Using a Feedforward Neural Network to Perform Linear and Logistic Regression

Leah Hansen, Ida Monsen, and Vetle Henrik Hvoslef
Department of Physics, University of Oslo
(Dated: November 18, 2023)

In this report we have used feedforward neural networks to perform both regression and classification analysis. The network is implemented with a back propagation algorithm using automatic differentiation in order to be applicable to different cost- and activation functions, allowing us to use the same base code for both regression and classification. We did linear regression on the Franke function and classification on the Wisconsin Breast Cancer data, using test mean squared error (MSE), test R^2 score and test accuracy score to search for the optimal parameters for our neural network. For regression our neural network had a best test MSE of 0.100 and test R^2 score of 0.903, while linear regression yields a test MSE of 0.0273 and test R^2 score of 0.999. For classification our best neural network had a test accuracy score of 0.886, while logistic regression obtained a test accuracy score of 0.930. We also compared the best results obtained with neural networks to those obtained with linear and logistic regression. We found that the neural networks performed worse than their simpler counterparts for both regression and classification. Additionally, a comparison with Scikit-learn shows that our implementations are worse than Scikit-learn's.

I. INTRODUCTION

The successful development and evolution of Neural Networks (NN) in the field of computer science has been a catalyst for many of the luxuries we enjoy today. For example, NNs are used to aid users in finding a movie or TV show they like on popular streaming platforms like Netflix [1], which employs NNs to determine which visuals are likely to attract a user. Google, Facebook, and many others employ Neural Networks to perform and aid their services as well, as mentioned in the introduction of Michael Nielsen's book on Neural Networks [2].

A strength of neural networks is that instead of us having to tell the network what kind of relationship we expect between the input and output, the network is able to learn this mapping, as explained in the introduction to chapter 6 of the book *Deep Learning* [3]. This makes the method flexible, so that it can be used to approximate mappings in widely different data sets.

In this report we will limit ourselves to studying feedforward neural networks. This type of network passes data through a network of function evaluations, going only in one direction from input to output [3]. An optimisation scheme is then employed to adjust this network of function evaluations in such a way that the network best represents the relationship between input and output data. For optimisation, we employ the widely used family of gradient descent methods. These methods uses the gradient of a multidimensional function to search for the function's minimum.

We are going to put the wide applicability of feedforward neural networks to the test by both using it to do regression on the continuous, two-dimensional Franke function, and to classify a binary classification data set. Traditionally, this would have to be done by two different methods, namely linear regression for regression of a continuous function and logistic regression for classification. We will compare the performance of these traditional methods to that of the neural network.

In Section II, we first explain the theory of optimisation, gradient descent and classification problems, before we move on to the theory of feedforward neural networks and theoretical details related to neural networks. The section ends with an explanation of the procedure used and presentation of the data sets. Results and the discussion of them are presented in Section III. The results and discussion section is structured so that we treat gradient descent in Subsection III A, regression in Subsection III B and lastly classification in Subsection III C. We then take a step back and discuss some overarching possible improvements in the Subsection III D. Finally, we conclude our report in Section IV.

II. FORMALISM

A. Optimisation

Most machine learning methods include optimising a function at some point [3]. More precisely, we want to find the global minimum of some cost function, which will then be the point at which our model best fits the data. This can mathematically be phrased as

$$\boldsymbol{\beta}^* = \arg\min_x f(\boldsymbol{\beta}),$$

where $\boldsymbol{\beta}$ are the parameters of our model and $f(\boldsymbol{\beta})$ is the cost function [3]. The choice $\boldsymbol{\beta}^*$ for the parameters will then give us the best model, in the sense that they are the parameters that minimise our chosen cost function.

If f is differentiable, the candidates for $\boldsymbol{\beta}^*$ are the points where the derivative, or gradient if f is a multi-

[1] https://github.com/leahelha/FYS-STK4155/tree/main/Project2

dimensional function, is zero, i.e. points satisfying the equation

$$\boldsymbol{\nabla} f(\boldsymbol{\beta}) = \mathbf{0}, \tag{1}$$

where $\boldsymbol{\nabla}$ is the gradient operator. These points are called critical points and will be either local minima, local maxima, or saddle points [3].

Finding the global minimum is in general difficult. For a convex function, however, it is relatively simple [4], at least when the function is differentiable. The minimisation problem then reduces to finding the single point where the cost function's gradient is zero, and this is guaranteed to be the global minimum. But in general, there may be many such minimum points where the gradient is zero, and not all of them are the global minimum $\boldsymbol{\beta}^*$ that we are searching for. Another difficulty is that there might not exist an analytical solution of Equation (1).

