Machine Learning: Using regression and neural networks to fit continuous functions and classify data

Håkon Olav Torvik, Vetle Vikenes & Sigurd Sørlie Rustad

FYS-STK4155 – Applied Data Analysis and Machine Learning
Autumn 2021
Department of Physics
University of Oslo

November 16, 2021

Abstract: Abstract coming soon.

Contents

1 Introduction

In the modern world, digital data has become one of the most valuable commodities there is. Not because of scarcity, like most other valuables, but rather the exact opposite; the vast abundance of data available makes being able to understand trends and patterns in it extremely valuable for companies looking for profit. However, data is complex, having many features, and understanding how one affect another can be impossible with human analysis alone. Luckily, there exists statistical methods that let us find the deeper connections, make models and even predict outcomes. In this paper we wish to study some stochastic methods, and look at their limitations and strengths.

First we will study a bi-variate continuous function known as the Franke function. We will use both stochastic gradient descent and a feed-forward deep neural network, with back propagation. Then we can also compare results with those obtained in a previous paper, using the deterministic methods ordinary least squares and ridge regression. Note that all methods used in this report is briefly covered in the theory section.

Next we will embark on a classification problem, using measurments of tumors in breast tissue to predict wether they are benign or malignant. Here we will again use a feed-forward deep neural network, along with logistic regression.

We will in no way answer all questions linked to the aforementioned methods. So that anyone can reproduce or continue our studies, we list all the code, results and instructions on running the code in our GitHub repository[1].

2 Theory

In the theory-section we aim to give a brief explanation of the main concepts and terminology used in this report. For a more in-depth explanation we recommend reading the appropriate sections in [3], which has been of great inspiration and help for us throughout the project.

In general, we have a dataset $\boldsymbol{x}$, where each point $\boldsymbol{x}_i$ takes a value y_i, for which we want to make a model $\boldsymbol{\beta}$, such that for a new data point $\boldsymbol{x}_k \notin \boldsymbol{x}$ we can make a prediction for the value y_k. The model $\boldsymbol{\beta}$ is a vector/matrix, where each element is a parameter of our model, such that $\boldsymbol{\beta}$ is sometimes called the parameters. For gradient descent, we have to choose what shape the model should be, as was done for linear regression in [4], while the neural network makes its own model.

2.1 Gradient Descent

In this section we cover gradient descent and different variations of it. More specifically we describe gradient descent (GD), stochastic gradient descent (SGD) and adding momentum to the aforementioned methods. All gradient descent methods start with an initial guess for what the model $\boldsymbol{\beta}$ should be, and iteratively updates the guess by training on the dataset, either until it reaches a minimum, or a certain number of iterations have been performed.

[1] https://github.com/sigurdru/FYS-STK4155/tree/main/project2

2.1.1 Ordinary Gradient Descent

Gradient descent methods is often used to minimize the so-called cost/loss-function, which tells us how good our model at predicting the dataset is (more on this in section 2.3.3). For now, we use a general cost function $C(\boldsymbol{\beta})$ for a given model $\boldsymbol{\beta}$, which can be expressed as the sum over the cost function for each datapoint $\boldsymbol{x}_i$, as such:

$$C(\boldsymbol{\beta}) = \sum_{i=1}^{n} c_i(\mathbf{x}_i, \boldsymbol{\beta}), \tag{2.1}$$

where n denotes the number of datapoints. The gradient with respect to the parameters $\boldsymbol{\beta}$, which represent the direction of optimal minimization of the cost function, is then defined as

$$\nabla_{\beta} C(\boldsymbol{\beta}) = \sum_{i=1}^{n} \nabla_{\beta} c(\mathbf{x}_i, \boldsymbol{\beta}). \tag{2.2}$$

The algorithm for GD is then:

$$\begin{aligned} \mathbf{v}_t &= \eta \nabla_{\beta} C(\boldsymbol{\beta}_t) \\ \boldsymbol{\beta}_{t+1} &= \boldsymbol{\beta}_t - \mathbf{v}_t, \end{aligned} \tag{2.3}$$

where η is what we call the learning rate, representing the step-length to move in the optimal direction. This algorithms iteratively finds a new $\boldsymbol{\beta}_{t+1}$ which (ideally) decreases the cost function. This is of course not always the case, and depends on the value of η.

For a model with p parameters, the cost-function is the surface of a p-dimensional hypersurface, and minimizing this can lead to several problems. For example, if η is too big, the cost-function can diverge and never find a minimum of the hypersurface, while if η is too small we will need too many iterations to reach a minimum in reasonable time. One method of avoiding the cost-function diverging, is using a dynamic learning schedule, where the learning rate η decreases during training. The model then makes larger steps in the beginning, and then smaller and smaller, such that we should be able to converge to a minimum, and not making too big steps, circling around it.

An additional problem is that the hypersurface is not a smooth terrain with a single minimum. Our model can potentially move down into a local minimum, which can be close to the level of the global minimum, or far worse than it. When our model converges, we have no way of knowing if we have found the optimal, global minium, or are stuck in one of the many local minima, with no way of getting out.

2.1.2 Stochastic Gradient Descent

With large datasets, a large number of computations is needed when calculating the gradient. It takes a lot of time, and the model is only updated once per iteration, making improvement slow. Stochastic Gradient Descent. SGD, combats this by approximating the total gradient (2.2). This is done by performing gradient descent on a subset of the data, called a minibatch. With n still denoting the total number of datapoints, we will

have $N_B = n/M$ minibatches, where M is the size of each minibatch. The minibatches are denoted by B_k. Thus our approximated gradient for a single minibatch B_k is defined as

$$\nabla_{\boldsymbol{\beta}} C^{MB}(\boldsymbol{\beta}) \equiv \sum_{i \in B_k}^{M} \nabla_{\boldsymbol{\beta}} c(\mathbf{x}_i, \boldsymbol{\beta}). \tag{2.4}$$

Then the aim is to use this approximated gradient, for all N_B minibatches, to update the parameters $\boldsymbol{\beta}$, at every step k. Doing this for all N_B minibatches, are what we refer to as an epoch. The SGD algorithm then becomes very similar to (2.3), however with an approximated gradient.

$$\begin{aligned} \mathbf{v}_t &= \eta \nabla_{\boldsymbol{\beta}} C^{MB}(\boldsymbol{\beta}_t) \\ \boldsymbol{\beta}_{t+1} &= \boldsymbol{\beta}_t - \mathbf{v}_t \end{aligned} \tag{2.5}$$

Choosing a smaller M gives a worse approximation of the full gradient, though at the same time the model will have more chances to move in the correct direction. Further, by choosing M such that N_B is neither too small or large, our program will run faster. This not only speeds up our program, it also helps prevent getting stuck in local minima because of the stochastic nature. The dataset is shuffled after each epoch, creating new minibatches such that we never use the same one twice.

2.1.3 Adding Momentum

These methods can still be optimized further by adding momentum. This is done by adding a term to the parameter $\mathbf{v}_t$ in equations (2.3) and (2.5). This so-called mass term, simulates the gradient having momentum, such that every update of $\boldsymbol{\beta}$ is a running average.

$$\mathbf{v}_t = \eta \nabla_{\boldsymbol{\beta}_t} C(\boldsymbol{\beta}) \rightarrow \mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta}_t). \tag{2.6}$$

Here $\gamma \in [0, 1]$ is what we could refer to as the mass, and is a free parameter. One of the benefits is for example that this lets us move faster in regions where the gradient is small.

2.2 Logistic Regression

Logistic Regression is used for classification problems, meaning that we want to predict discrete outputs, for instance true or false, given a set of information about a subject. This is the case for binary classification, where we only have one output. It is also possible to use multi-class classification, where one has several outputs, each representing the probability of the class being given by that output. In our case we want a model that takes in some values $\mathbf{x}_i$ and spits out zero or one. These values should correlate with the actual classification $y_i \in \{0, 1\}$ corresponding to true or false respectively. Lets define our model as

$$\sigma(s_i), \quad \text{where } s_i = \boldsymbol{x}_i^T \boldsymbol{w} + b_0 \equiv \boldsymbol{X}_i^T \boldsymbol{W}, \tag{2.7}$$

where $\boldsymbol{x}_i^T$ is our input data, $\boldsymbol{w}$ and b_0 are parameters in the model. As a shorthand we also defined $\boldsymbol{W} = (b_0, \boldsymbol{w})$ and $\boldsymbol{X}_i = (1, \boldsymbol{x}_i)$. We also have σ which is some soft classifier

that maps our output between zero and one (e.g. the sigmoid (2.11)). The reason why we want a soft classifier and not a hard one (like $\sigma = 1$ if $s \geq 0$ and 0 otherwise), is because then we can interpret the output as a probability, quantifying how certain the model is in its prediction. Here we also need a cost function to minimize. It is common to choose the cross entropy, which we derive in 2.3.3. However we will just use it without derivation for now. The cross entropy for this model is given as

$$C(\boldsymbol{W}) = \sum_{i=1}^{n} -y_i \log \sigma(\boldsymbol{X}_i^T \boldsymbol{W}) - (1 - y_i) \log \left[1 - \sigma(\boldsymbol{X}_i^T \boldsymbol{W})\right], \tag{2.8}$$

where n are the number of samples we want to classify, and y_i the true classification. Now with a cost function and model in hand we are ready to minimize the cost function in order to find the optimal parameters for the model. The gradient of the cost function is given by

$$\nabla_{\boldsymbol{W}} C(\boldsymbol{W}) = \sum_{i=1}^{n} \left[\sigma(\boldsymbol{X}_i^T \boldsymbol{W} - y_i)\right] \boldsymbol{X}_i. \tag{2.9}$$

Thus the only thing left to do is perform an algorithm similar to 2.5, where the parameters to update are $\boldsymbol{W}$.

2.3 Feed-Forward Deep Neural Networks

Neural networks are neural-inspired nonlinear models, which are taught by a way of learning. We will in this section explain what we mean by non-linearity, the basic architecture of a neural network and how the network *learns*. In this paper, only supervised learning will be studied, where we train our model on fully labeled data, as opposed to unsupervised learning, where the model first is shown unlabeled data.

2.3.1 Architecture of Neural Networks

The structure of the neural net we will use in this report is similar to that in figure 1. The gray circles are what we refer to as nodes, and are organized into layers. They hold some numerical value. The first and last layer are the input and output layer, respectively. The rest are so-called hidden layers, denoted h_i. In figure 1 there are three hidden layers. Between each layer is a set of weights, connecting each node in the preceding layer to each node in the succeeding one. One initializes the network by giving the nodes in the input layer numerical values. These values would correspond to some actual physical property, for example brightness of pixels in a picture. Then, the values are fed forward, where the values of the nodes in the next layer being the weighted sum of the values in the previous, plus a bias b. The values are *activated* by an activation function, before similarly being fed forward to the next layer, until reaching the output layer. In the case of regression, the output layer is a single node, giving the functional value. In the case of classification, for example classifying the type of animal in pictures, one node could correspond to a lion, next to a zebra and so fourth. By this we would know what animal the network predicts is pictured by finding the neuron has the highest numerical value.

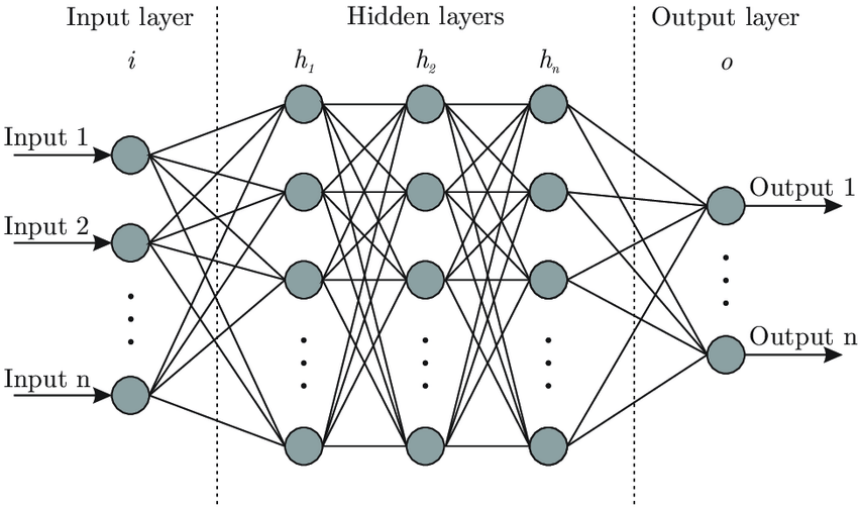

Figure 1. Basic outline of a neural network. It displays the different layers (input, hidden and output), nodes (gray circles) and the connection between the nodes (black lines).

(source: `https://www.researchgate.net/figure/Artificial-neural-network-architecture-ANN-i-h-1-h-2-h-n-o_fig1_321259051`)

We mentioned that the different nodes are connected through weights, biases and activation functions. Looking at figure 1, a neuron j in layer h_1 is connected to n input neurons, denoted by black lines. Each input neuron has a numerical value defined by the problem. The value neuron j in h_1 then gets is defined as

$$a_{h_1,j} = \sigma(x_1 w_1 + x_2 w_2 + \cdots + x_n w_n + b_j), \tag{2.10}$$

where x_i are the values of neuron i in the input layer, w_i are the weights between neurons i and j, b is what we refer to as the bias and σ is the activation function. Every neuron is connected like this, with different weights and biases. Initially the network will make random predictions. Through training, the weights and biases are updated using the backpropagation algorithm, described in section 2.3.4.

2.3.2 Activation Functions

The activation functions are where the non-linearity of the neural nets comes in, because they are non-linear. Now there are many such functions, in our project we have implemented the 3 displayed in figure 2. The exact functions are as follows

$$\text{Sigmoid: } \sigma(x) = \frac{1}{1+e^{-x}}, \tag{2.11}$$

$$\text{RELU: } \sigma(x) = \max(0, x), \tag{2.12}$$

$$\text{Leaky RELU: } \sigma(x) = \begin{cases} \alpha x, & \text{if } x \leq 0 \\ x, & \text{otherwise.} \end{cases} \tag{2.13}$$

For Leaky RELU, α is some parameter which we set to $\alpha = 0.01$. As we will show in section 2.3.4, the derivative of the activation function is used to update the weights and biases of the network. It can happen that inputs to the activation function of nodes are negative, making the derivative of the RELU function zero, such that the contribution from these neurons to the learning of the network vanishes. This is the main benefit of using Leaky RELU, as this won't happen. However, using good weight initialization, this should not

be a problem, so we do not expect the difference between the two activation functions to be very significant.

