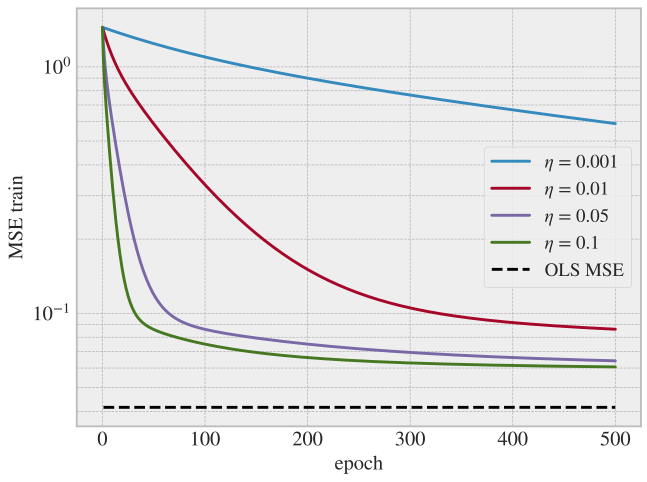


Figure 6: MSE on training data for SGD regression on Franke's function (default settings) with full gradient descent ($m = 10^4$) and varying learning rate η. The black-dotted line indicates the MSE for OLS regression.

From figure 5 we see that the convergence rate is only slightly affected by the batch size, with a small benefit of having the biggest batch size. But the difference becomes negligible already at the 10th epoch onwards (Note: this was even less clear with other seeds). However, from figure 6 we see that the learning rate has a much more distinct impact on the convergence rate. To get a better understanding of the influence from the learning rate we compute the final MSE after 10^3 epochs for both training and test data (default split: 0.2) as a function of the learning rate with full gradient descent (see figure 7).

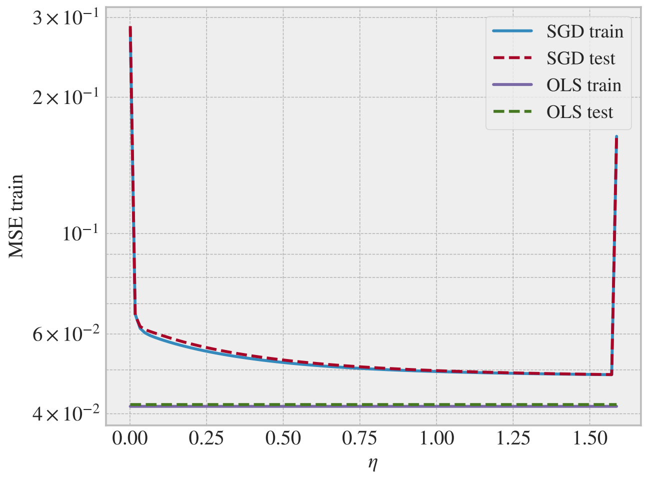


Figure 7: MSE on training and test data for SGD regression on Franke's function (default settings) with full gradient descent for 10^3 epochs as a function of learning rate η. This is compared to OLS regression for train and test data seen in the bottom of the plot.

From figure 7 we see that the best performance from the SGD is found with a quite high learning rate. The best MSE score was found to be

$$\text{SGD}: 0.0496,$$
$$\text{OLS}: 0.0424,$$

with

$$\arg\min_{\eta}(\text{MSE test}) = 1.57 \pm 0.02$$

With a resolution of 0.02 in the leaning rates, the minimum point occurs on the second last index of the tested learning rates, while the last index correspond to the peak on the right-hand side of the plot. Thus we see an apparently non-continuously behaviour (considering our resolution) where the best learning rate comes just before a critical limit where the learning rate will yield poor convergence. However, when running similar test on other data sets we see that this limit changes quite a lot, and thus one should be careful of using this as a safe limit. For the following results we use $\eta = 0.1$ as a default value.

We now compare OLS and SGD for different model complexities. Notice that we used $n = 5$ up to this point, but now we take on a smaller data set of 20×20 (400 data points) and $n = [1, 20]$. For SGD we use full gradient descent and 10^4 epochs. We continue to use a split of 0.2 and noise $\sigma = 0.2$. The results are shown in figure 8

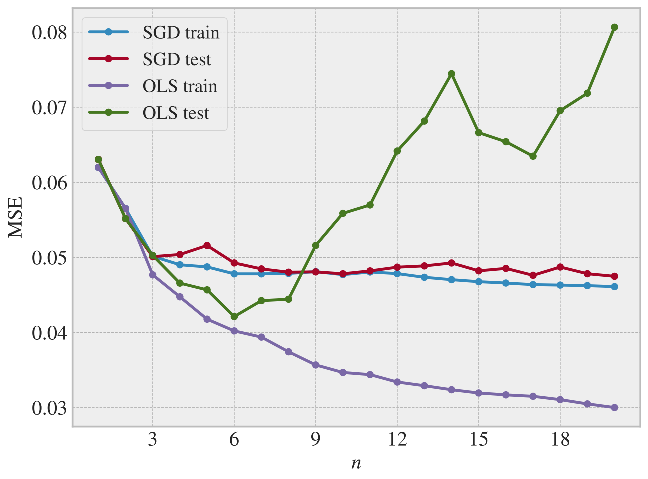


Figure 8: Comparison of SGD and OLS on Franke's function for varying model complexity n. SGD uses full gradient descent with 10^4 epochs and learning rate $\eta = 0.1$. The data consists of 400 data points.

As expected from earlier results, we see that OLS outperforms SGD on the low model complexity which we have already seen for $n = 5$ in figure 7. However, as the model complexity increases the OLS regression results in overfitting and thus the test MSE increases considerable. SGD is not really prone to overfitting in this particular case, and we see that the MSE is steady, actually slightly decreasing, for increasing model complexity.

Until now we have mostly been using full gradient descent as we saw (from figure 5) that this gives a slightly better convergence on a short scale of epochs for the Franke's function data. When introducing the neural network we expect to deal with far larger gradients (for weights and biases) which will result in heavier matrix operations. For smaller batch sizes we reduce the size of these matrix operations with a cost of having to run through a for-loop for more iterations. In order to get an idea of this computational balance of for-loop iteration and matrix operations we perform a timing of the algorithm for increasing sizes of the design matrix. We fix $n = 10$, resulting in 66 features and increase the size of the meshgrid from 100×100 (10^4 data points) to 1000×1000 (10^6 data points) while measuring the time pr. epoch. averaged over 10 epochs. We use $\eta = 0.1$ and initialize from same random seed each time. The results are shown in figure 9

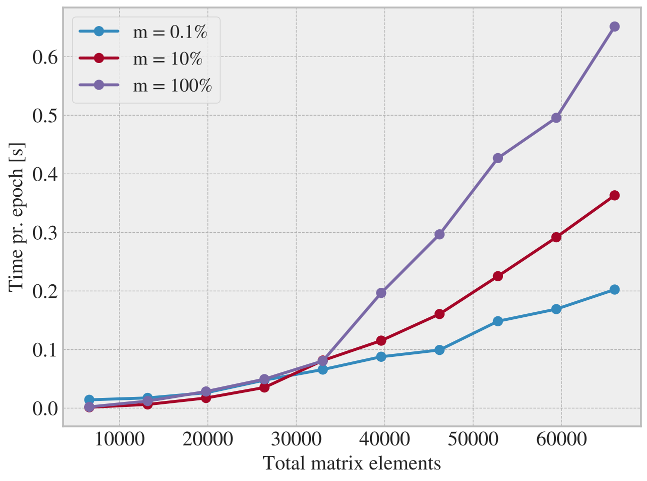
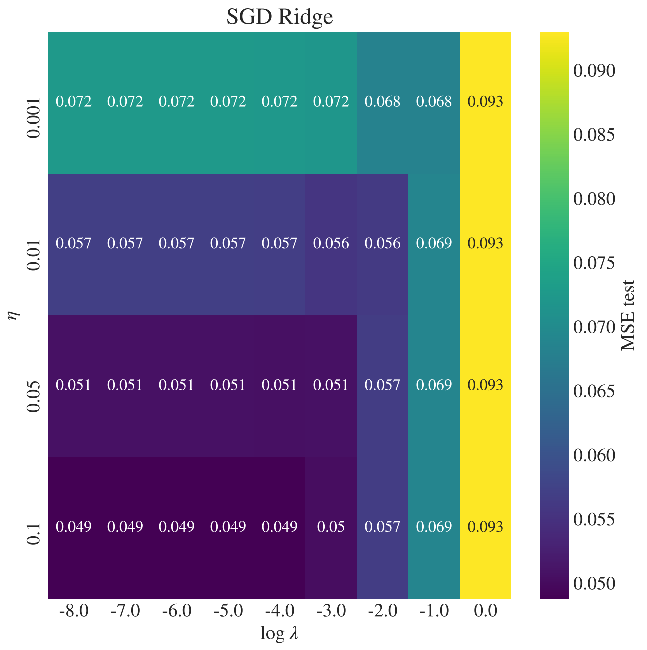