In practice, we often settle for finding a set of parameters $\boldsymbol{\beta}$ that give a low enough cost function, but that are not the global minimum [3]. But we can still utilise the gradient when searching for such a point.

B. Gradient descent

Gradient descent is a family of methods that uses the gradient of a function to search for its minimum. The general idea is to start at a chosen point $\boldsymbol{\beta}^{(0)}$ and then move to lower and lower values of f by taking steps in the direction of the negative gradient [3], resulting in the iteration

$$\boldsymbol{\beta}^{(k+1)} = \boldsymbol{\beta}^{(k)} - \eta \boldsymbol{\nabla} f(\boldsymbol{\beta}^{(\boldsymbol{k})}).$$

Here η is a positive parameter called the learning rate, governing how big steps we take in each iteration. If η is not too large, this iteration will converge to a minimum. Note that this may not be the global minimum that we want to find.

There are several different ways to choose η, and other ways of refining gradient descent, giving rise to different methods. We will here present the methods (gradient descent with momentum?) Stochastic gradient descent, AdaGrad, RMSprop, and ADAM. Stochastic gradient descent can also be combined with the other three methods to give you stochastic AdaGrad, and so on. Note that these methods are presented with different notation conventions in different texts. Here we have tried to use the same notation for all methods in order to make them more easily comparable.

1. Stochastic gradient descent

One drawback of gradient descent is that calculating the gradient is computationally expensive, and grows with the number of data points we have. Additionally, the method might converge to a local minimum instead of the global one. Stochastic gradient descent addresses both of these potential issues, and variations on this method are arguably the popular optimisation algorithms in machine learning [3].

In short, stochastic gradient descent uses only a subset of the data points used for training when it calculates the gradient. This means that the method is potentially much faster, and also allows us to escape local minima due to the randomness introduced [5].

2. AdaGrad

The AdaGrad method uses both the first and the second moment of the gradient, in other words, both the gradient and the gradient squared, when doing gradient descent. AdaGrad keeps a history of all squared gradients and adapts the learning rate as it goes by dividing by the square root of this sum. This allows the method to move faster in flatter directions of parameter space [3]. The method is presented in Algorithm 1. $\boldsymbol{g}_t$ denotes the gradient, $\boldsymbol{v}_t$ is a running average of the second moment of the gradient and η is the learning rate. The regularisation constant ϵ is included to avoid divergence and is typically chosen to be of the order 10^{-8}.

Algorithm 1 Algorithm for AdaGrad

Require: $\eta, f(\boldsymbol{\theta}), \boldsymbol{\theta}_0, \epsilon$
  Initialise $\boldsymbol{v}_0, t$ to zero
  while $\boldsymbol{\theta}_t$ not converged **do**
    $t \leftarrow t + 1$
    $\boldsymbol{g}_t \leftarrow \boldsymbol{\nabla} f(\boldsymbol{\theta}_{t-1})$
    $\boldsymbol{v}_t \leftarrow \boldsymbol{v}_{t-1} + \boldsymbol{g}_t^2$
    $\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta_t \boldsymbol{g}_t / (\sqrt{\boldsymbol{v}_t} + \epsilon)$
  return $\boldsymbol{\theta}_t$

3. RMSProp

RMSProp, or Root Mean Squared Propagation, is a gradient descent method that also uses the second moment of the gradient [3]. Compared to AdaGrad, RMSProp performs better on nonconvex functions [3] by using a weighted average over moments. The method is presented in Algorithm 2. Here $\boldsymbol{g}_t$ is the gradient, $\boldsymbol{v}_t$ is a running weighted average of the second moment of the gradient, β is a parameter controlling this average and η is the learning rate. The regularisation constant ϵ is included to avoid divergence, and is typically chosen to be of the order 10^{-8}.