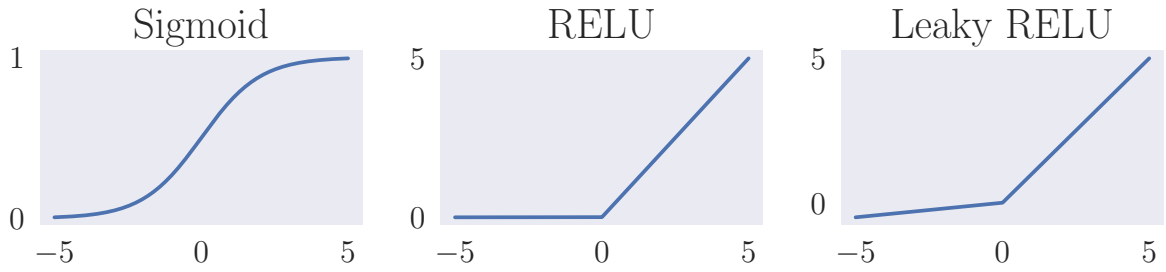

Figure 2. Some activation functions, sigmoid, RELU and Leaky RELU, respectively.

All neurons in all hidden layers are activated by the same activation function. The input layer is just the input, so it is not activated. In the case of regression, the output layer is not calculated with an activation function. Then the predicted function value is just the weighted sum from the last hidden layer. For classification, we want the values in the output nodes to indicate probabilities. Thus, we use an activation function that considers the value in the other nodes in the other output nodes. The softmax function does this

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \tag{2.14}$$

where K is the number of classes, output nodes.

2.3.3 Cost Function and Regularization

Before one can start training the data, we must have a cost function. This will quantify how well or poorly our network is performing, and is what we want to minimize when we train the network. For continuous data it is common to use mean square error (MSE) as the cost function. It is just the difference between desired output ($\hat{\mathbf{x}}$) and actual output ($\mathbf{x}$), squared, averaged over all datapoints $\boldsymbol{x}_i$, as such

$$C(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{\mathbf{x}}_i - \mathbf{x}_i)^2. \tag{2.15}$$

As was done with linear regression, [4] regularization can help prevent overfitting. Common regularization methods are L_1 and L_2 penalties, which add a regularization term to the cost function, as such

$$L_1: \quad C(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{\mathbf{x}}_i - \mathbf{x}_i)^2 + \lambda \sum_j |\mathbf{W}_j|, \tag{2.16}$$

$$L_2: \quad C(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{\mathbf{x}}_i - \mathbf{x}_i)^2 + \lambda \sum_j \mathbf{W}_j^2, \tag{2.17}$$

where λ is some regularization parameter and $\mathbf{W}$ are the weights and biases. We will in this paper study the effects of adding a L_2-regulizer term when updating the parameters,

by adding the term $\eta\lambda \mathbf{W}_{t-1}$ to equation (2.6).

We mentioned when talking about logistic regression (section 2.2), that for classification scenarios one will often use cross-entropy as the cost function. Continuing with the model defined by (2.7), we want to find an appropriate cost function. We define the probability of an outcome y_i given parameters $\boldsymbol{X}_i$ and $\boldsymbol{W}$ as

$$P(y_i = 1|\boldsymbol{X}_i, \boldsymbol{W}) = \frac{1}{1 + \exp(-\boldsymbol{X}_i^T \boldsymbol{W})}, \tag{2.18}$$

$$P(y_i = 0|\boldsymbol{X}_i, \boldsymbol{W}) = 1 - P(y_i = 1|\boldsymbol{X}_i, \boldsymbol{W}) \tag{2.19}$$

We can then map these probabilities to our soft classifier $\sigma(s_i)$

$$P(y_i = 1) = \sigma(s_i) = \sigma(\boldsymbol{X}_i^T \boldsymbol{W}). \tag{2.20}$$

Now we can define the cost function using Maximum Likelihood Estimation (MLE), which states that we should choose parameters that maximize the probability of our given data. Consider the dataset $\mathcal{D}\{(y_i, \boldsymbol{x}_i)\}$, where we remind that $\boldsymbol{x}_i$ are the input parameters. Then the probability of our dataset given $\boldsymbol{W}$ is

$$P(\mathcal{D}|\boldsymbol{W}) = \prod_{i=1}^{n} \left[\sigma(\boldsymbol{X}_i^T \boldsymbol{W})\right]^{y_i} \left[1 - \sigma(\boldsymbol{X}_i^T \boldsymbol{W})\right]^{(1-y_i)}. \tag{2.21}$$

Again we remind that n are the number datapoints we want to classify. This expression is difficult to work with, thus we take the logarithm.

$$l(\boldsymbol{W}) = \log(P(\mathcal{D}|\boldsymbol{W})) = \sum_{i=1}^{n} y_i \log\Big(\sigma(\boldsymbol{X}_i^T \boldsymbol{W})\Big) + (1 - y_i) \log\Big(1 - \sigma(\boldsymbol{X}_i^T \boldsymbol{W})\Big) \tag{2.22}$$

MLE entails finding the $\boldsymbol{W}$ that maximizes $l(\boldsymbol{W})$, or more commonly, minimizes $-l(\boldsymbol{W})$. Thus our cost function becomes

$$C(\boldsymbol{W}) = -l(\boldsymbol{W}) = \sum_{i=1}^{n} -y_i \log \sigma(\boldsymbol{X}_i^T \boldsymbol{W}) - (1 - y_i) \log\left[1 - \sigma(\boldsymbol{X}_i^T \boldsymbol{W})\right], \tag{2.23}$$

which is equation (2.8).

2.3.4 The Backpropagation Algorithm

With a desired cost function we are ready to train the neural network. This is done by the backpropagation algorithm. The method entails finding the derivative of the cost function, with respect to all parameters. When we have a neural network, we have thousands of parameters which can be tuned (weights and biases), meaning that we have to approximate the derivative somehow. The backpropagation algorithm does just that, by exploiting the layered structure displayed in figure 1.

Before we can embark on deriving the algorithm we will introduce some notation. We assume L total layers, indexed as $l = 1, \ldots, L$. Next we need to index the weights, nodes and biases. Let w_{jk}^l be the weight connecting k-th neuron in layer $l - 1$ and j-th neuron

in layer l. The index order in j and k are such that we can do matrix multiplication with index notation later down the road. Further let b_j^l be the bias for the j-th neuron in layer l. Thus the activation of the j-th neuron in layer l (a_j^l) becomes

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) = \sigma(z_j^l), \quad z_j^l \equiv \sum_k w_{jk}^l a_k^{l-1} + b_j^l. \tag{2.24}$$

Here σ is an activation function.

Now the cost function will depend directly on the activation of the output layer (a_j^L). However the activation of the output layer depends on the previous layers, meaning that the cost function depends indirectly on all the previous layers. Lets define define the error Δ_j^L of the j-th neuron in layer L, as the change in cost function with respect to z_j^L.

$$\Delta_j^L \equiv \frac{\partial C}{\partial z_j^L} \tag{2.25}$$

We can similarly define the error of neuron j in layer l, as the change in the cost function with respect to z_j^l,

$$\Delta_j^l \equiv \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l}\frac{d\sigma(z_j^l)}{dz_j^l}. \tag{2.26}$$

In the next few lines we are going to derive several equations needed for the algorithm, it will be apparent why after we have found them. Notice that (2.26) also can be written as

$$\Delta_j^l = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial b_j^l}\frac{\partial b_j^l}{\partial z_j^l} = \frac{\partial C}{\partial b_j^l}. \tag{2.27}$$

Because $\partial b_j^l / \partial z_j^l = 1$ from (2.24). Again using the chain rule we can rewrite (2.26)

$$\begin{aligned}\Delta_j^l = \frac{\partial C}{\partial z_j^l} &= \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \Delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \left(\sum_k \Delta_k^{l+1} w_{kj}^{l+1}\right)\frac{d\sigma(z_j^l)}{dz_j^l}.\end{aligned} \tag{2.28}$$

To find the last equation, we differentiate the cost function with respect to the weight w_{jk}^l

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^j} = \Delta l a_k^{l-1}. \tag{2.29}$$

The equations (2.26), (2.27), (2.28) and (2.29) define what we call the backpropagation algorithm. Then, what exactly is the algorithm? It entails six steps:

1 Activation: First activate the neurons in the activation layer (a_j^1) with desired data.

2 Feedforward: Activate the nodes in following layers, this is done by equation (2.24).

3 Error at layer L: Calculate the error at the last layer using (2.26).

4 Backpropagate error: With (2.28) we can the calculate the error, iterating backwards in the network.

5 Calculate gradient: Find the gradient by using equations (2.27) and (2.29).

6 Update parameters: Update the parameters similarly to (2.5), however $\boldsymbol{\beta}_t$ are our weights and biases in this case.

The expression for updating the weights and biases are

$$w_{jk}^l \leftarrow w_{jk}^l - \eta \Delta_j^l a_k^{l-1} \tag{2.30}$$

$$b_j^l \leftarrow b_j^l - \eta \Delta_j^l \tag{2.31}$$

2.3.5 Numerical gradients

When computing the derivatives involved in the backpropagation algorithm numerically, we will mainly use the autograd module in python. The exception to this is when we deal with classification and use the cross-entropy cost function (equation (2.23)) combined with the softmax activation function (equation (2.14)) for the output layer. It can be shown analytically, that the error, equation (2.26), of the output layer in this case can be written as

$$\Delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{d\sigma(z_j^L)}{dz_j^L} = a_j^L - y_j \tag{2.32}$$

where a_j^L is the j-th node of the output layer, and y_j is the corresponding target value.

2.3.6 Initialization of weights

We mentioned earlier that the network has weights and biases between the layers. These needs to be initialized in some way. The biases are simple to initialize, as they are a single number for every node. These are initialized as a small, non-zero value b_0.

Before 2006, most neural networks were performing quite badly on most tasks, as they did not learn during training. One of the (several) reasons were due to bad initialization of weights. A common way of doing this was using the standard normal distribution $W_{i,j} \sim \mathcal{N}(0, 1)$. The problem with this is that it does not consider the size of the layers. In 2010, it was shown that when using sigmoid as the activation function, Xavier-initialization give better results [1]. This is given as $W_{i,j} \sim \mathcal{U}\left(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right)$, where $\mathcal{U}$ is the uniform distribution, and n is the number of nodes in the preceding layer.

In 2015, He-initialization was shown to work well with ReLU and Leaky ReLU [2]. Here, the weights are initialized using the normal distribution, but with a variance given by $v = 2/(1 + \alpha^2)n$, where again n is the number of nodes in the preceding layer, and α is the parameter of the Leaky ReLU-function. For ReLU, this is 0. This initialization only considers the number of nodes in the preceding layer, though normalized initialization where the number of nodes in the succeeding layer is considered as well could yield better

results. We use the forementioned method to initialize the network for the appropriate activation-function, and will not study other methods of initialization.

3 Methods

As we mentioned in the introduction, we wish to study different ways of fitting two types of datasets. The first is fitting the *continuous* Franke Function (3.1), i.e. regression. We will use both stochastic gradient descent and feed forward neural network to fit the data. Both methods are covered in the theory sections 2.1 and 2.3 respectively.

Next we will embark on an classification problem. Namely predicting if breast tumors are malignant or benign, using the Breast Cancer Wisconsin data set[2]. We will again use our feed forward neural network, as well as logistic regression. The latter is covered in the theory section 2.2.

3.1 Franke Function

$$\begin{aligned} f(x,y) =& \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4}-\frac{(9y-2)^2}{4}\right)+\frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49}-\frac{(9y+1)}{10}\right) \\ &+\frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4}-\frac{(9y-3)}{4}\right)-\frac{1}{5}\exp\left(-(9x-4)^2-(9y-7)^2\right). \end{aligned} \tag{3.1}$$

In [4], we already studied the Franke function using linear regression, specifically OLS and OLS with an L2 and L1 parameter λ, so-called Ridge and Lasso regression. The results from these methods will form the basis for comparing our results using SGD and neural networks. The report, along with the code can be found at our GitHub[3]. In order to have comparable results, we will use the same parameters for the data. Only the methods will be different. In that project we generated the data using $N = 30\times30$ uniformly distributed datapoints in x- and y-direction, respectively. To simulate it being real data, we also added normally distributed noise with mean zero and standard deviation 0.2: $\epsilon \sim \mathcal{N}(0, 0.2)$. We also split the input and target data in the same way as before, using 80 % of the data for training and 20 % of the data for testing. The two splitted data sets are then scaled by subtracting the mean of the relevant training data.

3.1.1 Stochastic Gradient Descent

As in [4], we have to choose a model to fit the data to, when using SGD. The simplest is a bivariate polynomial of degree P, such that our model will have p features. This is the design matrix X used in the previous project. Having obtained good results for OLS using $P = 6$, we use the same polynomial degree for SGD. Writing our own code for implementing SGD, we will analyze the results with MSE (equation (2.15)) as our cost function for various parameters. We include the L_2 regularization term, equation (2.17), in the gradient of the

[2] https://www.kaggle.com/uciml/breast-cancer-wisconsin-data
[3] https://github.com/sigurdru/FYS-STK4155/tree/main/project1

cost function, which gives the gradient corresponding to Ridge regression. The expression for the stochastic gradient is given in equation (3.2)

$$\nabla_{\boldsymbol{\beta}} C^{\mathrm{MB}}(\boldsymbol{\beta}) = 2X^T(X\boldsymbol{\beta} - \boldsymbol{z})/M + 2\lambda\boldsymbol{\beta} \tag{3.2}$$

where OLS regression is obtained by setting $\lambda = 0$. There are multiple parameters to consider for stochastic gradient descent, and the ones we will study are the choice of learning rate η, number of epochs N_e, number of minibatches N_B, regularization parameter λ, and the momentum parameter γ. This gives us a 5-dimensional hyper-parameter space, and we would ideally optimize each parameter to get the lowest possible MSE. This could be done by grid search, where we test a range of values for every parameter. Testing, for instance, 10 values of each parameter at once means that we have to run the algorithm 10^5 times, which makes the process extremely slow. We will therefore simplify our search, only doing grid search over 1 or 2 of the parameters at a time, keeping the rest constant. We then study how each parameter affects the results, independently of the other parameters.

We begin by studying the MSE of the Franke function. For the MSE values we will include the result from both the train and test data in our first simulation only, confirming that overfitting eventually takes place, while focusing on the test MSE in all the remaining simulation, which is the result of interest. The MSE is studied for different values of η as a function of $N_e \in [0, 150]$, setting $\lambda = \gamma = 0$ and $N_B = 20$, i.e. minibatches of size $M = 36$ for the 720 points in the training dataset. We choose 21 evenly spaced values of $\eta \in [0.01, 0.9]$.

We then choose one of the favorable learning rates to study the MSE for minibatch sizes of $M \in [720, 360, 240, 144, 72, 48, 36, 30, 24]$, as a function of epochs, using $N_e \in [0, 150]$ once again. The values of M are chosen such that N_B becomes integers. This does not have to be the case, but will make the results more consistent. Next, using $N_B = 20$ minibatches, we then study the MSE after $N_e = 150$ epochs as a function of η and λ to study the effect of regularization. For this we use 11 linearly distributed values of $\eta \in [0.1, 0.5]$ and 11 uniformly distributed values of $\lambda \in [10^{-5}, 1]$, on a logarithmic scale. We only consider small λ values, since our previous study of the Franke function with linear regression indicated that higher λ yielded poor results [4]. Finally, we use equation (2.6) to study momentum, using 15 evenly spaced values of $\gamma \in [0, 0.7]$ with a fixed learning rate, and plot the MSE over 150 epochs.