Figure 9: Timing of SGD as a function of the total number of elements in the design matrix for different batch size (given as a percentage of the total number of data points). While keeping $n = 10$, corresponding to 66 features, we varied number of data points in the interval $N \in [10^4, 10^6]$ and measured the computation time pr. epoch. We ran each SGD-regression for 10 epochs staring from the same random seed. We see that small mini batches gets computational beneficial for big matrices.

From figure 9 we see that for big matrix sizes, the smallest batch size yields the lowest computation time. For smaller matrices, the small batches actually gives a slightly longer computation time, and the intersection between these domains is visually determined to be around 30,000 matrix elements.

We also want to compare Ridge regression and SGD with Ridge gradient. In order to do this we begin by studying the optimal hyperparameters for SGD with with Ridge gradient. When fixing the batch size to full gradient descent we are left with learning rate η and the regularization parameter λ as the main hyperparameters. We go back to using the default data set, as stated in the beginning of this section and compute the MSE on the test data for different combinations of η and λ. The result is shown in figure 10



η \ log λ	-8.0	-7.0	-6.0	-5.0	-4.0	-3.0	-2.0	-1.0	0.0
0.001	0.072	0.072	0.072	0.072	0.072	0.072	0.068	0.068	0.093
0.01	0.057	0.057	0.057	0.057	0.057	0.056	0.056	0.069	0.093
0.05	0.051	0.051	0.051	0.051	0.051	0.051	0.057	0.069	0.093
0.1	0.049	0.049	0.049	0.049	0.049	0.05	0.057	0.069	0.093

Figure 10: MSE test for SGD with Ridge gradient for different combinations of learing rates η and Ridge regularization parameter λ on Franke's function. We used the default data set with full gradient descent.

From figure 10 we see that the best MSE test is found at the highest tested learning rate $\eta = 0.1$ and the lowest regularization values $\lambda \leq 10^{-4}$. Looking back to figure 7 one could theorize that we have a similar dependence on the learning rate as we had with the OLS gradient.

We then fix the learning rate to $\eta = 0.1$ and compute the MSE on test data (default data) for different λ-values using 10^4 epochs for the SGD. The result is shown in figure 11.

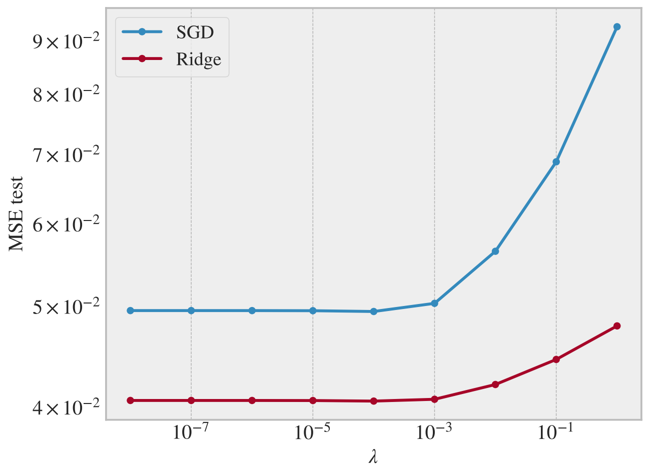


Figure 11: MSE test for SGD (Ridge gradient) and Ridge regression as a function of λ on Franke's function. We used 10^4 epochs and $\eta = 0.1$ with full gradient descent on the default data set.

From figure 11 we see that the lowest MSE is achieved for decreasingly values of λ, reducing the comparison to the one of OLS. The best MSE is found to be

$$\text{SGD} : 0.0477$$
$$\text{Ridge} : 0.0401$$

Considering the previous results it appears that there is no big benefit of introducing the regularization parameter associated with Ridge regression.

Finally we introduce the SGD with momentum. We use full gradient descent on the default data and investigate the convergence rate for different learning rates and momentum parameter γ. The result is shown in figure 12.

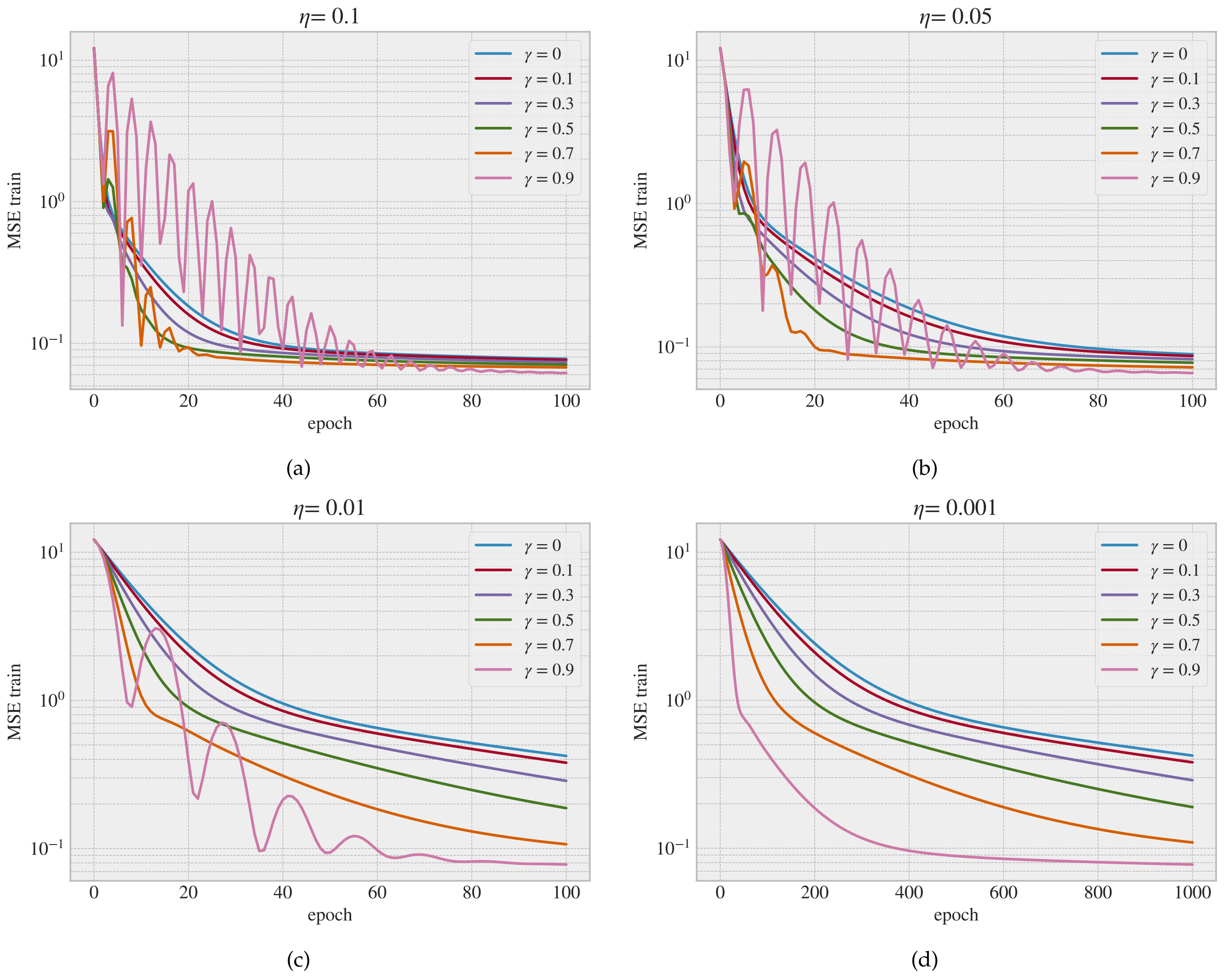


Figure 12: MSE on training data for different momentum parameters γ and learning rates η as a function of epochs. We used full gradient descent on the default Franke's function data set.