Algorithm 2 Algorithm for RMSProp

Require: $\eta, \beta, f(\boldsymbol{\theta}), \boldsymbol{\theta}_0, \epsilon$
  Initialise $\boldsymbol{v}_0, t$ to zero
  while $\boldsymbol{\theta}_t$ not converged **do**
    $t \leftarrow t + 1$
    $\boldsymbol{g}_t \leftarrow \boldsymbol{\nabla} f(\boldsymbol{\theta}_{t-1})$
    $\boldsymbol{v}_t \leftarrow \beta \boldsymbol{v}_{t-1} + (1-\beta)\boldsymbol{g}_t^2$
    $\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta_t \boldsymbol{g}_t / (\sqrt{\boldsymbol{v}_t} + \epsilon)$
  return $\boldsymbol{\theta}_t$

4. Adam

The Adam method is a newer method inspired by RMSProp [6]. Adam, for adaptive moment estimation, is presented in Algorithm 3, where all vector operations are done elementwise. It uses moving averages $\boldsymbol{m}_t$ over the gradient and $\boldsymbol{v}_t$ over the gradient squared, and exponential decay rates for these averages governed by $\beta_1, \beta_2 \in [0, 1)$, respectively. $\boldsymbol{m}_t$ and $\boldsymbol{v}_t$ are called the first and second moments of the gradient. We also need to specify the step size α and the regularisation constant ϵ. Good starting values to try are $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$ [6].

Algorithm 3 Algorithm for Adam

Require: $\alpha, \beta_1, \beta_2, f(\boldsymbol{\theta}), \boldsymbol{\theta}_0, \epsilon$
  Initialise $\boldsymbol{m}_0, \boldsymbol{v}_0, t$ to zero
  while $\boldsymbol{\theta}_t$ not converged **do**
    $t \leftarrow t + 1$
    $\boldsymbol{g}_t \leftarrow \boldsymbol{\nabla} f(\boldsymbol{\theta}_{t-1})$
    $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t$
    $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2$
    $\hat{\boldsymbol{m}}_t = \boldsymbol{m}_t / (1-\beta_1^t)$
    $\hat{\boldsymbol{v}}_t = \boldsymbol{v}_t / (1-\beta_2^t)$
    $\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon)$
  return $\boldsymbol{\theta}_t$

C. Classification

For this section, our main reference source will be chapter 5 of the book *Deep Learning* [3], unless otherwise stated. Classification is a type of machine learning problem where you want to classify your data into one of k groups, instead of approximating a continuous function. The output can either be a discrete value denoting a specific group, or a probability distribution over the groups. The case with discrete values is called hard classification and the case with probabilities is called soft classification. When using hard classification, information about the uncertainty of a prediction is lost. We can use neural networks to solve classification problems, but the classical way of doing it is with logistic regression.

1. Logistic regression

The logistic regression method uses the Sigmoid function, also called the logistic function, to turn a non-bounded function into a function with output values in the interval (0,1). The output from the logistic function can therefore be interpreted as a probability. The logistic function is given by

$$p(t) = \frac{1}{1 + e^{-t}} \tag{2}$$

where t can be the output of another continuous function, for example $t = f(x) = \beta_1 x + \beta_0$. This is the typical function choice when we are doing binary classification. If we are doing hard classification, a value $p(t) < 0.5$ will be interpreted as category 0, and a value $p(t) \geq 0.5$ will be interpreted as category 1.

To turn this into a regression problem, we need to define a cost function that we can optimise with respect to β_0 and β_1. We choose to search for the parameters that maximise the probability $P(\mathcal{D} \mid \boldsymbol{\beta})$ of getting the data we have, where $\mathcal{D} = \{(x_i, y_i)\}$ are all our data points. The x_i's are continuous values, while the y_i's are either 0 or 1 in the case of binary classification. This scheme is called the maximum likelihood estimator, and the probability can be expressed as

$$P(\mathcal{D} \mid \boldsymbol{\beta}) = \prod_{i=1}^{n} \left[p\left(y_i = 1 \mid x_i, \boldsymbol{\beta}\right)\right]^{y_i} \left[1 - p\left(y_i = 1 \mid x_i, \boldsymbol{\beta}\right)\right]^{1-y_i}. \tag{3}$$

where $\beta = (\beta_0, \beta_1)$ are the parameters of the function.

Equivalently to maximising the above expression, we can instead minimise the negative logarithm of the probability. This is done for ease of computation. We then arrive at our cost function for logistic regression in the binary case, namely

$$\begin{aligned} \mathcal{C}(\boldsymbol{\beta}) = \sum_{i=1}^{n} (y_i \log p\left(y_i = 1 \mid x_i, \boldsymbol{\beta}\right) \\ + (1 - y_i) \log\left[1 - p\left(y_i = 1 \mid x_i, \boldsymbol{\beta}\right)\right]) \end{aligned} \tag{4}$$

There are in some cases analytical solutions to this minimisation problem, but in general we can use a numerical optimisation scheme as explained in Section II B.