We expect the results to be very dependent on the choice of learning rate η. It is important that it is not too small, then we don't converge. Also if it is too large, we might never reach a minima. To combat these effects we use a dynamic learning rate, which starts high, such that the model gets better quick in the beginning, and then decreases as function of epoch, preventing the model from overshooting and circle around the minima in $\boldsymbol{\beta}$-space. We choose the following function for dynamic learning rate, where $t \in [0, N_e]$ denotes the

current epoch.

$$\eta_t = \eta_0 \cdot \left(1 - \frac{t}{N_e}\right) \tag{3.3}$$

It will decrease lineraly from η_0, to $\eta_{N_e} = 0$ at the last epoch. Using (3.3), we rerun the first results, to study the effects of changing η during training.

3.1.2 Feed Forward Neural Network

When using the neural network to fit the Franke function, we use a lot of the same methods as for SGD. A key difference is that instead of iteratively updating a model $\boldsymbol{\beta}$, we now train a network of several layers, each with many nodes. One of the results of this is that we do not have to choose the shape the model will take.

Instead of giving our network the design matrix X for a certain polynomial degree P, we can pass it only the collection of points (x_i, y_i), and let the network adjust the weights and biases accordingly. Since the Franke function is an exponential function we know that it can be approximated as a higher order polynomial, so by using the design matrix as an input we exploit this property such that the network converges faster. Having already fitted the Franke function with a design matrix with linear regression and SGD, we now choose the x_i and y_i values only. Not providing the network with any initial information has the advantage that we get a more rigorous test of the network's performance, since we ensure that the result is not directly reliant on the information in question. Another important motivation for this is that if we were to fit some other data, e.g. terrain data, we may not have any a-priori information regarding the input data. Omitting the design matrix when we train our neural network will thus yield a final model capable of fitting various types of data.

Since we are dealing with a regression problem and we're fitting a continuous function, a natural choice of the cost function is the MSE. For the neural network we will not compute the total MSE of the output layer as we have previously done, but the individual MSE of each output node. This takes into account the error at each individual output neuron when we update the weights with backpropagation.

As with SGD, neural networks have many parameters we have to tune. To reduce the dimensionality of the hyper-parameter space, we choose some to study, namely the learning rate η, number of epochs N_e, regularization parameter λ, momentum parameter γ, and choice of activation function. We will also study the number of hidden layers and number of nodes in these. We do not change the batch size, choosing one based on the results from SGD, nor the weight initialization, using the ones described in section 2.3.6, given the activation function. For the initial biases we choose $b_0 = 0.01$ for their initial values. There are still many parameters to test, but again we are more interested in how the different paramters affect the result independently of the others.

A good indicator for our neural network being implemented correctly is making sure that it learns during training. This we check by plotting the MSE calculated after every epoch. We do this for different η values. We will then also find reasonable choices for the learning

rate. We choose two hidden layers with ten nodes in each, and use the sigmoid activation function. As mentioned, since we are fitting a function, there is no activation function for the output layer. For the initial analysis we again choose $\lambda = \gamma = 0$.

Having initialized the neural network we are now going to train it. Each training iteration begins by randomly shuffling the data and dividing them into minibatches, just as we did when we performed the SGD analysis. Now the feed-forward and backpropagation is performed for each minibatch, updating the weights and biases.

For the first simulation we choose 21 linearly distributed values of $\eta \in [10^{-3}, 0.9]$, and calculate the MSE from the training data and testing data over each epoch. We plot the evolution of the train MSE, in order to get a feel for a suitable range of learning rates. We then repeat this analysis using 21 new η values on an interval where the resulting MSE was low, plotting the test MSE only. Another quantity we plot for this analysis is the R^2 score, given in equation (3.4), where $\boldsymbol{z}$ and $\tilde{\boldsymbol{z}}$ is the target and prediction values, respectively. This is because the R^2-score gives a relative performance, which the MSE does not with our method of scaling the data.

$$R^2(\boldsymbol{z}, \tilde{\boldsymbol{z}}) = 1 - \frac{\sum_{i=0}^{n-1}(z_i - \tilde{z}_i)^2}{\sum_{i=0}^{n-1}(z_i - \bar{z})^2}, \quad \bar{z} = \frac{1}{n}\sum_{i=0}^{n-1} z_i \tag{3.4}$$

We then look at the MSE using dynamic learning rate, keeping the other parameters the same. Next the activation function for the hidden layers is tested using RELU and leaky RELU instead of sigmoid. We also want to study the dependence of different numbers of layers and nodes. There are mutliple options for this, but to keep it simple, we keep the total number of nodes constant at 20, while using 1, 2, 3 and 4 layers, where the nodes are equally divided among the hidden layers. For 3 layers, the middle has 10 nodes, while the other two have 5 nodes each. Lastly, we study the dependence on the regularization parameter λ, choosing 11 good η-values, and 15 values for λ.

3.2 Wisconsin Breast Cancer Data

We will now use the Wisconsin Breast Cancer dataset to predict wether a tumor is benign or malignant, based upon some features of it. This dataset is widely studied in the scientific litterature on machine learning, as it is easily availible, have relatively few datapoints and not too many features, and is a simple dataset to test binary classification algorithms on. The dataset contains measurments of 30 features of a tumor from 569 patients. Of these 30 features, some might be stronger predictors than others of wether a tumor is malignant or benign. Had the dataset been quite a bit larger, and more complex with many more features, our neural network would have taken a long time to train. It would therefore be important to study the correlation matrix. This is a symmetric matrix saying how much 2 features correlates with eachother. If two features corrolate strongly, they are not linearly independent. It is then not nessisary to make the network train on both, and eliminating one can speed up the training without much loss in performance. Because of the small

size if the dataset, our program runs quite quickly without doing this, so we do not spend much time studying the correlation matrix.

We split and scale the data in the same way we did for the Franke Function, using 20 % as test data. Because the features can have widely different values, it is important to scale them so that the features exist in the same domain. We subtract the mean and divide by the standard deviation. The splitting gives us 455 data points in the train data, and 114 in the test data. Since we now classify data, we no longer use MSE as a performance measurement, rather the accuracy score. It is simply the percentage of correct predictions. Because we only have 114 test data points, the accuracy is fairly quantized, taking values of $n \cdot 0.09\%$, where n is the number of correct predictions. This will later explain the jumps in performance. It must further be noticed that of the 569 tumors in our data set, 63 % are benign, and 37 % are malignent.

3.2.1 Feed Forward Neural Network

We now want to use the same neural network code as before to study and predict the breast cancer data. No changes has to be made, but now we use cross entropy (2.8) as the cost function, and an activation function for the output layer, namely softmax (2.14). We can then interpret the output as probabilities. Because the dataset has binary targets, either 0 or 1, we could have used a single output node. Then sigmoid would have done the same. However, we choose to use 2 output nodes, one for 0 and one for 1, because this generalises to other datasets, which might be multiclass. A test is implemented to make sure that the probabilities sum to 1. When we make the final prediction, the node with the higest probability is picked. More on this in the next section, about logistic regression.

The analysis done with the neural network on the classification data is very similar, as we will study the same hyperparameter dependencies as was done for regression. We will assume the optimal parameters are similar to those found earlier for the neural network.

3.2.2 Logistic Regression

Lastly we want to study the breast cancer data using logistic regression. See theory section 2.2 for an explanation of the method. We use the sigmoid (2.11) as our soft classifier in the output, and cross entropy (2.8) (gradient given by (2.9)) as our cost function. See theory section 2.3.3 for derivation of the cross entropy. To optimize the weights and biases we implement stochastic gradient descent algorithm (2.5), using five minibatches. We will also study the implementation of L_2-regularization (2.17). Our aim is to compare the result with those obtained from out feed forward neural network and Scikit-Learn's logistic regression functionality.

The main quantity we will study is the accuracy score, which is the number of correct classification divided by total cases. The sigmoid is not a hard classifier, i.e. it returns continuous values between zero and one. To calculate the accuracy we will therefore round the answer such that any output less than 0.5 will be set to zero, and above set to one. This means that we loose information of how *confident* our model is when predicting the result. For example if our output is 0.51, then it will set the result to 1, even though that

could be interpreted as inconclusive. Therefore we will also perform a simple analysis in the end on how confident our algorithm is. More precisely we will, for what we interpret as the optimal parameters, look at how many cases are classified within the range $(0.1, 0.9)$. We discuss this a bit more in-depth in the discussion-section.

We initialize the weights and bias with random normal distribution with mean around zero and standard deviation of one. We want to look at what happens to the accuracy as epochs increases, for different learning rates. Therefore we first plot the accuracy score as a function of learning rates and number of epochs. Specifically we will use 200 epochs, and 10 different learning rates η between 10^{-5} and $10^{-0.5}$ distributed evenly on a logarithm scale. We expect the answer to diverge when η is large, and not converge when η is small.

Next we are ready to perform a grid search, to find the optimal learning rate η and L_2 parameter λ, still using 200 epochs. Therefore we again plot the accuracy score, however this time as a function of λ and η, picking out the accuracy score after the last epoch. Specific parameters we will test for are 10 different learning rates η and regularization parameters λ, distributed evenly on a logarithm scale. Learning rate will be between 10^{-5} and 10^{-1}, and λ will be between 10^{-5} and 10^{-3}.

4 Results

4.1 Franke Function

4.1.1 Stochastic Gradient Descent

We plot the MSE evolution of the Franke function using SGD for our train and test data for $\eta \in [0.01, 0.9]$, shown on the top and bottom panel of figure 3 respectively. Note that the MSE can be larger than 0.07 displayed. We have set that as the maximum to better display the nuances. This is done in almost all plots, and is implicit unless specified otherwise.

In figure 3 we see that the test MSE converges faster for increased values of η as expected. For $\eta = 0.01$ we get an MSE barely below 0.07 after $N_e = 150$, as the gradient steps are too small. When η exceeds 0.5 we get abrupt increases of the test MSE after certain epochs. These learning rates are too high, as a gradient step goes beyond the actual minima. For $\eta = 0.9$ this effect is apparent, with large fluctuations of the MSE over certain epochs.

We proceed by plotting the test MSE for $\eta \in [0.01, 0.5]$, shown in figure 4. We clearly see how the increased learning rate yields a faster converging result, but at the expense of MSE stability, which we can see from clear fluctuations at $\eta = 0.5$. When our prediction approaches the desired model, the gradients are relatively small, such that high learning rates potentially overshoots minima. The lowest $MSE = 0.0476$ is achieved with $\eta \approx 0.476$.

When studying different number of minibatches and momentum parameters γ, we will choose a specific η. Although the lowest MSE is with $\eta \approx 0.476$, we deem that the MSE fluctuates too much with that value. Looking at figure 4, although there are many candidates, $\eta = 0.25$ yields fairly quick convergence of the MSE without significant variations.

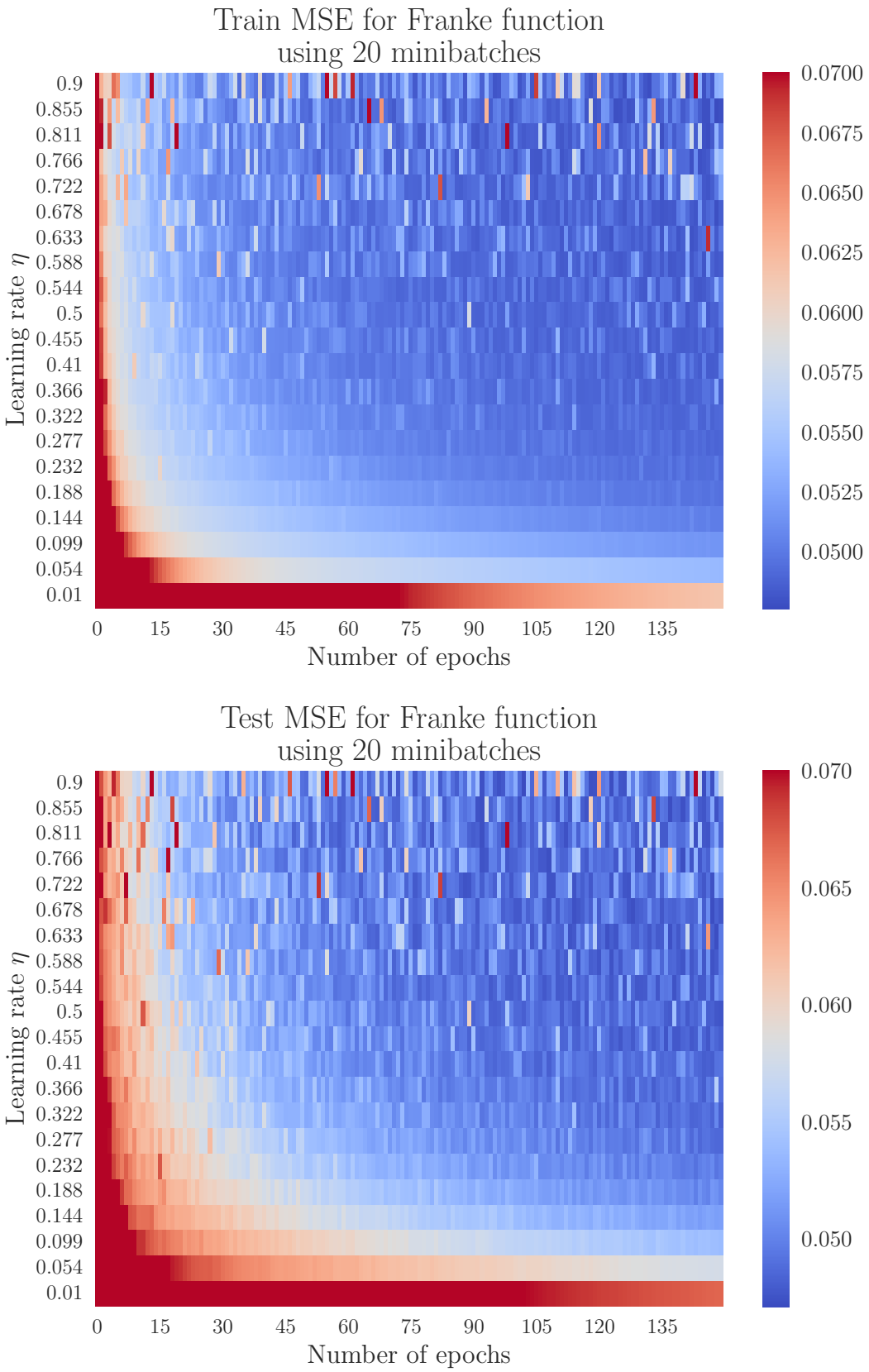

Figure 3. Initial MSE of the train and test data as a function of epochs for different learning rates η. A maximum MSE value of 0.07 is set, so dark red regions may correspond to significantly higher MSE than it appears to.