From figure 12 we see that momentum parameter has a quite significant influence on the convergence rate. For the small learning rate $\eta = 0.001$ (subfigure 12d) the highest tested $\gamma = 0.9$ showcased a clear advantage. However, for bigger learning rates, $\eta = 0.1$ (subfigure 12a) the higher γ's gave an oscillation MSE and $\gamma = 0.9$ converged the slowest, meaning that it decreases more slowly. But as the oscillations settles down we actually get the lowest MSE after 100 epochs with $\gamma = 0.9$ for all learning rates tested. When increasing γ to 0.99 we see the same effect but on a much bigger scale of epochs. Although the high momentum seems beneficial throughout the testing on Franke's function data this might be completely different for other data sets.

2. *Neural network*

In this part we study our implementation of the neural network. As a starting point we choose three logic gates: AND, OR, XOR (exclusive or) which is represented as the data points in table II

Table II: Logic gates: AND, OR and XOR

Input		Output		
x_1	x_2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

When performing standard OLS we get the predictions shown in table III, where we get 100% accuracy on the AND and OR gate but only 75% on the XOR gate. When displaying the data points graphically it is clear to see that the data points of the XOR-gate cannot be separated by a single straight line (see figure 13). Hence it is clear that OLS will never give a satisfactory prediction on this nonlinear problem.

Table III: OLS predicitons on the logic gates: AND, OR and XOR

Gates	Predictions	Accuracy
AND	$y = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$	100%
OR	$y = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$	100%
XOR	$y = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$	75%

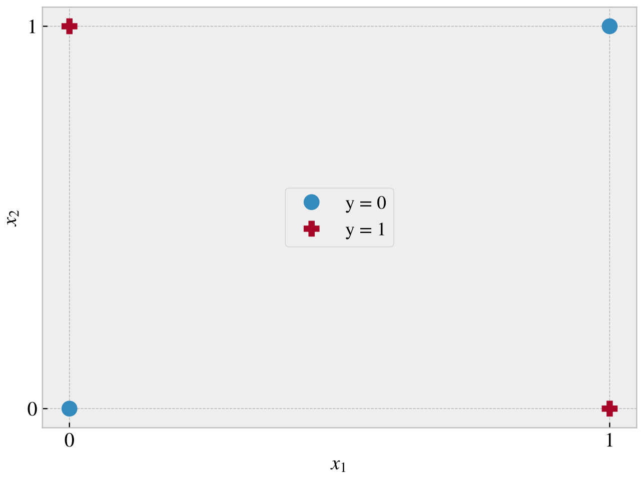


Figure 13: Data points for the XOR-gate. Showcases graphically how we cannot separate the 0's (blue) and 1's (red) by a single straight line

Hence the XOR gate is a good benchmark for our neural network. We choose a simple architecture with 1 hidden layer containing 4 hidden nodes. Notice that it is possible to solve the XOR-problem with only 2 hidden nodes, but we found that having 4 nodes made the training faster and more consistent. We use full gradient descent, $\eta = 1$ and $\lambda = \gamma = 0$ with Sigmoid as activation function and cross entropy as the cost function. We manage to get a perfect fit, 100 % accuracy, on the training data after roughly 70 epochs. The learning accuracy development is shown in figure 14.

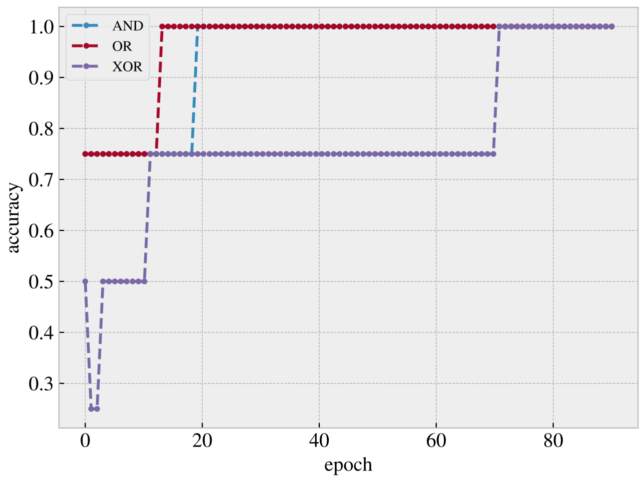


Figure 14: Accuracy as function of epochs for AND, OR and XOR gates.

From the training on the logic gates we have validated that a single neural network is able to outperform OLS for the non-linear problem of the XOR-gates.

Next we study regression on the Franke's Function data from section III F 1 using our neural network. We use a default data set of $N = 100$ data points, and add normal distributed noise to the output with standard deviation $\sigma = 0.2$. When comparing train and test results we use a default split of 80% training data and 20% test data. Before making the comparison between the neural network and the regression models we perform a grid search for the optimal hyperparameters for the neural network. First we optimize for η and λ (see figure 15) before running an additional grid search with numbers of layers and number of nodes per hidden layer (see figure 16 and 17).

In figure 15 we find the optimal hyperparameters to be $\lambda = 10^{-4}$ (smallest tested) and $\eta = 10^{-1}$. From a previous study of Ridge regression[2], we found that the best fit to Franke's function was achieved with a rather small regularization parameter as well, and the demand of a small λ seems to be a problem specific trend. Using the above hyperparameters we perform the grid search for number of layers and number of nodes as shown in figure 16

In figure 16 we observe initially that 4 hidden layers and 50 nodes per layer is the optimal choice for the Franke's function data. By decreasing the grid spacing even more we perform a second grid search as shown in figure 17.

In figure 17 we observe upon further inspection, with a much smaller grid spacing, that an ideal architecture for this problem is given by having 3 hidden layers and 42 nodes. Using the optimal hyperparameters found from the previous grid searches we train our neural net-

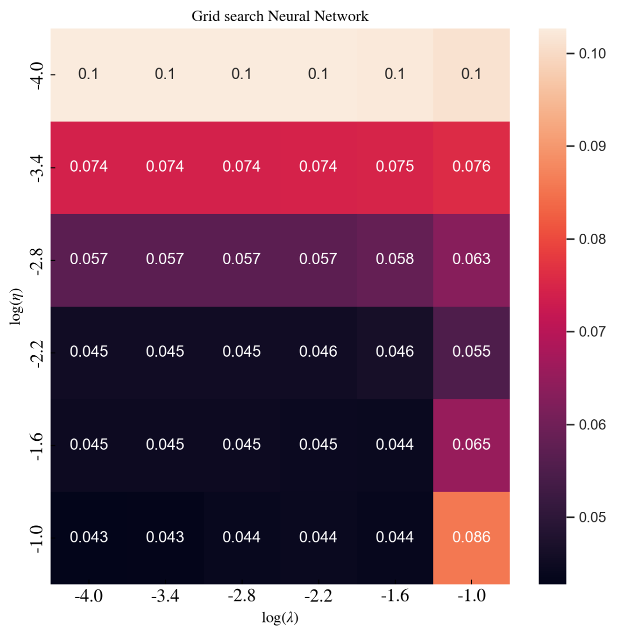


Figure 15: Heatmap showing test MSE score for different hyperparameters $\lambda \in [10^{-4}, 10^{1}]$ and $\eta \in [10^{-4}, 10^{1}]$. Max number of epochs is 400, batch size 25, 2 hidden layers and 70 hidden nodes