As a measure of performance for a logistic regression model, we can use the accuracy score defined as

$$\text{accuracy} = \frac{1}{n} \sum_{i=1}^{n} I(\hat{y}_i = y_i), \tag{5}$$

where n is the number of data points, $\hat{y}_i$ is the predicted group of sample i and $I(\hat{y}_i = y_i)$ is equal to 1 if $\hat{y}_i = y_i$ and 0 otherwise. This is the percentage of samples where the predicted group is the correct group.

D. Feedforward neural networks

Neural networks (NNs) are a subset of machine learning methods inspired by the structure and functioning of biological neurons in the human brain. A NN is comprised of neurons, an input layer, an output layer, and potentially several hidden layers. Hidden layers are named so because they are neither input nor output layers. A neuron is the fundamental unit of NNs, and takes one or more input values. It processes the input using weights and an activation function and produces an output. The neurons in the input layer must represent the features of the input data. In other words, the input layer has one neuron for each feature of the raw data. An illustration of a neural network is given in Fig.1. For more on activation functions see Section II D 3.

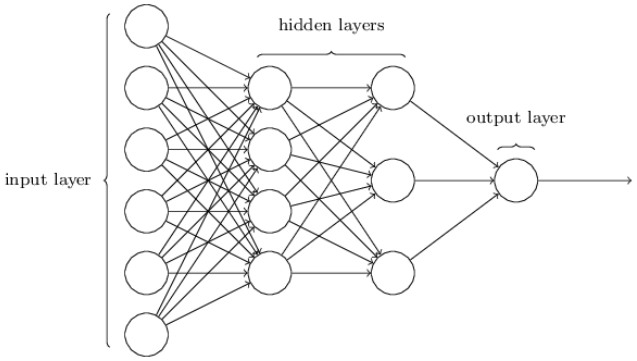

FIG. 1: Figure illustrating a NN with an input layer consisting of six neurons, two hidden layers, and one output layer. From Nielsen's book, Chapter 1 Section 3. [2]

For this section, our main reference source will be Nielsen's, *Neural Networks and Deep Learning* [2], unless otherwise stated. Feedforward neural networks (FFNNs) are neural networks where the information is always fed forward to the next layer. This distinction is made as some NNs contain loops in the network. In our study, we will solely focus on FFNNs.

To train a feedforward neural network, we consecutively do one feedforward pass and then one back-propagation pass, again and again until we are satisfied with the result or do not see an improvement in performance. This training process is summarised in Algorithm 4, and the feedforward and back-propagation steps themselves are explained in the following subsections.

Algorithm 4 Algorithm for training

Require: Vectors of all the biases for all the layers, $\hat{b}$
Require: Matrices of all the weights between the layers, $\hat{W}$
Calculate all $\hat{z}^\ell$ and $\hat{a}^\ell$ through the Forward-propagation
Then back-propagate the errors through the layers, δ^L and δ^ℓ
Then use this to update the weights and biases update rule

1. Forward-propagation

As previously mentioned, the data first gets passed to the input layer of the FFNN. The input layer then feeds the data into the first hidden layer. All layers after the first one has both weights and biases in addition to that layer's activation function. The algorithm for doing forward-propagation is presented in Algorithm 5, and will be explained in detail in the following paragraphs.

Algorithm 5 Algorithm for feedforward

Require: Vectors of all the biases for each layer
Require: Matrices of all the weights between two layers
for $\ell = 2, ..., L$ **do**
$\hat{z}^\ell = \hat{W}^\ell \hat{a}^{\ell-1} + \hat{b}^\ell$
$\hat{a}^\ell = f^\ell(\hat{z}^\ell)$

$\hat{W}^\ell$ is a $N_{\ell-1} \times N_\ell$ matrix containing all the weights between layer ℓ and $\ell - 1$, with N_ℓ being the number of nodes in a certain layer ℓ. Element w^ℓ_{ij} thus describes the weight between neuron i in layer ℓ and neuron j in layer $\ell - 1$, which can be seen as a description of how tightly these two neurons are connected. We observe that each neuron i in layer ℓ gets a weighted sum of the output from the entire previous layer $\ell - 1$.

The activation function then takes the weighted sum of the output from the previous layer as its function argument, and produces a new output for each neuron in the next layer. To be precise we use a super-script ℓ in f^ℓ to highlight that the activation function can be different for each layer, but it is common to choose the same function for each hidden layer.