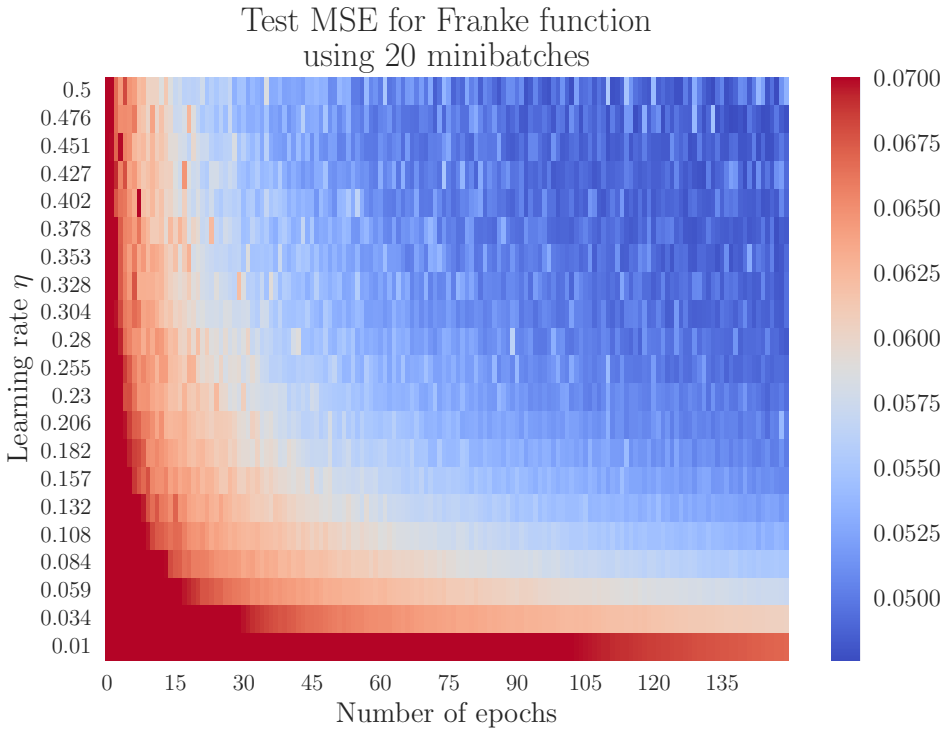

Figure 4. Test MSE for different η values. $\eta < 0.25$ gives a relatively slowly converging MSE, while $\eta > 0.25$ gives fluctuating MSE values. Maximum MSE= 0.07 is chosen.

This choice appears to lie near a boundary between slow convergence and minima overshooting, and when testing other parameters we are more likely to encapsulate both cases at once.

The MSE as a function of epochs for different number of minibatches is shown in figure 5. We notice that by using too few minibatches, we converge more slowly. When we use $N_B = 30$ we get a quickly converging MSE with a low minimum value, but there appears to some fluctuations. Because of the small batch size, the batches are more prone to consist of points not giving a good approximation of the gradient.

We plot the test MSE after $N_e = 150$ epochs with different regularization parameters in figure 6. Using the same η interval as before and $\lambda \leq 1$ we do not set an upper limit on the shown MSE, as we are interested in the general impact of the λ values, and not nuances of the lowest values. We see in the figure that the results are not very affected by the choice of η. However, the MSE is lowest for small values of λ, increasing quickly for higher λ values. This is similar to what we saw with linear regression [4], though we then observed a small dip in the MSE for a range of λ for certain polynomial degrees. The actual lowest $MSE = 0.0482$ is with $\eta = 0.5$ and $\lambda = 10^{-5}$.

Figure 7 shows the resulting MSE values for the test data with different momentum parameters as a function of $N_e \in [0, 150]$. We see that increasing γ values causes the MSE to drop faster, particularly in the beginning when the gradients are large, which is expected. We also notice that $\gamma \gtrsim 0.35$ results in fluctuations of the MSE, especially evident for $\gamma = 0.7$, where it appears that step sizes taken following large gradient values are too big.

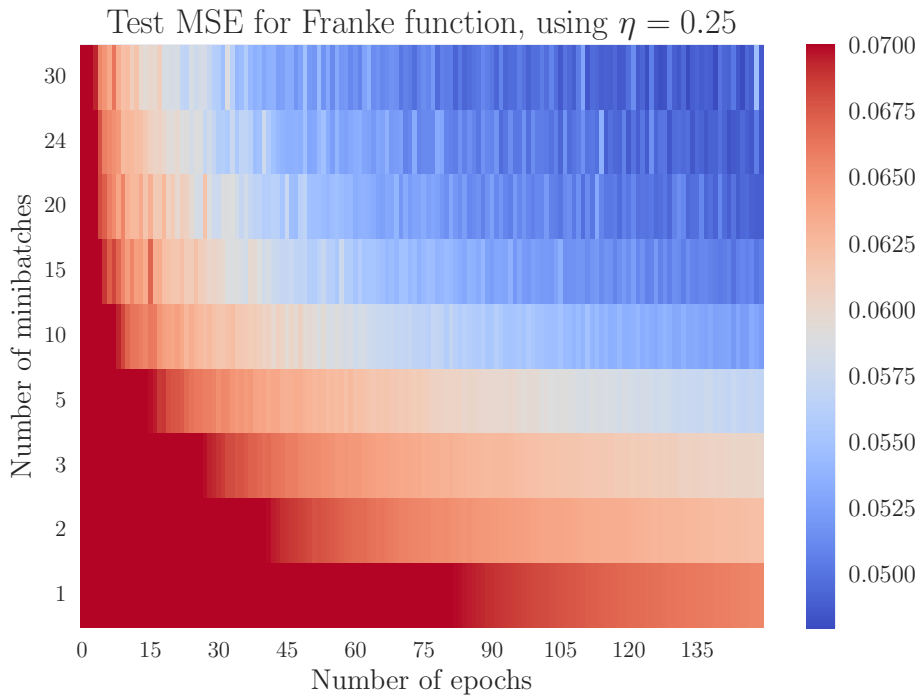

Figure 5. Test MSE for different number of minibatches. The bottom line corresponds to no minibatches overall, and clearly gives the worst result. The heatmap is produced with $\text{MSE}_{\text{max}} = 0.07$.

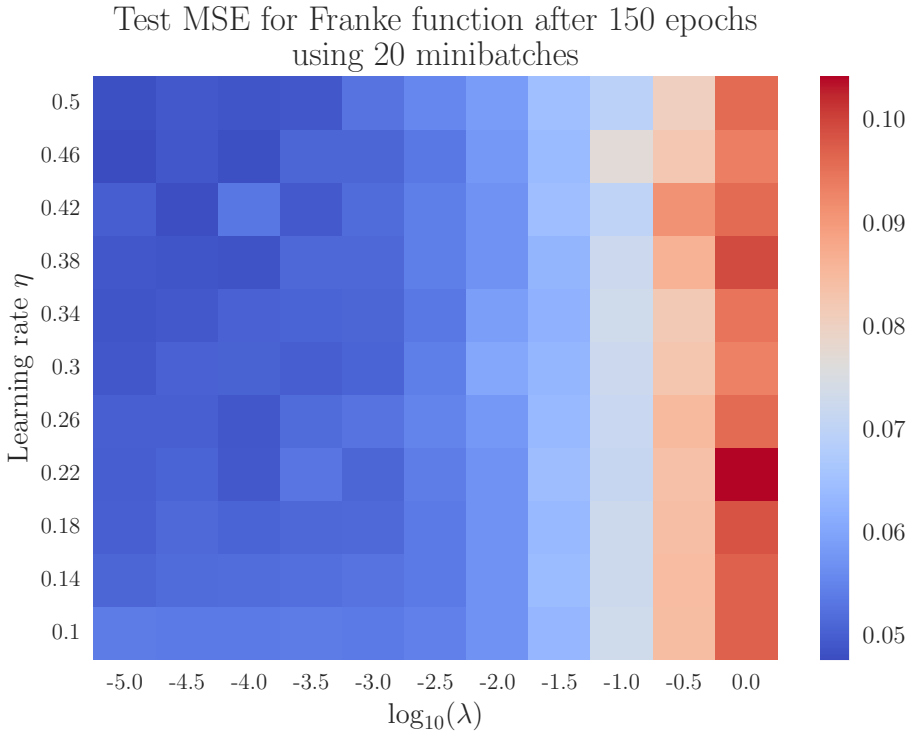

Figure 6. MSE for different values λ, corresponding to Ridge regression, for different η values.

It appears that $\gamma = 0.2$ is the optimal value for this particular test, as it is small enough to avoid overshooting minima for high gradients, while still increasing the convergence rate. Here we actually manage to reproduce the $MSE = 0.0471$, with $\gamma = 0.7$. Which was the

lowest MSE in the previous project [4].

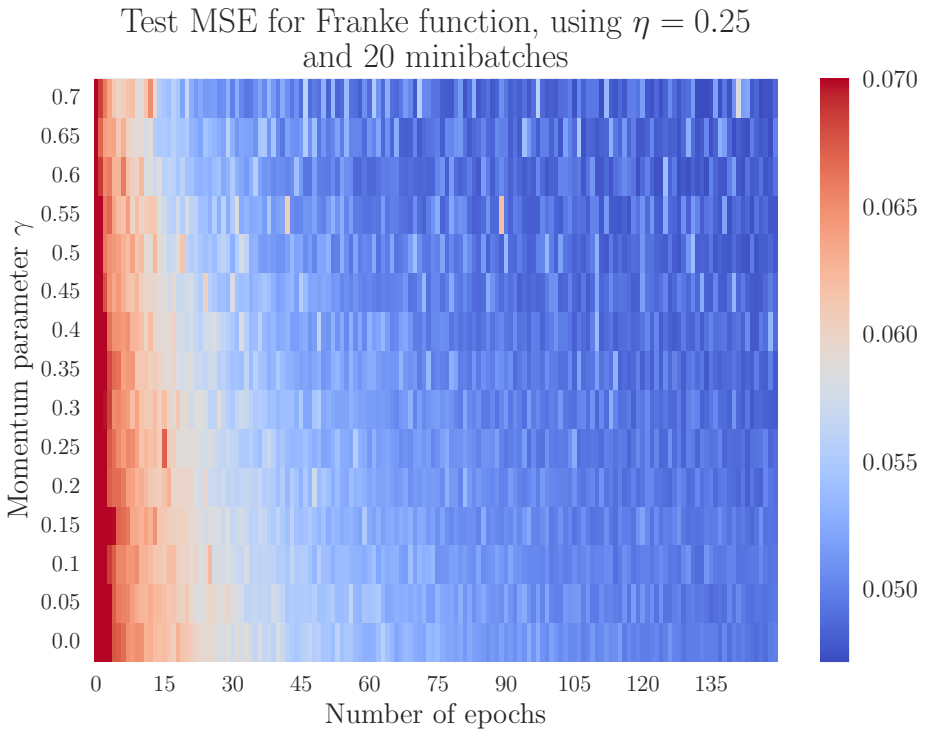

Figure 7. MSE evolution for different momentum parameters, using $\text{MSE}_{\text{max}} = 0.07$, as before. The bottom line corresponds to no momentum.

The MSE calculated with dynamic η is shown in figure 8, using $\eta_0 \in [0.1, 0.9]$. We include the η values previously found to be too high, in order to see whether the scheduled learning gives better results here, while omitting values of $\eta < 0.1$. The figure gives us the expected result that high initial values of η no longer causes abrupt variations of the MSE towards the end of the simulation, as $\eta \to 0$ for $N_e \to 150$. Equation (3.3) appears to be a very good choice for scaling the learning rate when doing SGD, since all values of $\eta_0 > 0.5$ gives decent results after 150 epochs. These solutions converge quickly in the beginning, and as the learning rate drops we get very stable evolutions of the MSE, with no apparent instability in MSE. It appears however that the learning rates become so small towards the end that there is little evolution of the MSE after 120 epochs, meaning that we might not actually have reached an optimal MSE score if the small learning rates near the end have effectively halted convergence. Further, the dynamic learning schedule gives more or less identical MSE values in the end for $\eta_0 > 0.5$. The advantage of this is that our model is less dependent on very specific parameter choices for η. If however, the final result is inadequate, adjusting the initial learning rate will no longer have a desired impact. More training epochs is then required to get optimal results. Here we also managed to get $MSE = 0.0471$ for $\eta = 0.9$.

4.1.2 Feed Forward Neural Network

The train MSE of the Franke function for different learning rates as a function of epochs is plotted and shown in figure 9. We do not cap the upper limit of the MSE. We get a clear indication of good choices of η, regarding where it is too high or too low. The

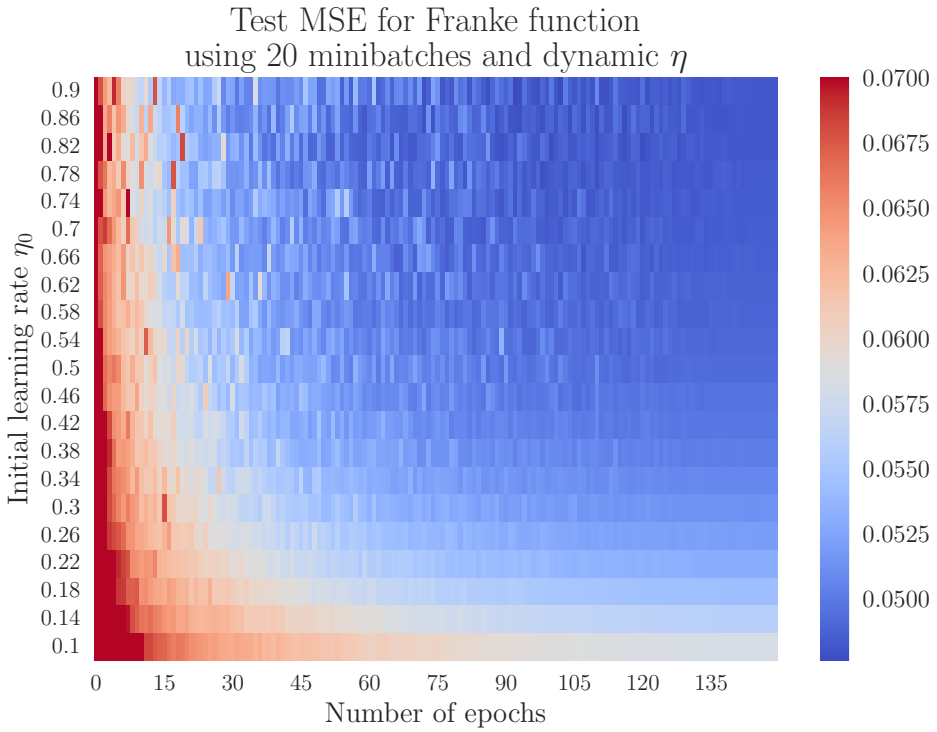

Figure 8. Evolution of the MSE for different values of η_0, all of them decreasing linearly to a final value of $\eta = 0$ after 150 epochs.

corresponding plot for the test data is not shown, but was very similar, with a slightly higher MSE with a maximum value of 0.4, indicating overfitting. We will in the following figures cap the MSE at 0.07, like what was done for SGD.