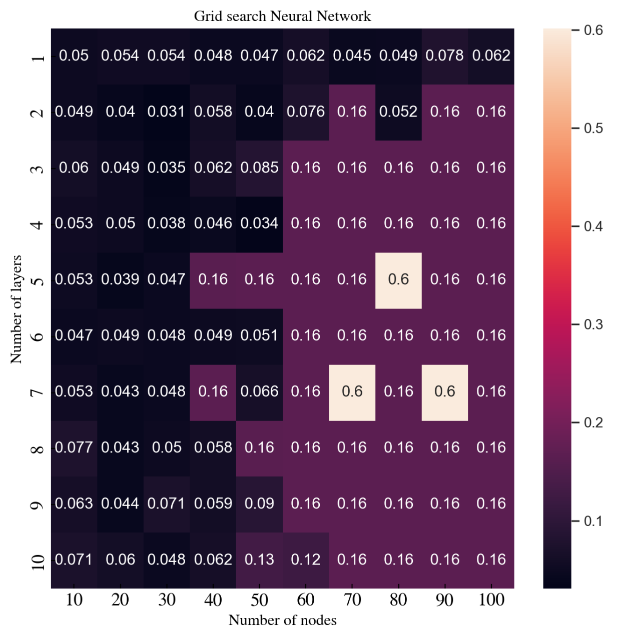


Figure 16: Heatmap showing test MSE score for different hyperparameters $\lambda = 10^{-4}$ and $\eta = 10^{-1}$. Max number of epochs is 400, batch size 25, hidden layers $\in [1, 10]$ and hidden nodes $\in [10, 100]$

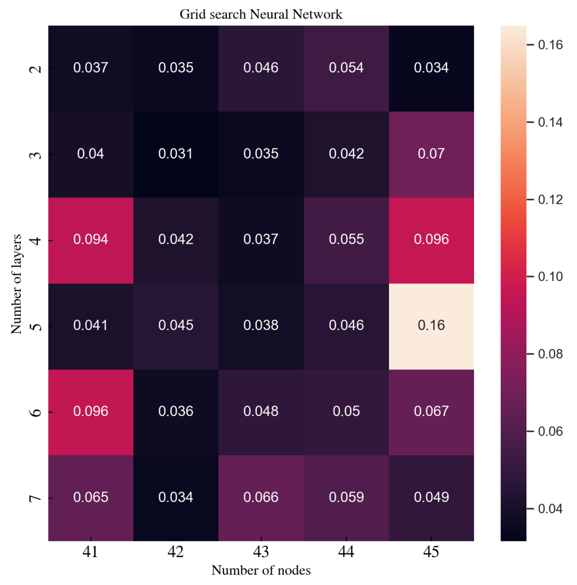


Figure 17: Heatmap showing test MSE score for different hyperparameters $\lambda = 10^{-4}$ and $\eta = 10^{-1}$, a maximum of 400 epochs, batch size 25, hidden layers $\in [2, 7]$ and hidden nodes $\in [41, 45]$

work and compare the test scores against OLS and Ridge regression using optimal hyperparameters determined in a earlier study [2]. The results displayed in table IV.

Table IV: Regression scores (MSE and r^2) for regression on Franke's function with our neural network, OLS regression and Ridge regression with $\lambda = 10^{-4}$, $\gamma = 0$, $\eta = 10^{-1}$, 3 hidden layers, 42 nodes, batch size = 25 and a max number of epochs equal to 400 epochs.

Score	**NN**	**OLS**	**Ridge**	**TensorFlow**
MSE	0.032	0.049	0.049	0.062
R2	0.844	0.716	0.716	0.489

From table IV we see that our neural network gives a slightly better fit compared to OLS and Rigde. Surprisingly TensorFlow gave the worst resultat, but we suspect that it might be due to an unfortunate initialization of the weights and biases. For a more fair comparison one should also make a grid search using TensorFlow since the optimal hyperparameters might be affected by the weights and bias initialization specific to TensorFlow.

Using the optimal hyperparameters we plot the network predictions as 3D plot. Since the number of data points in the test data (20 data points) is not really sufficient to get a clear picture of the predicted model we generate a new validation data (see figure 18) set with similar settings to plot the predictions. The prediciton result is shown in figure 19.

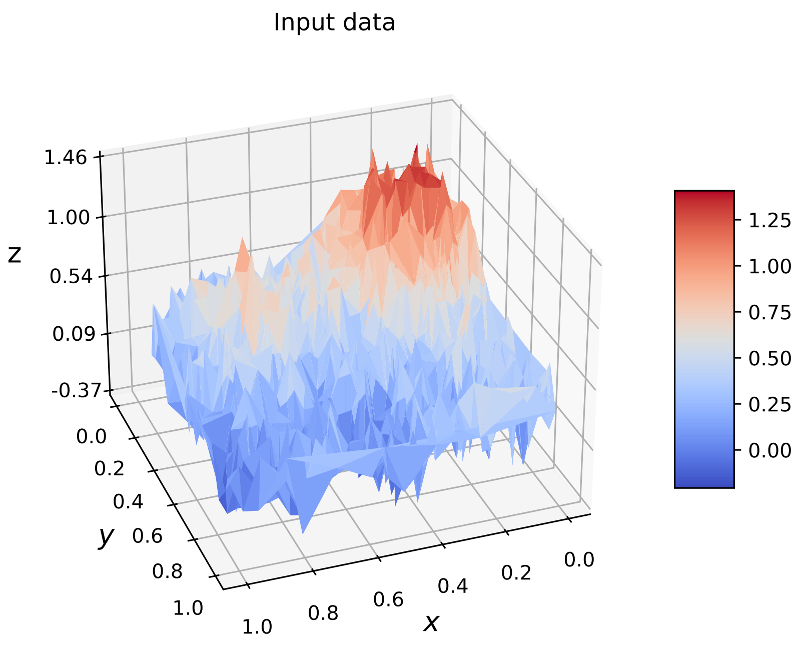


Figure 18: 3D plot of the test data with hyperparameters $\lambda = 10^{-4}$ and $\eta = 10^{-1}$. Max number of epochs 200, batch size 25, 2 hidden layers and 30 hidden nodes.