The vector $\hat{a}^\ell$ is the vector containing all the outputs from the neurons in layer ℓ, with the element a^ℓ_i being the output from the activation function of neuron i in layer ℓ. Similarly, $\hat{b}^\ell$ describes all the biases in layer ℓ, and b^ℓ_i is the bias of neuron i in layer ℓ.

L denotes the total number of layers, such that layer L is the output layer. The activation function of that layer is chosen specifically to be well-suited to the type of problem we are trying to solve. This function can therefore be different to the activation function in the hidden layers. For example is the activation function for a regression problem is just the value the output layer gets from the hidden layers.

2. Back-propagation

Back-propagation is the way our FFNN can update its weights and biases, by searching for the weights and biases that minimise the cost function we have chosen. This is how the FFNN can then learn to replicate the data. All the outputs $\hat{z}^\ell$ are already calculated in the feedforward pass. Here the cost function of our FFNN is a function of all the weights $\hat{W}$ and biases $\hat{b}$, $\mathcal{C}(\hat{W}, \hat{b})$, and we can thus use an optimisation scheme to search for the

weights and biases that will minimise the cost function. The Algorithm 6 summarises the back-propagation step, which is further explained below.

Algorithm 6 Algorithm for back-propagation

Require: All the biases for all the layers, $\hat{b}^\ell$
Require: Matrices of all the weights for all the layers, $\hat{W}^\ell$
Require: $\hat{z}^\ell$ for all the layers
Require: Regularisation parameter λ
$\delta^L = \boldsymbol{\nabla}_a \mathcal{C} \odot f'(z^L)$
$\boldsymbol{\nabla}\hat{W}^L = \hat{a}^L \delta^L + \lambda \|\hat{W}^L\|$
$\boldsymbol{\nabla}\hat{b}^L = \sum \delta_i^L + \lambda \|\hat{b}^L\|$
Update weights and biases for the output layer using the gradients $\boldsymbol{\nabla}\hat{W}^L$ and $\boldsymbol{\nabla}\hat{b}^L$
for $\ell = L-1, ..., 2$ **do**
$\delta^\ell = ((\hat{W}^{\ell+1})^T \delta^{\ell+1}) \odot \sigma'(z^\ell)$
$\boldsymbol{\nabla}\hat{W}^\ell = \hat{a}^\ell \delta^\ell + \lambda \|\hat{W}^\ell\|$
$\boldsymbol{\nabla}\hat{b}^\ell = \sum \delta_i^\ell + \lambda \|\hat{b}^\ell\|$
Update weights and biases using the gradients $\boldsymbol{\nabla}\hat{W}^\ell$ and $\boldsymbol{\nabla}\hat{b}^\ell$

The update rule step in the Algoritm 6 is specified in the method section and based on the algorithms in gradient descent section. $\boldsymbol{\nabla}_a \mathcal{C}$ denotes the gradient of the cost function with respect to the output α^L and $\odot$ denotes elementwise multiplication. For a derivation of these formulas see the book [2] by Nielsen. We can see here why the algorithm is called back-propagation, since it calculates the gradient for each layer backwards from the output layer.

3. *Activation functions*

The activation function is a crucial part of any NN. Ideally the activation functions should be computationally fast, as we are going to evaluate them for every neuron each time we do one feedforward pass. We also want the activation functions to have well-behaved gradients, since exploding or vanishing gradients can make the back-propagation less effective [7]. If the gradients increase uncontrollably, the update step would change to weights or biases too much, and if the gradients become too close to zero the network effectively stops training.

The activation functions should lastly satisfy the requirements of the universal approximation theorem [8], namely be continuous, bounded and monotonically increasing. Below we present the three activation functions used in this report, all of which satisfies the universal approximation theorem.

The Sigmoid function is given by

$$f(x) = \frac{1}{1+e^{-x}}, \tag{6}$$

and is the same function as Equation (2) used for logistic regression. This function is a popular choice, but as discussed in the paper [7], vanishing gradients can be a problem. Starting from a normal distribution of weights and biases, the Sigmoid function would in many cases output just 0 or 1, in which case the function is almost flat and the gradient is close to zero.

The rectified linear unit (RELU) function is given by

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{7}$$

This function fixes the problem of vanishing gradients that the Sigmoid function suffers from, and is also computationally fast. However, some neurons can start outputting only 0 and thus effectively die, since the gradient will then always be 0 as well.