We choose $\eta \in [0.05, 0.5]$ and plot the test MSE as a function of $N_e = 250$ epochs shown in figure 10. The figure shows how the MSE drops as we continue to train the network, with little evidence of overfitting, which would be a general increases of the MSE towards the end of training. We do however see some clear vertical lines; abrupt increases of the MSE for all η. These were not expected, though can possibly be explained through the way the network is trained. In order to keep results comparable over many runs, the seed of the random distribution is set before each net is created. This was mainly to ensure that the weights are initialized the same for all. However, a side effect of this is that the dataset is split into the same batches for each net, at a given epoch. Sometimes these batches lead to a bad approximation of gradient for the enitre dataset, giving worse test results. Here that happens at the same epoch every time. The lowest MSE at end of training is 0.0443, for $\eta = 0.2075$.

We proceed by plotting the R^2 score, setting a minimum value of $R^2 = 0.55$ to highlight results in the same vicinity of what we got for linear regression in [4], shown in figure 11. While in the results for MSE, red indicated bad values as we want to minimize MSE, for R^2, red is good, as we want it maximized. The figure shows that we obtain R^2 scores exceeding the ones we obtained with linear regression, where the test $R^2_{\text{test}} < 0.6$ was obtained without resampling, while we exceed this value for multiple configurations of the neural network. Other than a relative score, the R^2 score does not appear to provide any

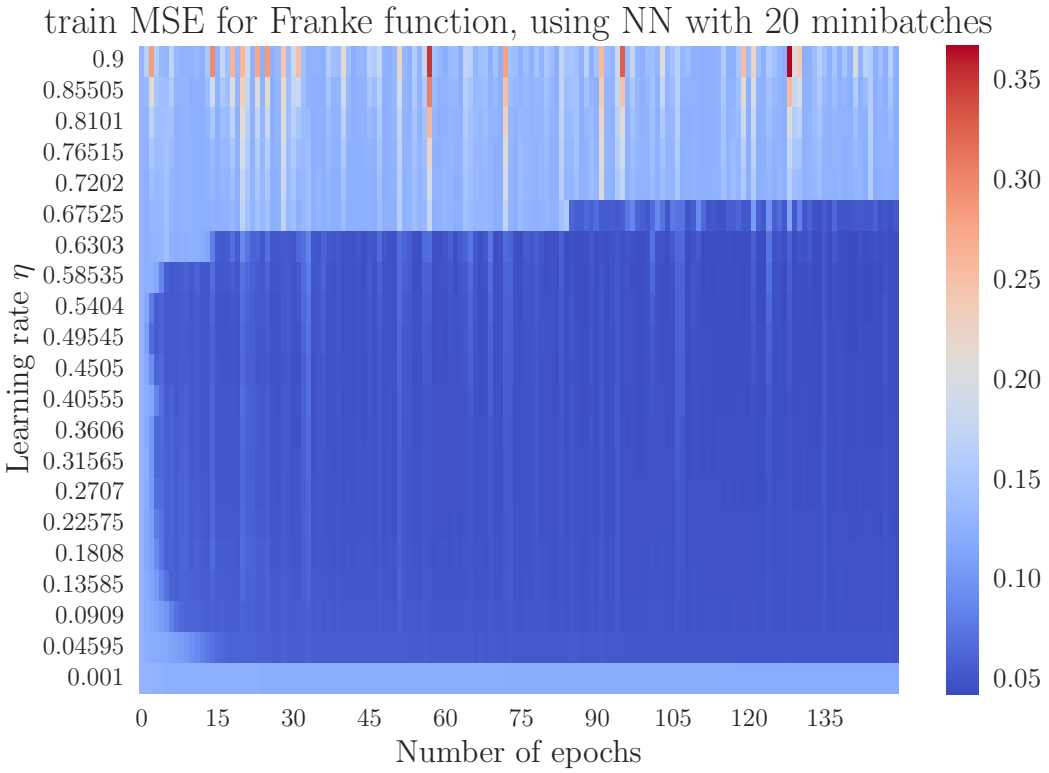

Figure 9. MSE of the train data during 150 epochs for different learning rates using a Neural network.

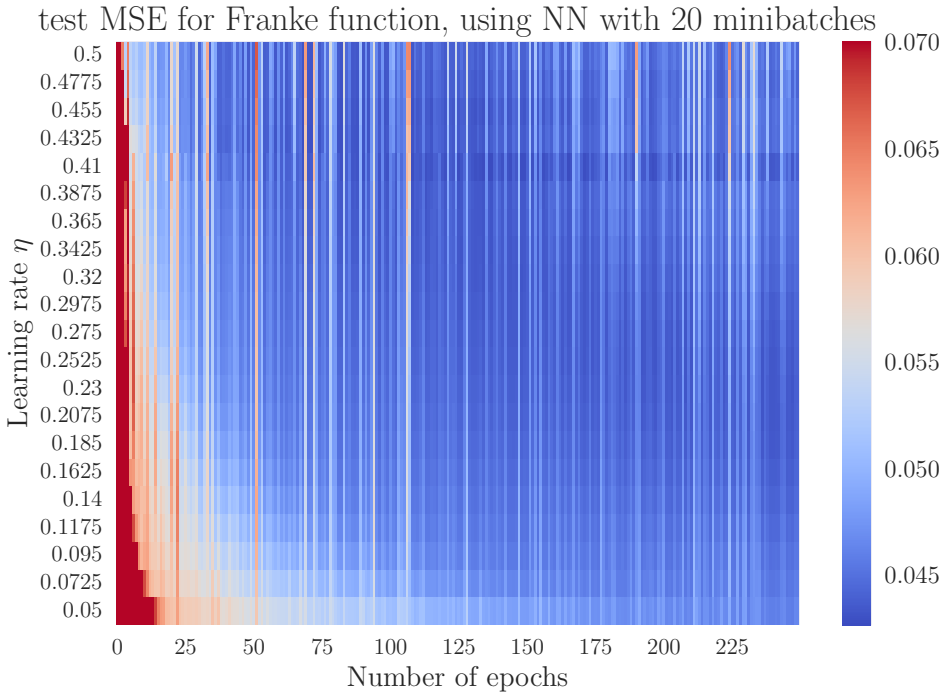

Figure 10. Test MSE of the Franke function for different learning rates, during training.

new information, as we got similar MSE values using linear regression. Thus, the R^2 figures are essentially the negatives of the MSE figures, and will not be used for further assessing

our model.

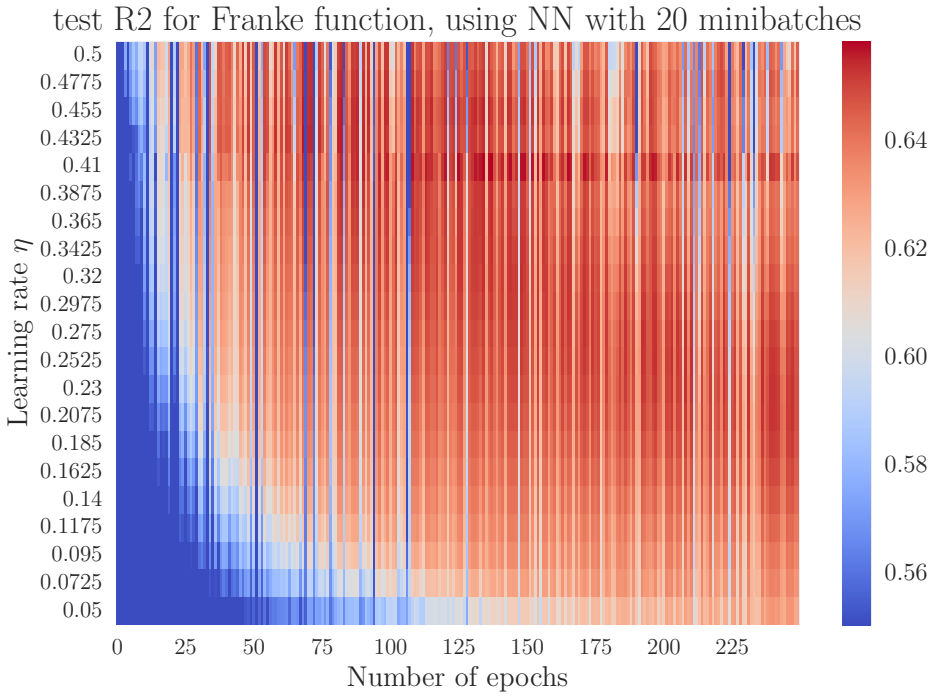

Figure 11. The R^2 score of the Neural network, using a minimum value of the heatmap of $R^2_{\text{min}} = 0.55$.

Figure 12 shows the evolution of the MSE when we use dynamic learning rates, calculated with equation (3.3) as before. The result is similar to what we obtained with SGD, with suppressed fluctuations of the MSE for higher epochs. This has a fairly large effect on the Neural network, as the fluctuations of the MSE for $N_e > 200$ in figure 10 are no longer present. Because the learning rate goes down as function of epoch, the bad batches changes the weights less, so the abrupt increases in test MSE are gone, while the MSE is still low. The lower MSE achieved is 0.0433, for $\eta_0 = 0.3425$. This is lower than with the constant learning rate.

Next we want to see the effects of different activation functions. We plot the test MSE over 250 epochs using the RELU and Leaky RELU activation functions for the hidden layers, shown in the upper and lower panel of figure 13 respectively. The dynamic η is not used. An upper value of 0.07 have been used for the MSE once again. Because the functions are linear for values above 0, high activations can occur, leading to overflow. This happened for $\eta > 0.15$, and thus not shown. The difference between the two methods are not significant, as was expected, though Leaky RELU gave slightly better results. While RELU yeilded a lowest MSE of 0.0427, Leaky RELU got 0.0425. In Figure 14 we show the results from sigmoid, using the same η values. It is apparent that RELU and Leaky RELU learn much more quickly then sigmoid. While the latter needs at least 25 epochs to get a good result, the former two need 1. However, for RELU and Leaky RELU, the MSE increases again for towards the end of training for high η-values, a sign of overfitting. This indicates it is better to train those two for fewer epochs, and lower η.

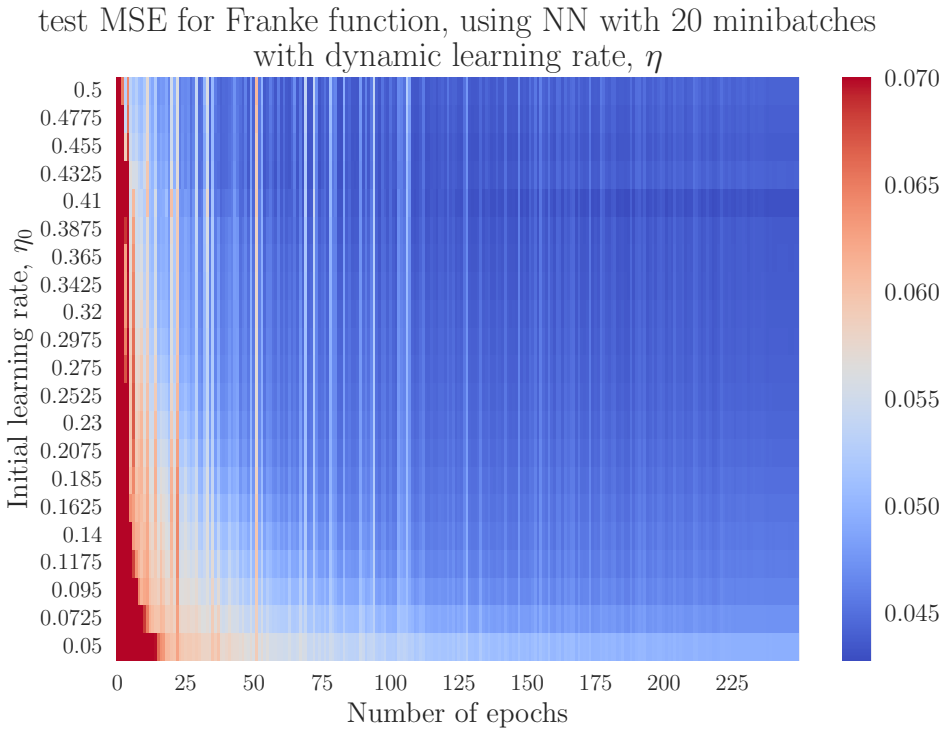

Figure 12. MSE of the Franke function from the Neural network, using a linearly decreasing learning rate, where the initial learning rates are shown on the y-axis.

Since a very integral part of neural networks are the neurons, we also study the way the network is structued. Until now, we have used a network with 2 hidden layers with 10 nodes in each. Without changing the total number of nodes, we train 4 networks for 100 epochs, using the activation function RELU. First, 1 (hidden) layer with 20 nodes is used. Then 2 layers with 10 each, for comparison. Then 3 layers, the middle with 10 nodes and the others with 5, and lastly 4 layers, with 5 nodes each. The same η range as before is used. The four resulting plots are shown in Figure 15. The network with 4 layers is shown in the bottom right. This only learns for a very narrow range of η's, and seems to be too deep, at least for the given value of the other parameters. The network with 3 layers, shown in the bottom left, learns slower but does reach good results. However, along with the network with 1 layer in the upper left, there is much variation. It seems the network with 2 layers perform the best, escpecially as it gives good results for a wide range of η, compared to the other networks. The 3-layered network hits the lowest MSE out of the 4, getting 0.0437.

Lastly, we study the regularization parameter λ. Using the same η values as in Figure 10, and sigmoid as the activation function, we plot the test MSE after 100 epochs for 15 logarithmically distributed values of λ between 10^{-7} and 1. This is shown in Figure 16. We see that we get better result for lower λ, strictly. In [4] we saw that the λ parameters was useful if the noise in the data was higher. We made a similar figure, with $\epsilon = 0.5$. The resulting MSE was (naturally) higher, but the trend was the same as seen here. We see no benefit using other values than $\lambda = 0$. Other parameters or datasets could change this.