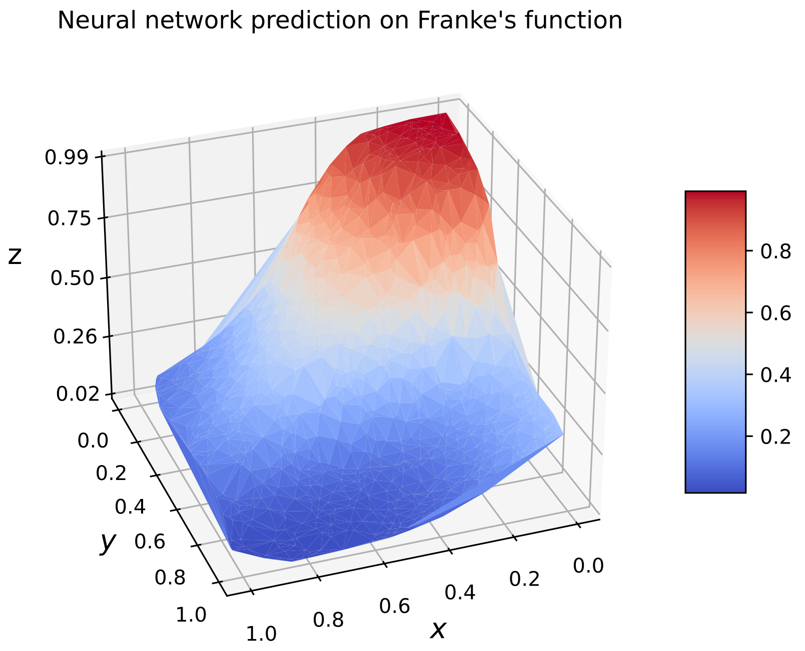
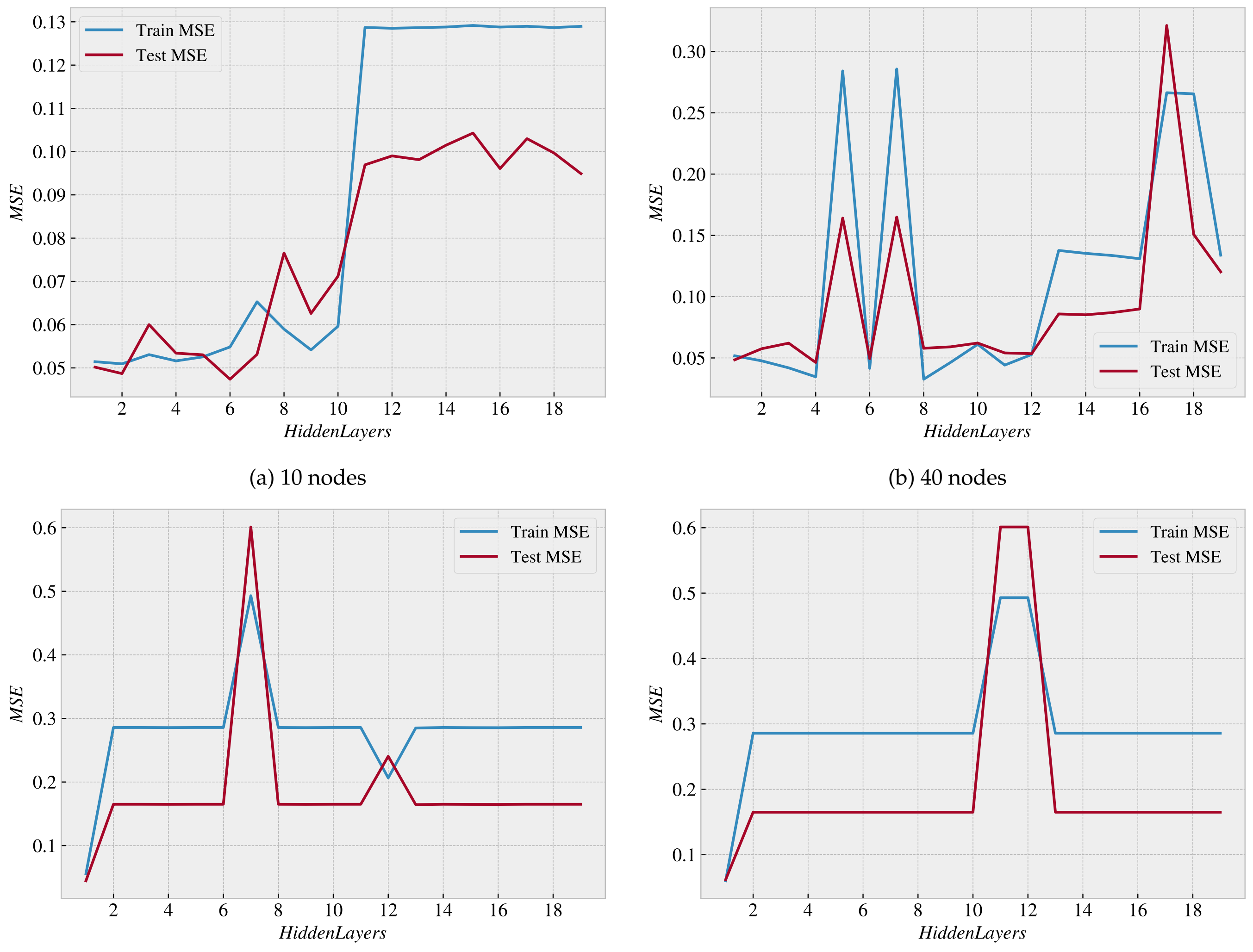


Figure 19: 3D plot of our approximation with hyperparameters $\lambda = 10^{-4}$ and $\eta = 10^{-1}$. Max number of epochs 200, batch size 25, 2 hidden layers and 30 hidden nodes.

In figure 19 and figure 18 we observe that our network makes a very good fit to the noisy data. This is further evidence of good validation of our network, and expected, given the MSE and R2 score from table IV.

Next we study the MSE on training and test data as a function of hidden layers in order to check whether we can get the network regression to yield overfitting. We do this repeatedly for $10, 40, 70, 100$ nodes per hidden layer. The results is shown in figure 20.



(a) 10 nodes (b) 40 nodes

(c) 70 nodes (d) 100 nodes

Figure 20: MSE as function of hidden layers for both training and test data. Top left corner has 10 nodes, top right has 40 nodes, bottom left has 70 nodes and bottom right has 100 nodes. We also have $\eta = 0.1$, $\lambda = 10^{-4}$, batch size $= 25$ and at most 400 epochs.

From 20 we do not observe any clear signs of overfitting as the difference between the training and test MSE does not diverge for increasing number of hidden layers.

B. Analysis of activation functions

1. *Franke's function*

For Franke's function data we want to test the three different activation functions: Sigmoid, RELU and Leaky RELU. We do so by choosing all other parameters arbitrarily, and observe how the training MSE varies as a function of number of epochs. The calculations were done with a network containing 2 hidden layers, 30 nodes for each hidden layer and a batch size of 25. γ and λ are both set to 0 and η was set to 0.01. The results are shown in figure 21.

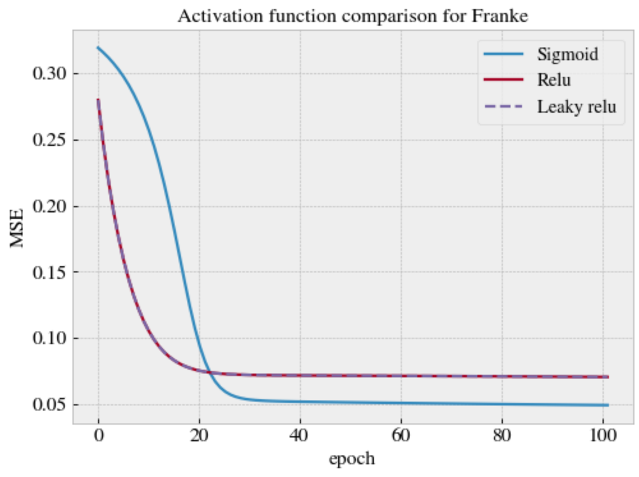


Figure 21: Comparison between sigmoid (blue), RELU (red) and leaky RELU (purple) as activation functions for calculations on Franke's function data. The calculations were done with 2 hidden layers with 30 nodes each. γ and λ were both set to 0 and η was set to 0.01.

From figure 21 we observe that RELU and leaky RELU result in an equal performance. This is an indication that the unactiavted values going into the activation functions are always larger than zero. In addition we observe that sigmoid seems to produce larger MSE-values in the first 30 epochs, but stabilizes at a smaller MSE value further into the learning evolution. This means that for the regression case sigmoid gives the better regression performance compared to RELU and leaky RELU, given enough epochs.

2. *Breast cancer data*

As done in subsection IV B 1, we want to compare the different activation functions, but this time for the classification problem given by the breast cancer data. All network parameters are put to the same as in the previous comparison. Given that this is a classifying problem, the comparison is done with the accuracy score. The results is shown in figure 22.

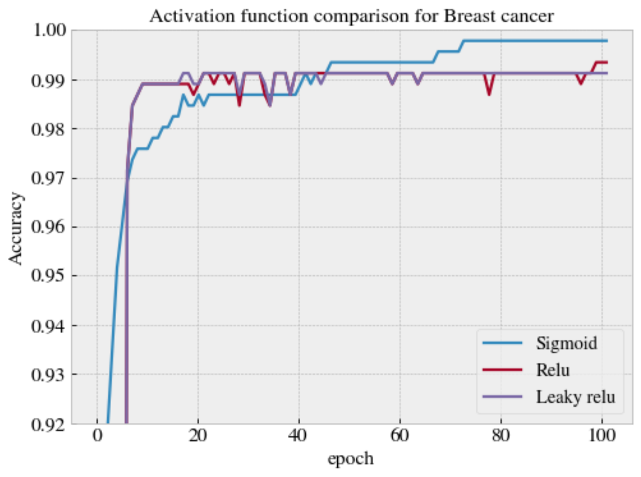


Figure 22: Comparison between sigmoid (blue), RELU (red) and leaky RELU (purple) as activation functions. The calculations were done with 2 hidden layers with 30 nodes each. γ and λ values were both set to 0 and η was set to 0.01.

From figure 22, we observe again that sigmoid gives a small accuracy benefit over RELU and leaky RELU, after the first 30 epochs as. In addition we observe that RELU and leaky RELU produce rather similar accuracy, but not completely equal as we saw in the regression case. For the classification case the general trend of the accuracy produced from RELU is slightly more unstable than that produced from leaky RELU when judging from the dips in the accuracy curve.

C. Breast cancer classification

1. *Hyperparameter optimization*

In the previous section we have studied the breast cancer classification for different activation functions using an arbitrary choice of hyperparameters. Now we want to optimize the hyperparameters to investigate the best accuracy possible. This achieved by first performing a grid serach for the hyperparameters λ and η. From the previous study of the activation functions we found the sigmoid to give the best results, and thus we use sigmoid as an activation function for all layers. For the first grid search we use 2 hidden layers, 30 nodes per hidden layer, a max number of 200 epochs and a batch size of 25. The result is shown in figure 23.

The first grid search shown in figure 23 indicates that the ideal hyperparameters are $= 10^{-}2$ and $\eta = 10^{-2}$ after running each network for a maximum of 200 epochs. By choosing the above values as fixed hyperparameters we perform a second grid search for the optimal number of hidden layers and number of nodes per hidden layer using the above. This is shown in figure 24. From figure 24 we find three candidates for the best network

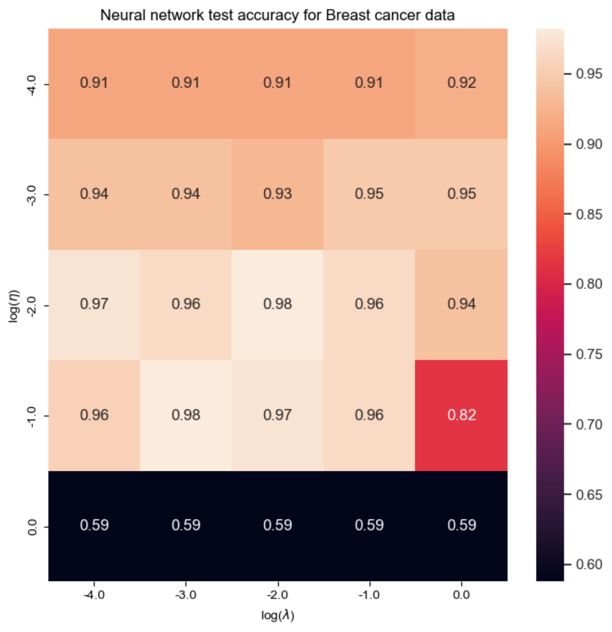


Figure 23: Heatmap for hyperparameters $\lambda \in [10^{-4}, 1]$ and $\eta \in [10^{-4}, 1]$. Number of epochs 200, batch size 25, 2 hidden layers and 30 hidden nodes.

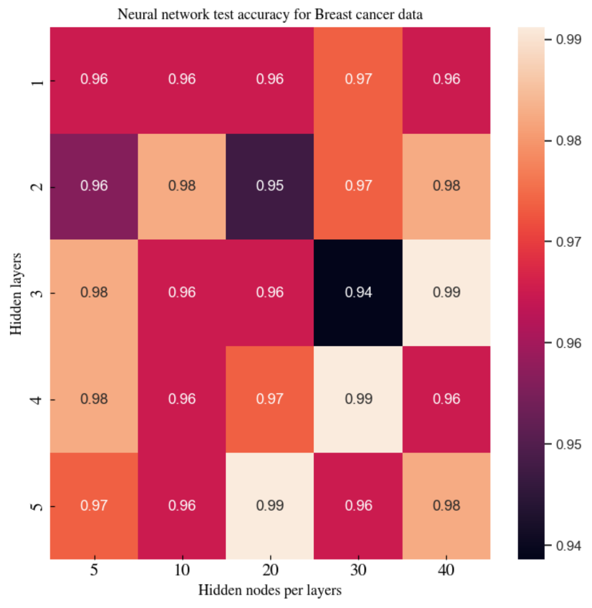


Figure 24: Heatmap for different values of hidden layers and nodes per layer. The calcluation were done with $\eta = 0.01$, $\gamma = 0$, batch size 25 and run for a maximum of 50 epochs.

architecture given as

$$\begin{pmatrix} \text{\# Hidden layers} \\ \text{\# Hidden nodes} \end{pmatrix} = \begin{pmatrix} 5 \\ 20 \end{pmatrix}, \begin{pmatrix} 4 \\ 30 \end{pmatrix}, \begin{pmatrix} 3 \\ 40 \end{pmatrix},$$

which creates a diagonal strip in the bottom right corner of the heatmap. As the total number of weights and biases can be viewed as a measure of the complexity in the network, this can be seen as a preference to a certain level of complexity. We choose the candidate with 3 hidden layers and 40 nodes per hidden layer as the optimal architecture since this minimizes the number of layers needed.

Using the optimal parameters found from figure 23 and 24 for both our network and the network created by Scikit learn, we find the final test accuracy displayed in table V.

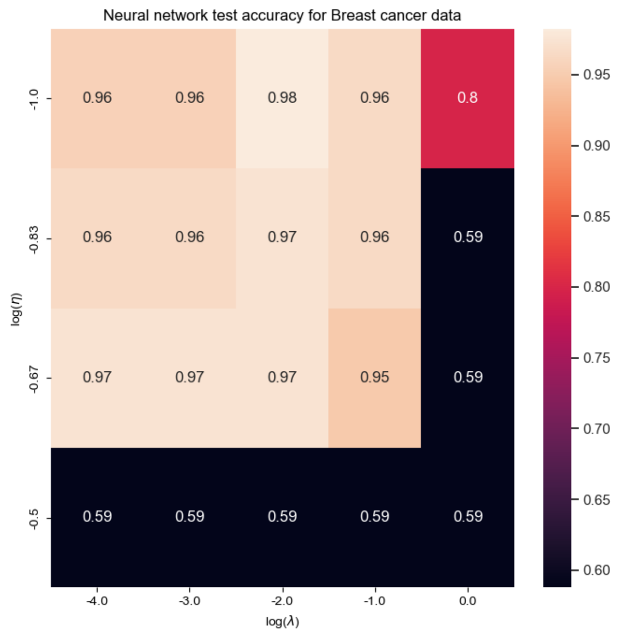
Table V: Comparison between our neural network and Scikit-Learn with hyperparameters $\lambda = 10^{-2}$ and $\eta = 10^{-2}$. Both with 3 hidden layers with 40 nodes each, batch size equal to 25 and with a maximum of 50 epochs.

	Neural network	**Scikit-Learn**
Accuracy	99,12 %	96,49 %

Table V shows that our network produces outperforms Scikit-Learns by roughly 3%. Similar to the comparison of our neural network against TensorFlow for the regression case in section IV A 2, we can explain the difference in score to the difference in the initialization of weights and biases between our model and then one from Scikit-Learn. If we have done hyperparameter optimization for Scikit-Learn as well, we would expect the final scores to be equally good.