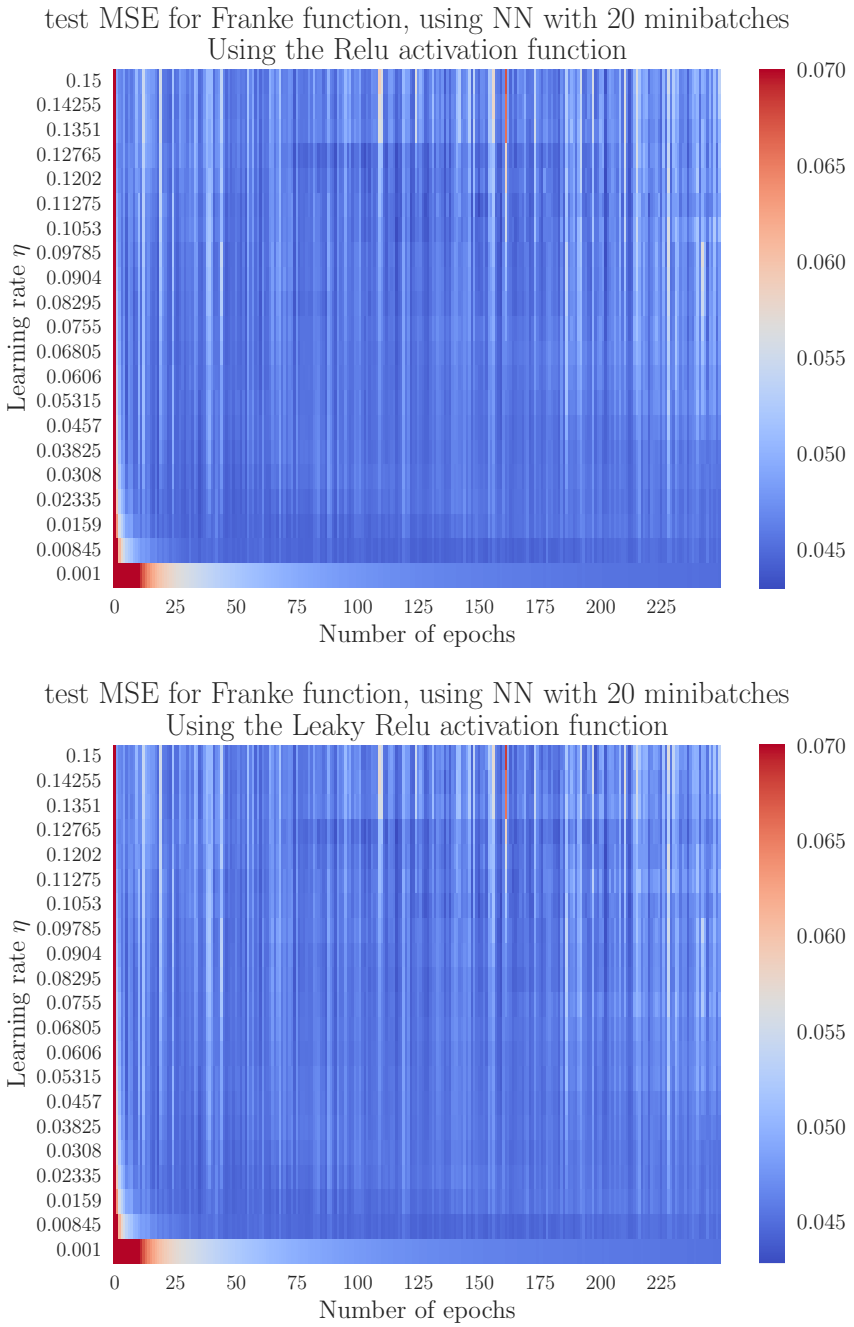

Figure 13. MSE using Relu (top) and Leaky Relu (bottom) as activation function for the hidden layers

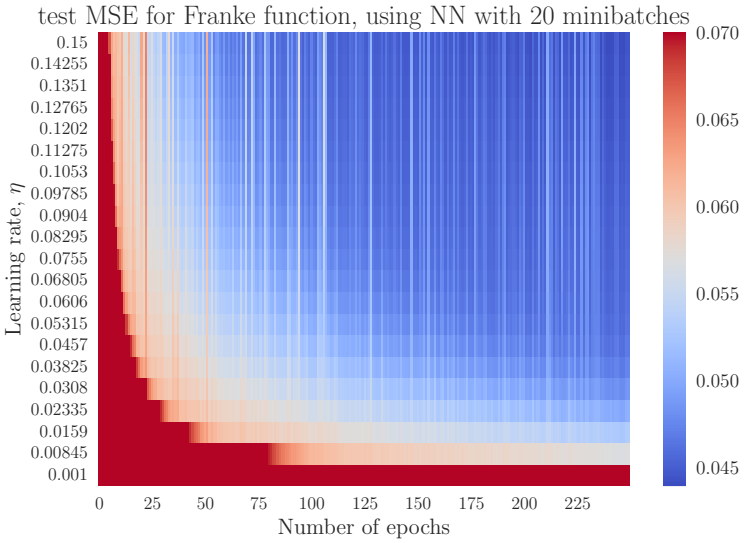

Figure 14. MSE of the Neural network with the sigmoid activation function for the hidden layers. The learning rates and epochs are identical to that of figure 13, allowing us to compare the effect of the different activation functions.

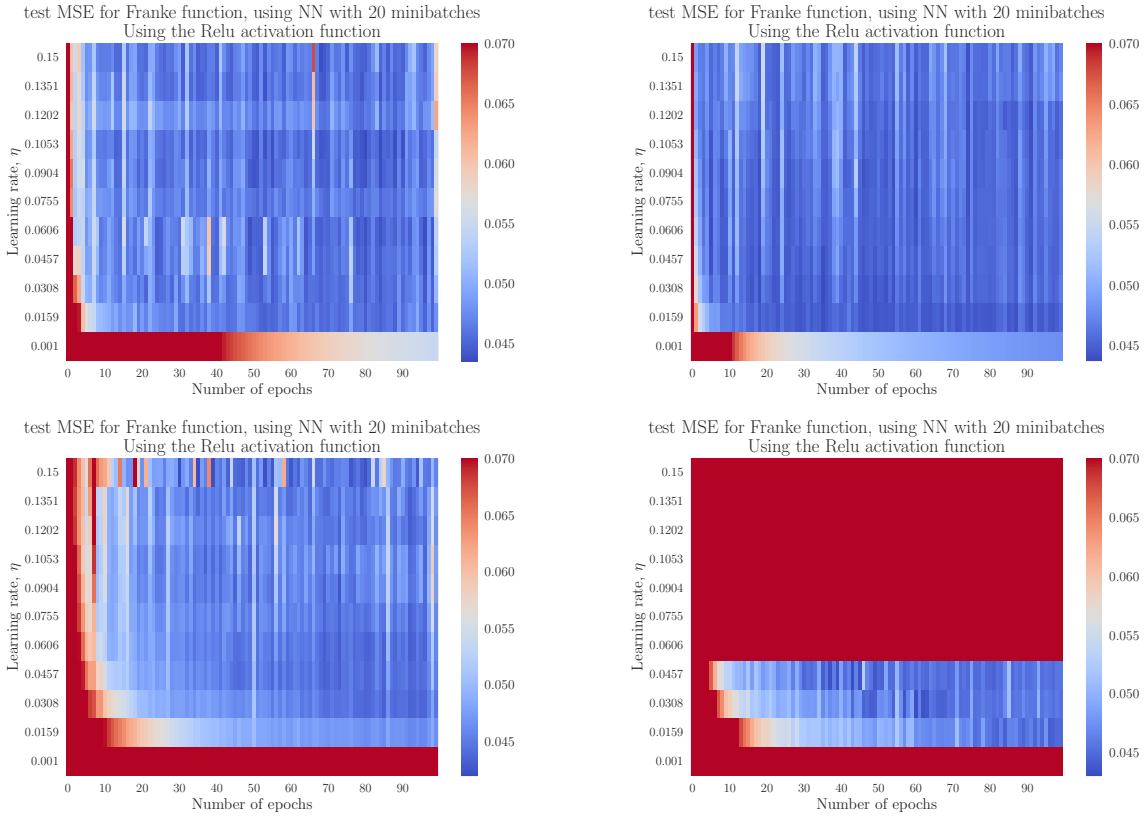

Figure 15. Neural networks with different number of layers, but same number of total nodes. Top left has 1 hidden layer, top right has 2. Bottom left has 3, while bottom right has 4. All have a total of 20 nodes.

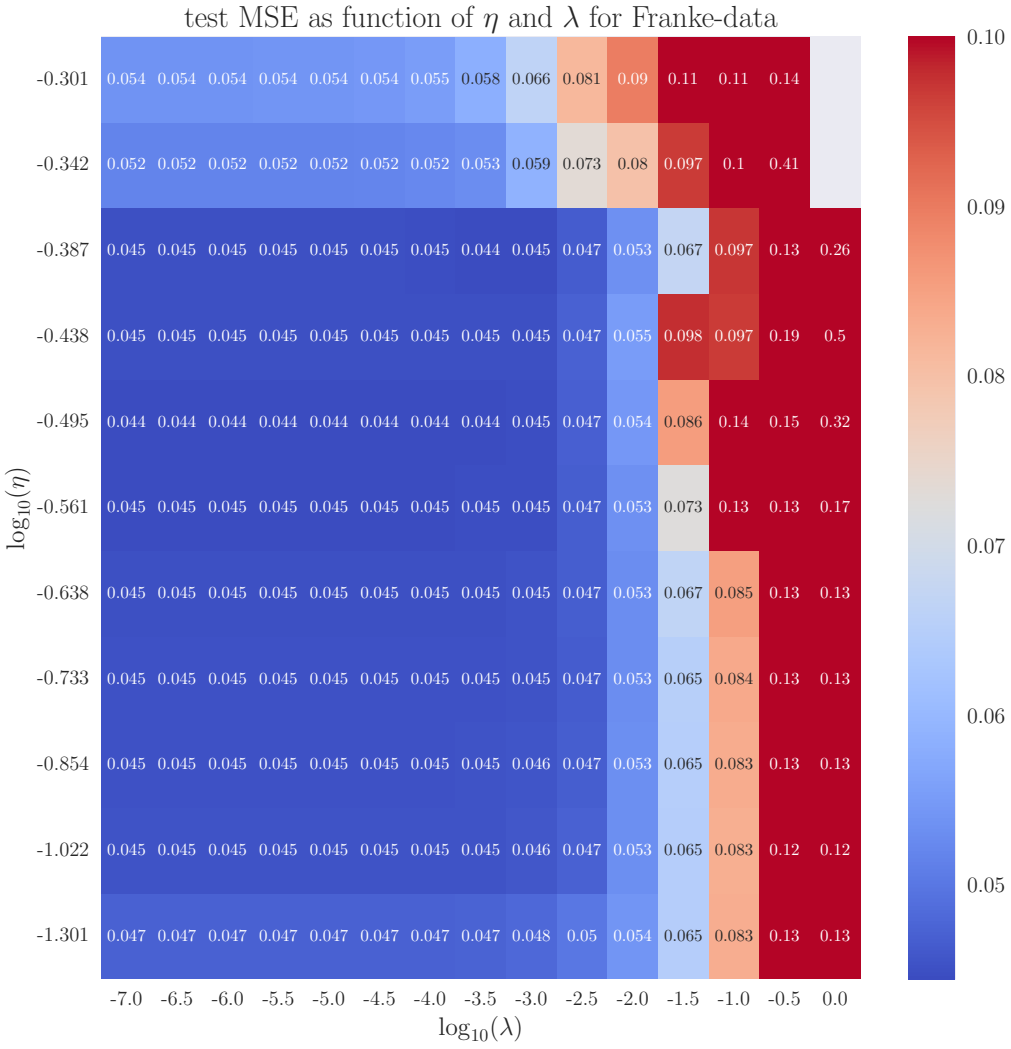

Figure 16. Test MSE as function of learning rate η and regularization parameter λ. The two blank squares in the top right corner correspond to MSE higher than 1, and thus not shown.

4.2 Wisconsin Breast Cancer Data

4.2.1 Feed Forward Neural Network

Again we first want to make sure our neural network actually learns, and study how fast it does this. Using a network with 2 hidden layers with 10 nodes in each, and activating them with sigmoid, we plot the accuracy as function of epoch over 100 epochs for 9 uniformly distributed values of η on a logarithmic scale. This is shown in Figure 17. At the end of training, an accuracy of 98.2% is acheved for several of the η-values. This corresponds to 2 incorrect assesments. This level is first reached and maintained for $\eta = 10^{-2.5}$. Before good results are reached, the accuracy is 63% for a long while. This corresponds to our network predicting that all tumors are benign, because the weights are not properly trained yet. For $\eta = 0.1$, a high level of accuracy, $> 90\%$, is reached already after 1 single epoch. This does not mean the network is confident in its predictions, but it certainly comes with good guesses.

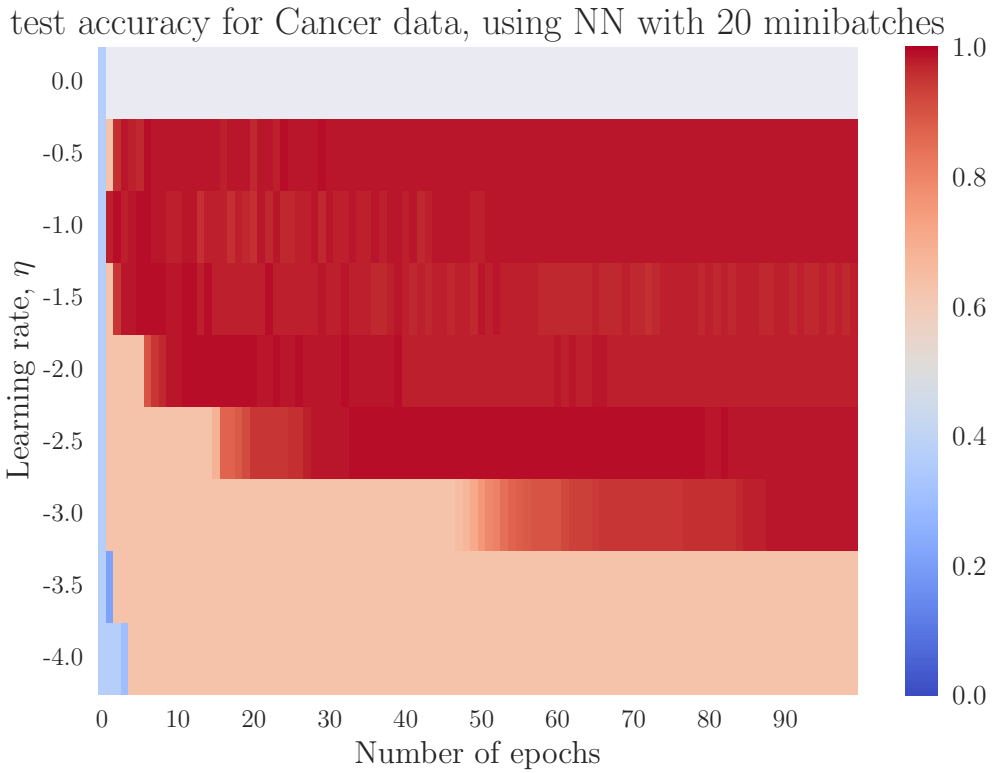

Figure 17. Test accuracy using sigmoid as the activation function, with 2 hidden layers with 10 nodes each. The y-axis shows to base 10 logarithm of the η values. For $\eta = 1$, we get overflow, so no value is shown.

In Figure 18, we make the same plot, now using the learning schedule in equation (3.3). This looks very similar to Figure 17, though one obvious difference is that for $\eta = 10^{-3}$ using 100 epochs is not enough for it to converge. Using the static learning rate, this learning rate only gets good results after about 50 epochs, so this could be expected. The best result is 99.1%, corresponding to 1 mistake, again for $\eta = 10^{-2.5}$. This is higher than when using the static learning rate.

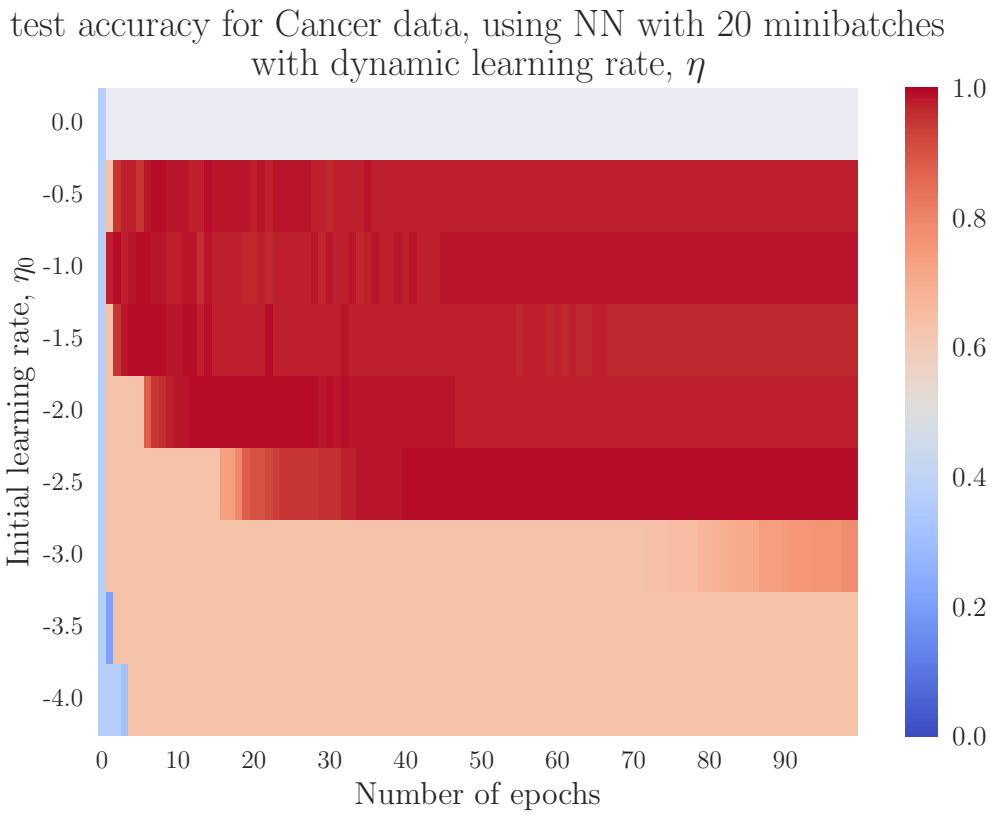

Figure 18. Test accuracy using sigmoid as the activation function, with dynamic learning rate. For $\eta = 1$, we get overflow, so no value is shown.

We now study how different activation functions affect the results. We keep using the dynamic learning rate. For these activation functions, we have to use lower learning rate, choosing 11 values between 10^{-6} and 10^{-1}, as higher learning rates lead to overflow in the gradients. This is shown in Figure 19, with RELU being the top plot, and Leaky RELU the bottom. These activation functions are able to converge to good results within 100 epochs for lower learning rates than sigmoid. This is what we saw in the regression case. However, at the end of training, the best result achieved for both functions were 97.3%, 3 mistakes. There are more η-values for which the accuracy is very high after the first epoch.

To study the L_2-regularization parameter λ, we use the activation function Leaky RELU, for the same values of η as before. Again 100 epochs are used, and the resulting accuracies are only calculated at the end of training. We use 17 logarithmically spaced values of λ, between 10^{-7} and 10^{1}. This is shown in Figure 20. For small values of η the λ parameter does not affect the results. However, for $\lambda = 0$ and $\lambda = 10^{0.5}$, the network achieves better accuracy than before, making just 1 mistake, for several values of η. Further, it can be noted that $\lambda = 10$ gives the network a value for $\eta = 0.1$, without causing overflow.

Lastly, as for the Franke function, we test how different numbers of hidden layers affect the result. Figure 21 shows the results for 4 different structures. All the hidden layers in all the nets have 20 nodes. In the top left figure, 1 layer is used. In the top right, 2 layers. Bottom left has 3 layers, and bottom right 4. At the end of training, all networks had a best accuracy of 98.2%, except for the one with 3 layers, achieving 99.1%. The accuracy

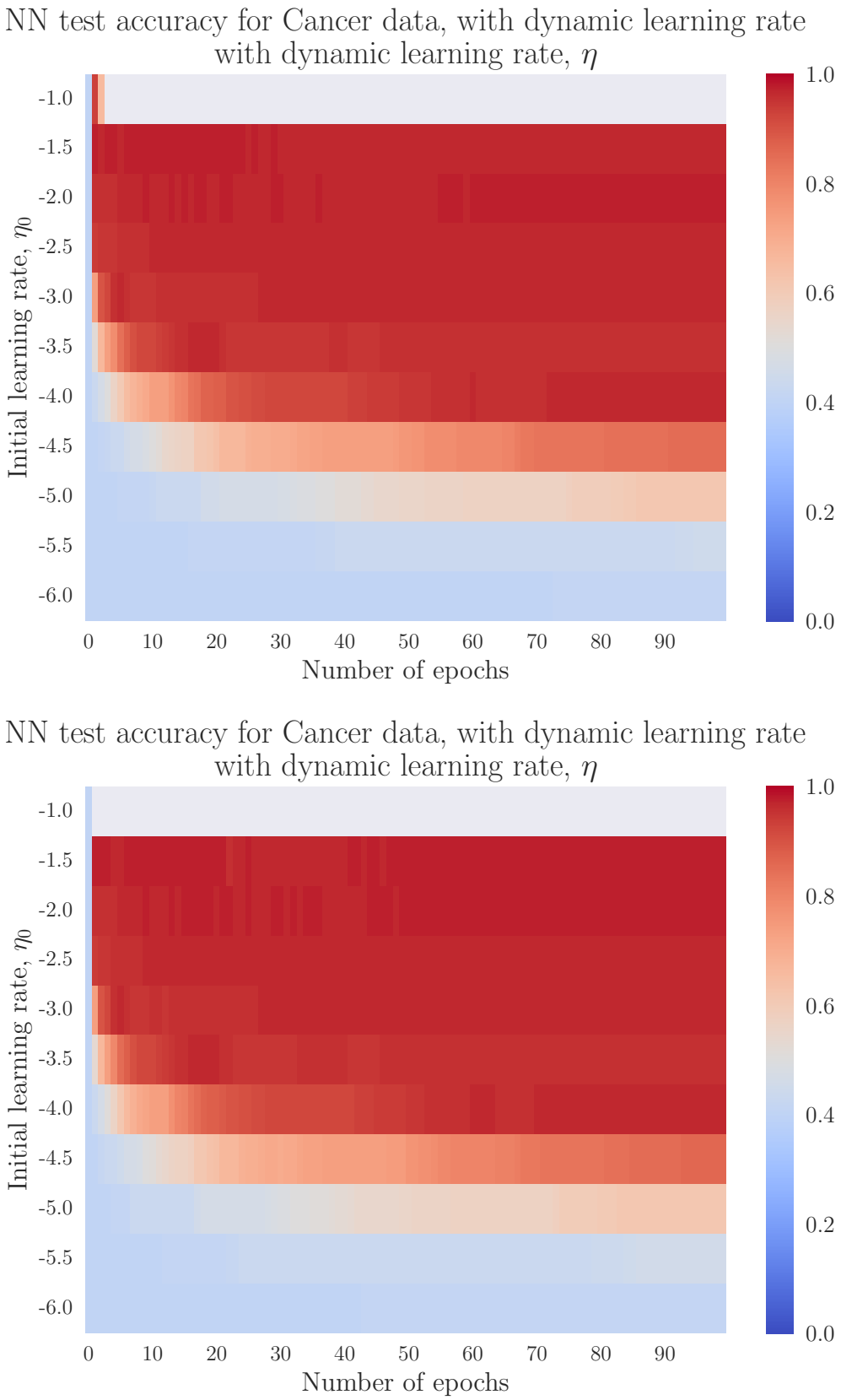

Figure 19. Test accuracy using RELU (top) and Leaky RELU (bottom) as the activation functions. For $\eta = 0.1$, we get overflow, so no value is shown.

before the layers learn are different for each. This is because they have different initial weights, and not nessisarily because one is initially better in the sense that is has learned

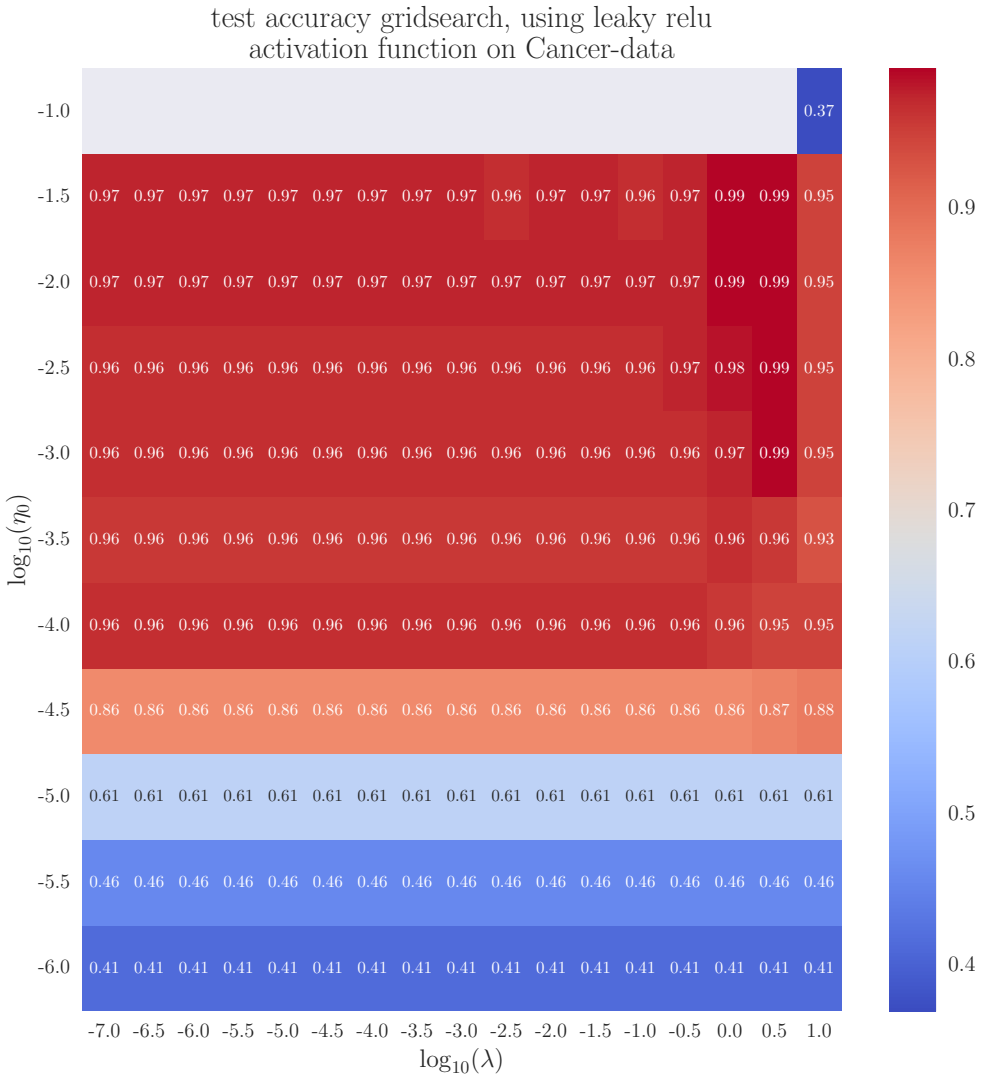

Figure 20. Test accuracy using sigmoid as the activation function, with 2 hidden layers with 10 nodes each. For $\eta = 1$, we get overflow, so no value is shown.

more. It can be noted that for $\eta = 10^{-5}$, the net with 2 layers, in the top right, learns significantly slower than the others.

4.2.2 Logistic Regression

In figure 22 we plot the accuracy score as a function of epochs, for different learning rates η and L_2 parameter set to $\lambda = 0$. The top plot is for the train data, and bottom for test data. As mentioned we want to see how the network learns, meaning that we are not that interested in the precise accuracy score before we perform a grid search. First, looking at the train accuracy, when $\eta \leq 3 \cdot 10^{-5}$ we do not converge within the 200 epochs. We also notice unstable accuracy for $\eta \geq 0.1122$. These effects is more apparent in the test accuracy, where we see clear variations in accuracy at $\eta \geq 0.03981$, and non converging accuracy at $\eta \leq 8 \cdot 10^{-5}$. The optimal learning rate should therefore be within $8 \cdot 10^{-5} \geq \eta \leq 0.1122$.

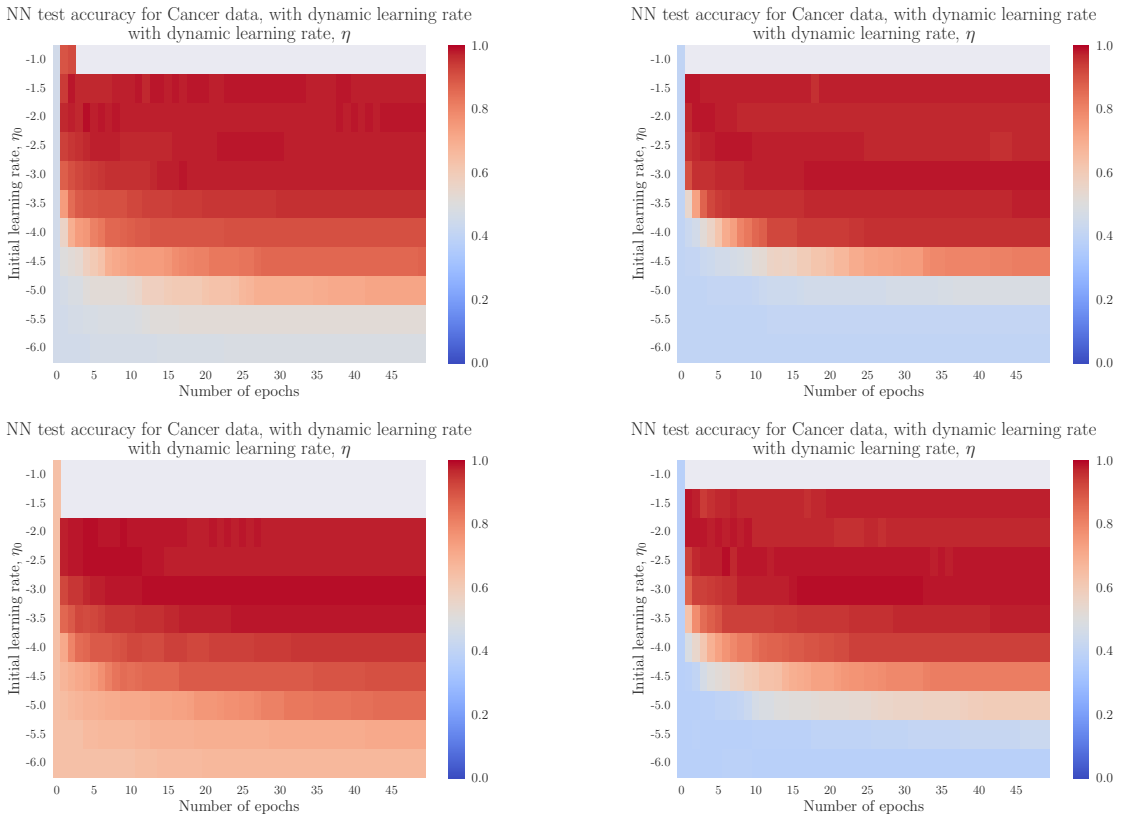

Figure 21. Neural networks with different number of layers. Top left has 1 hidden layer with 20 nodes, top right has 2, each with 20 nodes. Bottom left has 3 hidden layers, each with 20 nodes, while bottom right has 4, each with 20 nodes.