2. *Adaptive learning rate*

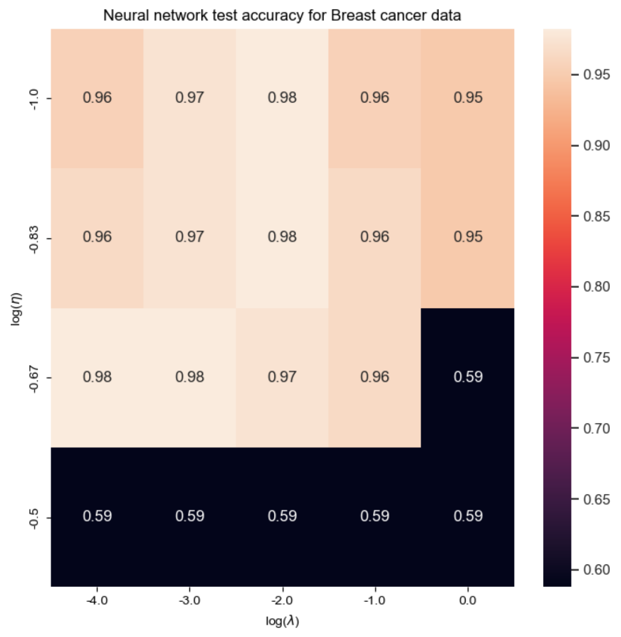
In the previous study of the breast cancer data we have used a constant learning rate, but as mentioned in section III E 1 we might get improved results by introducing an adaptive learning rate. For all of the cases so far we have chosen relatively small learning rates and large number of epochs. But, for larger sets of data one might be computationally limited to a small number epochs. Therefore an alternative strategy is do choose a larger learning rate which decreases as a function of epochs. To test out this strategy we limit our self to a max number of 20 epochs. We begin by performing a grid search with relatively large learning rates $\eta \in [10^{-1}, 10^{-0.5}]$ and similar λ's as tested earlier. With a constant learning rate the result becomes as shown in figure 25.



log(η) \ log(λ)	-4.0	-3.0	-2.0	-1.0	0.0
-1.0	0.96	0.96	0.98	0.96	0.8
-0.83	0.96	0.96	0.97	0.96	0.59
-0.67	0.97	0.97	0.97	0.95	0.59
-0.5	0.59	0.59	0.59	0.59	0.59

Figure 25: Heatmap for hyperparameters $\lambda \in [10^{-4}, 1]$ and constant $\eta \in [10^{-1}, 10^{-0.5}]$. Number of epochs 20, batch size 25, 2 hidden layers and 30 hidden nodes.

We repeat the grid search from figure 25 with the adaptive learning rate described in section III E 1 with k = 0.2 and $t_D = 10$ epochs we found the results displayed in figure 26.



log(η) \ log(λ)	-4.0	-3.0	-2.0	-1.0	0.0
-1.0	0.96	0.97	0.98	0.96	0.95
-0.83	0.96	0.97	0.98	0.96	0.95
-0.67	0.98	0.98	0.97	0.96	0.59
-0.5	0.59	0.59	0.59	0.59	0.59

Figure 26: Heatmap for hyperparameters $\lambda \in [10^{-4}, 1]$ and adaptive $\eta \in [10^{-1}, 10^{-0.5}]$. Number of epochs 20, batch size 25, 2 hidden layers and 30 hidden nodes.

By comparing figures 25 and 26 we observe that the introduction of an adaptive learning rate in general gave a slight improvement for the relatively big learning rates, where almost half of the values were improved by 1%. In the cases where both η and λ are large, the adaptive learning rate far exceeds the constant learning rate.

However, when letting the network run for more epochs the difference between constant and adaptive learning rates where negligible for our use, and we did not include in the later studies.

D. MNIST classification

So far we have compared the performance of our neural network to OLS and Ridge regression. In this section we want use the MNIST data to carry out a comparison between our network and logistic regression. Here we compare the two methods ability to recognize low resolution images of hand written digits. As this is a classification problem, we measure the result accuracy as number of correct predictions divided by the total number of data points.

Before we compare the two methods we choose some arbitrary values for the hyperparameters: $\eta = 10^{-1}$, $\lambda = 0$, 2 hidden layers and 30 nodes for each hideen layer, and run the network for a maximum of 20 epochs. The network achieves an accuracy of 91%. Given that a method of simply guessing each digit would achieve an accuracy of $\approx 10\%$, this is a good result given the small number of epochs and non-optimized hyperparameters. The prediction on the test data is showcased in figure 27.

Figure 27: Digits in test data with the neural networks predicted values. Green digits indicate correct prediction, red indicate incorrect. The neural network was trained with 2 hidden layers with 30 nodes each, batch size 50, $\eta = 10^{-4}$, $\lambda = 10^{-3}$ and after 20 epochs.

From figure 27 we can observe that the network has indeed picked up many of the general patterns associated to each digit. By further examination of the wrongly predicted digits, we observe that most of them are somewhat understandable mistakes, such as mistaking the top-curvature of a 2 to the curvature of an 8 or the the bottom tail of a 3 to the tail of a 9.

Next, we perform a grid search for our neural network for the hyperparameters $\lambda \in [10^{-3}, 10^{-1}]$ and $\eta \in [10^{-4}, 10^{-2}]$ with a maximum of 200 epochs. The result is shown in figure 28.

From 28 we observe that the optimal values for our neural network are $\lambda = 10^{-2.33}$ and $\eta = 10^{-2}$.

By doing the same grid search for our logistic regression solver for the hyperparameters $\lambda \in [10^{-3}, 10^{-1}]$ and $\eta \in [10^{-4}, 10^{-3}]$, we find the results shown in figure 29.

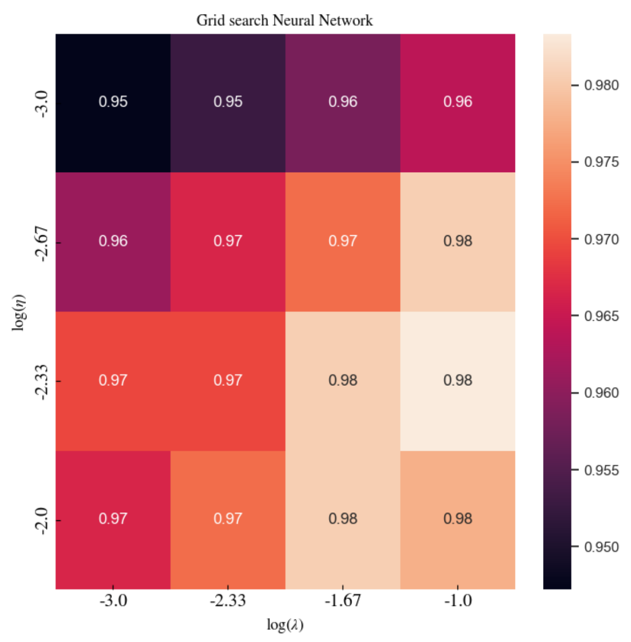


Figure 28: Neural network heatmap for hyperparameters $\lambda \in [10^{-3}, 10^{-1}]$ and $\eta \in [10^{-4}, 10^{-2}]$. Maximum number of epochs 200, batch size 50, 2 hidden layers and 30 nodes per hidden layer.

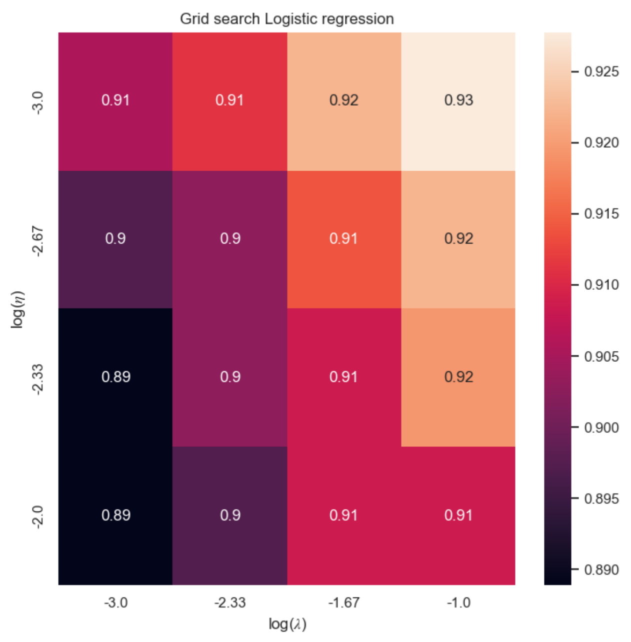


Figure 29: Logistic regression heatmap for hyperparameters $\lambda \in [10^{-3}, 10^{-1}]$ and $\eta \in [10^{-4}, 10^{-3}]$. Number of epochs 200, batch size 50, 2 hidden layers and 30 nodes per hidden layer.

From figure 29 we can read of the ideal hyperparameters for our logistic regression method as $\lambda = 10^{-1}$ and $\eta = 10^{-3}$.

Using the optimal η and λ hyperparameters found in the previous analysis we compare our neural network with our logistic regression solver and Scikit-Learns logistic regression. The comparison results is shown in table VI.

Table VI: Accuracy score for the neural network, scikit learn and logistic regression on the MNIST data set

	Neural network	**Logistic regression**	**Scikit**
Accuracy	97.89%	92.50 %	94.83%

From table VI we see that the neural network gives a better accuracy than the logistic regression method by 3-5%, indicating that the neural network is better suited for classifying the MNIST data. We see that the various logistic regression methods gives a somewhat similar result.

Using the optimal hyperparameters for our neural network we can recreate figure 27 with our optimized model. This is seen in figure 30.

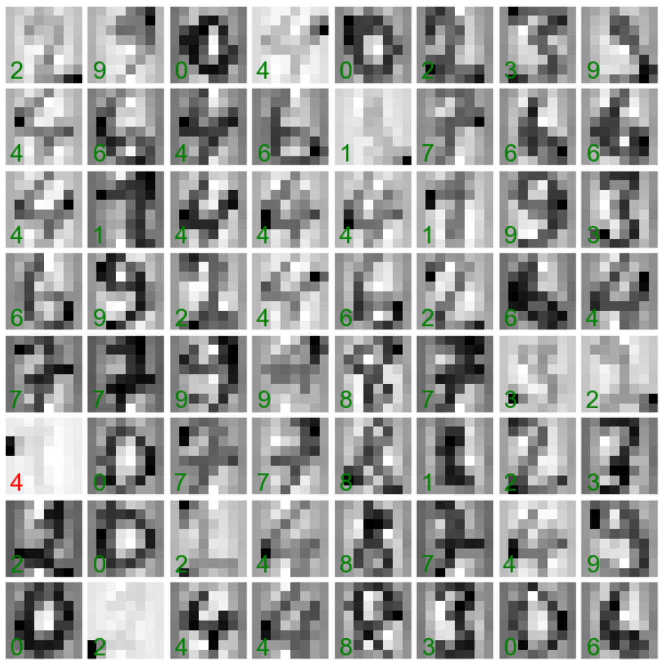


Figure 30: 64 digits from the test data with the neural networks predicted values. Green digits indicate correct prediction, red indicate incorrect. The neural network was trained with 2 hidden layers with 35 nodes each, batch size 50 and after a maximum 200 epochs. The hyperparameters were chosen from a grid search as $\lambda = 10^{-2.33}$ and $\eta = 10^{-2}$.

Comparing figure 27 and 30 we observe that the new parameters greatly improved the networks ability to predict the digits. All but one of the digits were correctly predicted, for which the single wrong classification might be excused since it proves to be difficult for humans to classify as well. Also, this digit slightly deviates from the center of the image, which might make it even harder for the network to classify.

V. CONCLUSION

From our validation analysis of the stochastic gradient descent method we found that the SGD method was able to give a reasonable good fit on the Franke's function data with a best MSE on the test data of 0.048 for 10^3 epochs compared to OLS with 0.042 and Ridge 0.40. When studying the MSE as a function of epochs for Franke's function we found that the variation of batch sizes gave a slightly (almost negligible) benefit in favour of full gradient descent, however a significant speed up when choosing small batch sizes for big data sets of roughly more than 30000 elements in the design matrix. We also found that SGD is much less prone to overfitting compared to OLS, which resulted in a better prediction on the test data by SGD for a model with polynomial degrees larger than 9. Adding momentum to SGD gave generally a better fit in the training data, but for big learning rates it produced initial oscillation in the training MSE.

From the validation analysis of the neural network we found that the neural network was able to accurately predict all three gates with 100 % accuracy after 75 epochs, where as OLS was limited to a 75 % accuracy on the XOR gate. While doing linear regression on the Franke function we found that the optimal hyperparameters were $\eta = 10^{-1}$, $\lambda = 10^{-4}$ and an optimal network architecture of 3 hidden layers, each with 43 hidden nodes. Using these paramters we found that our network outperformed OLS, Ridge and TensorFlow with a MSE score of 0.032 and a R2 score of 0.844.

By testing Sigmoid, Leaky RELU and RELU on both Franke's function and breast cancer data, we found that the Sigmoid activation function produced the best predictions overall given enough epochs ($\approx 25 - 40$). In addition we found that Leaky RELU and RELU produced similar results, where RELU produced slightly more stable solutions in the classification case.

When studying the breast cancer data we found the optimal hyperparameters to be $\eta = \lambda = 10^{-2}$, with an optimal network architecture of 3 hidden layers and 40 hidden nodes. This network was able to achieve get an accuracy of 99.12 %, a 3 % out performance of Scikit-learn.

In the case of small number of maximum epochs and relatively large initial learning rates, the adaptive learning rate slightly improved most combinations of hyperparameters in the interval $\eta \in [10^{-1}, 10^{-0.5}]$. This method also completely outperformed a constant learning rate in the cases where both η and λ were large and the number of epochs was kept low.

When classifying digits from the MNIST Scikit-learn data set, we found that the optimal hyperparameters were $\eta = 10^{-2}$ and $\lambda = 10^{-2.33}$, giving us a accuracy of 98 %. For logistic regression we found that the optimal hyperparameters were $\eta = 10^{-3}$ and $\lambda = 10^{-1}$, giving us a accuracy score of 93 % and Scikit-Learns logistic regression an accuracy of 95%.

REFERENCES

[1] 3B1B. *What is backpropagation really doing? | Chapter 3, Deep learning*. URL: https://www.youtube.com/watch?v=Ilg3gGewQ5U. (accessed: 07.11.2021).

[2] Frette et al. *Project 1 on Machine Learning*. URL: https://github.com/Gadangadang/Fys-Stk4155/blob/main/Project%201/article/Project_1_current.pdf. (accessed: 10.11.2021).

[3] Sarah Dai. *China's subways embrace facial recognition payment system despite rising privacy concerns*. URL: https://www.scmp.com/tech/apps-social/article/3040398/chinas-subways-embrace-facial-recognition-payment-systems-despite. (accessed: 14.11.2021).

[4] DeepMind. *AlphaGo*. URL: https://deepmind.com/research/case-studies/alphago-the-story-so-far. (accessed: 14.11.2021).

[5] Tesla Inc. *Tesla Vehicle Safety Report*. URL: https://www.tesla.com/VehicleSafetyReport. (accessed: 25.10.2021).

[6] Kilian Q. Weinberger. *Gradient Descent (and Beyond)*. URL: http://www.cs.cornell.edu/courses/cs4780/2015fa/web/lecturenotes/lecturenote07.html. (accessed: 14.11.2021).