In figure 23 we have plotted the results from our grid search. There we have displayed the precise test accuracy score, as a function of learning rate η and hyper parameter λ. Looking at the figure, we notice that our best accuracy score is 99.1%, for $0.001 \geq \eta \leq 0.00631$ and all λ's. We acquire the same accuracy score for $\eta = 0.01585$ and $\lambda \geq 10^{-3.6}$. This is the exact same accuracy as when using Scikit-Learn's logistic regression with L_2 regularization, and corresponds to one miss-categorization.

There are many potential optimal parameters, however lets choose $\eta = 0.00251$ and $\lambda = 0$, and print the number of cases where our prediction lies withing $(0.1, 0.9)$, which we deem as being inconclusive. We remind that this is not a very comprehensive analysis, and we talk more about this in the discussion-section. Nevertheless, after the first epoch we have 58 cases, and after 200 epochs we have eight cases where the output is within (0.1, 0.9).

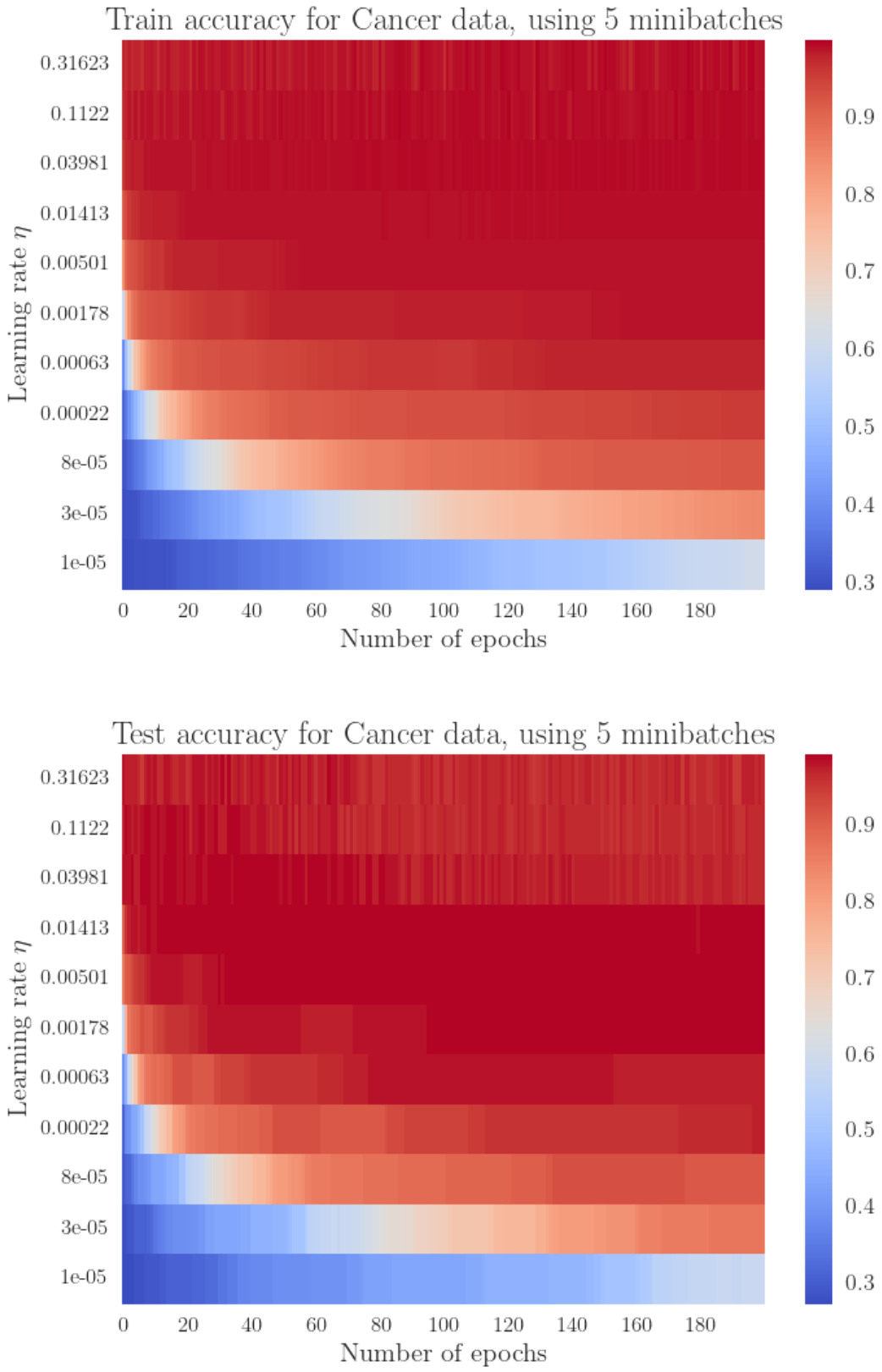

Figure 22. These two figures illustrates how the accuracy score evolves as a function of epochs (x-axis) and learning rate η (y-axis), using logistic regression. An accuracy score of one corresponds to 100% correct classifications.

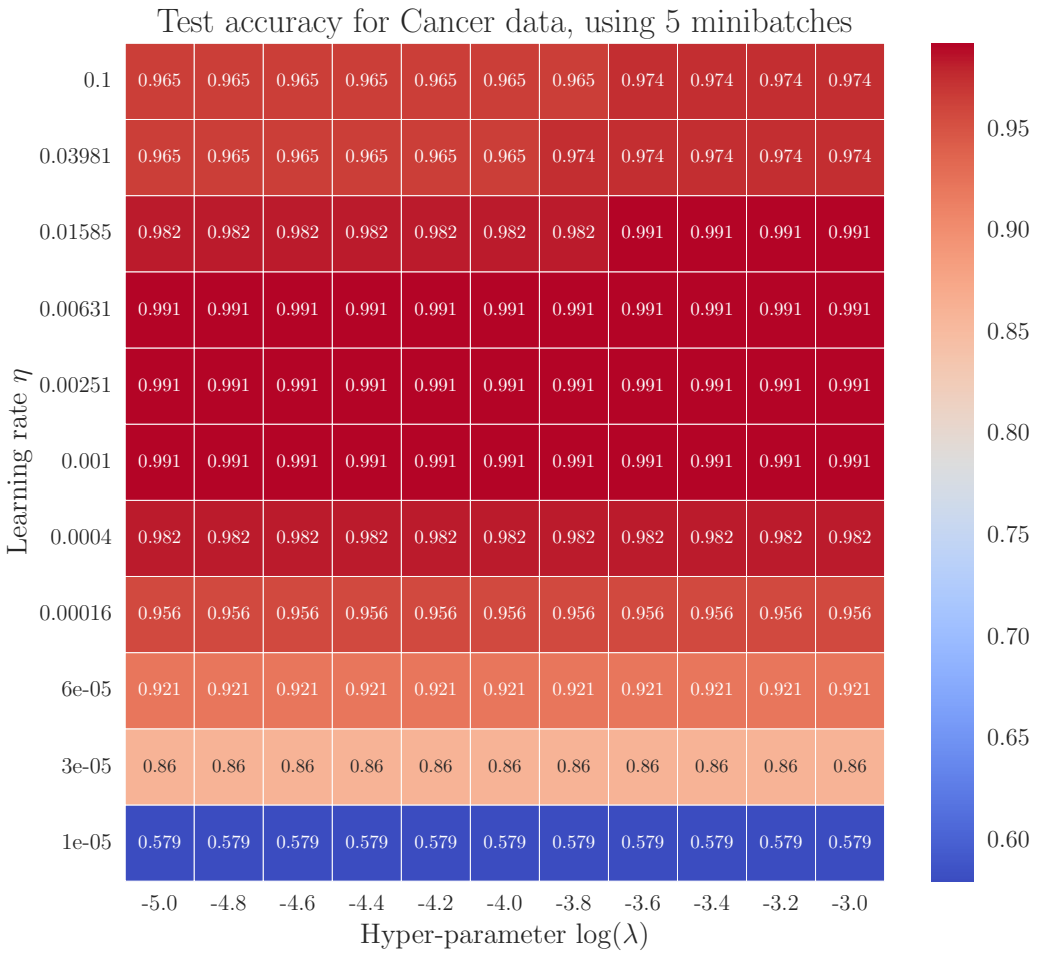

Figure 23. Here we have plotted the accuracy for the test data, as a function of both learning rate η (y-axis) and logarithm of hyper parameter λ (x-axis), using logistic regression. An accuracy score of one corresponds to 100% correct classification.

5 Discussion

We remind the reader that when studying the Franke function using linear regression methods, the lowest MSE we achieved was $MSE = 0.0471$, using bootstrapping as resampling method, and a L_2-regularization parameter $\lambda = 10^{-2.07}$ [4]. In our analysis using SGD, we now found the same lowest MSE. This was achieved with both the implementation of a momentum parameter γ and dynamic learning rate $\eta(t)$. It makes sense that they should be in the same region, given that both methods use the same design matrix to fit a model from the same data. However it surprises us that we get the exact same MSE. With this in mind however, the fact that we obtain the same results with different methods, gives us confidence that our application is correct. The numerical precisions themselves are affected by numerous circumstances, one of which might be randomness in the noise of the data, which differs between the two methods. Using the MSE to conclude that one of the methods is superior to the other is simply something we do not have sufficient information to do. With SGD we omitted the study of the R^2 scores, which would have been reasonable to implement for comparing our result to linear regression.

The neural network was able to score better than linear regression and SGD, getting a lowest MSE of 0.0425. A lower MSE than 0.0471 was generally seen for most configurations of the network, so this seems like the superiour method for minimizing the error. While both linear regression and SGD fits a bivariate polynomial of degree p, having P terms, the neural network has no constraint in the model it makes. This is one of the reasons machine learning is often called a black-box, as we can not study how the network represents the Franke function. If the only goal is getting a good result, this is not a problem. However, we can gain no deeper understanding of the problem at hand.

Our main justification for omitting a design matrix with polynomial data points for the neural network training was to obtain a model that could reproduce data different from the Franke function. This would have been a sensible thing to do. However, throughout the report we have seen how sensitive neural networks can be to initial parameters, and fitting e.g. terrain data would require a fair amount of analysis once again in order to tune parameters appropriately. For this reason we have chosen not to do it, but throughout the report we have developed a fairly decent insight of how the different parameters affects the network, and how to adjust parameters depending on a result we have, compared to one we wish to obtain. Achieving this insight is among the most important results of this report.

We know that the optimal choice of a given parameter are affected by changes to the other parameters. This implies that our method is not guaranteed to optimize the entire parameter space. However, we have assumed that it gives a good approximation, supported by the fact that our results are consistent with those obtained from linear regression.

It would have been nice to plot η vs γ, to see how the different combinations work. This would also provide a deeper insight of the general impact of γ. Lower η values yield more stable evolutions of the MSE, making it a favorable choice if we increase the number of epochs. It would have been interesting to see whether momentum could have been

implemented on these values in a way to speed up convergence and simultaneously achieve stable MSE convergence. Nonetheless, our analysis showed us how it affects the result in a way we expected.

When we fitted the Franke function with a neural network we did not study the effects of the momentum parameter. Multiple learning rates caused unstable MSE convergence for the network. Lower learning rates reduced these fluctuations, but at the expense of slowly converging models. Studying momentum would therefore be an analysis of great interest, checking whether one can obtain a model with quickly converging MSE while also avoiding the fluctuations of the MSE during the process. This is a netural point of furhter study, and from our results for the parameter using SGD, we hypotesise we can achieve faster, and possibly better results.

Regarding the dynamic learning rate, we could have chosen many different methods of scaling it. For instance exponential decreasing η, an upside-down sigmoid or a linear function ending on a non-zero value, e.g. a small percentage of the initial value of η_0. The first two forementioned methods could have been adjusted to get high learning rates when the errors are large, declining as it drops and stabilizing at low values when the errors become small. However, this would simply give us even more parameters to tune, and we obtained interestnig results with our simple model, clearly displaying the impact of scaling, and its advantages.

When we studied to minimum MSE values obtained when fitting the Franke function, it's important to emphasize that these values do not necessarily correspond to the optimal parameter choices for our model. Certain parameters cause abrupt variations of the MSE for multiple epochs, and when the lowest MSE values are found for these parameters it is a result most likely due to fortunate "mishaps". Such parameter choices are not optimal, despite giving the lowest MSE values, as the model is very sensitive to our specific setup, as well as halting the calculations at very specific epochs.

The fact that we obtain the same accuracy (99.2%) as when using Scikit-Learn's logistic regression, gives us confidence in our results. The 99.2% accuracy corresponds to one missclassification. We do not believe it feasible to better this result, given that a professionally written library yields the same. One area which we would have liked to analyze further is the confidence of our algorithm. After 200 epochs, for $\eta = 0.00251$ and $\lambda = 0$, we had eight cases where the output was within (0.1, 0.9). That threshold is arbitrary, however it illustrates potential further analysis. To find the optimal parameters we should not only analyze accuracy, but also minimize the number of cases which one deems to inconclusive. Methods of optimization one could try, is implementing different output functions, initialization of weights and bias and momentum parameter (2.6).

6 Conclusion

With SGD we managed to get the exact same $MSE = 0.071$ as our previous project [4]. This was done with learning rate $\eta = 0.25$ and with implementation of momentum

parameter γ and dynamic learning rate η separately.

References

[1] Xavier Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track*, 9:249–256, 01 2010.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *IEEE International Conference on Computer Vision (ICCV 2015)*, 1502, 02 2015.

[3] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre G.R. Day, Clint Richardson, Charles K. Fisher, and David J. Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics Reports*, 810:1–124, May 2019.

[4] Håkon Olav Torvik, Vetle Vikenes, and Sigurd Sørlie Rustad. Analysis of regression and resampling methods. *University of Oslo*, October 2021.