The Leaky RELU (LRELU) function is a modification of the RELU function that fixes the problem of dying neurons. It is given by

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \delta x & \text{otherwise.} \end{cases} \tag{8}$$

where δ can be any positive, typically small, number. We have chosen to use $\delta = 10^{-4}$.

E. Automatic differentiation

Training a neural network involves calculating many derivatives, and a reliable and fast method of differentiating is therefore called for. We will be using automatic differentiation, a method that avoids symbolic differentiation and can calculate derivatives to machine precision [9].

Automatic differentiation utilises the chain rule,

$$\frac{d}{dx} f(g(x)) = \frac{d}{dg} f(g) \cdot \frac{d}{dx} g(x),$$

multiple times. Given a function, we can decompose it into a set of smaller operations with well-known derivatives. Automatic differentiation then starts at the innermost operation, calculates its derivative and builds up to the derivative of the whole function using the chain rule.

In our implementation, we use the Python package *Autograd* [9] for calculating derivatives. This way we avoid having to explicitly calculate and write the derivatives for different functions, allowing us to use exactly the same code for different activation functions.

F. Procedure

We will throughout the report compare our neural network performance with those implemented in Scikit-learn [10], as well as with standard linear and logistic regression. The linear regression results we will compare to are taken from the report [11], and we refer to this report as well for a theoretical explanation of linear regression and the performance metrics mean squared error (MSE) and

R^2 score. Logistic regression is treated in detail in this report.

For all methods we are going to split the data used into 80 % for training and 20 % for testing. The suitable performance metrics for the test data will then be used to find the best models and parameter combinations for the different methods.

For both gradient descent and, most notably, the feedforward neural network, we used code provided from the FYS-STK4155 syllabus [12], and adjusted it to our specific needs. The biggest difference between this source code and our implementation can be found in the back propagation algorithm and how weights and biases are handled.

1. *Testing and developing Gradient Descent*

We develop our own gradient descent and stochastic gradient descent codes, following Section II B. Implementing an option for using RMSprop, Adam, and Adagrad methods as described in the section.

We implemented a tuneable learning rate η for our stochastic gradient descent code. For our gradient descent code, we created a set learning rate that could be sped up by a factor r when wanted, by $r \cdot \eta$. The set learning rate η, was calculated by scaling the learning rate based on the most positive direction of the gradient of the function representing our data. This was done by finding the maximum eigenvalue of the Hessian matrix. We define the Hessian matrix as

$$H = \frac{2}{n} X^T X.$$

Here X is the design matrix and n is the number of rows in the design matrix. The eigenvalues of the Hessian matrix represent the most significantly curved direction and degree of the function it consists of, as described in the Medium article [13].

2. *Testing and developing FFNN*

As described in the section on feedforward neural networks, there are many different parameters to tune, in addition to choosing cost function, activation function, output function and the shape of our network. Therefore, we will only be able to explore a few of the multitude of different combinations, and we need a structured approach to this search.

We will use the MSE and R^2 score of our test data to measure the network's performance. Assuming that all other parameters are held fixed, we can use a grid search over the learning rate η and regularisation parameter λ to find the optimal combination of these two parameters. This grid search can then be repeated for different choices of network shapes and functions. In the end, we then compare and find the combination with the best MSE and R^2 score.

Our approach is outlined in Algorithm 7, and this scheme works equally well for both the regression and the classification case. We use nested loops over model shapes, activation functions, η values, and λ values. The cost functions and output functions are chosen to suit the problem and are not part of this search. Interesting regions of parameter space can then be examined further if we want to. Be aware that while this is in theory an easy and orderly way to search, it can quickly become too computationally demanding if the dataset is huge.

Algorithm 7 Structured approach to testing parameter combinations

```
Require: List of model shapes
Require: List of η values
Require: List of λ values
Require: List of activation functions
for each model shape do
    for each activation function do
        Initialise neural network with given model shape and activation function
        for each η do
            for each λ do
                Train network with given η, λ
                Make prediction of test data
                Calculate and store performance metrics
```

3. *Testing and developing logistic regression*

Our approach for doing logistic regression is similar to the one outlined for feedforward neural networks. The key difference is that the only parameters we need to tune are the learning rates η and the regularisation parameters λ. But since logistic regression uses a gradient descent method during training, we can also try different gradient descent methods. This approach, which is essentially only a grid search over η and λ, is shown in Algorithm 8.

Algorithm 8 Structured approach to logistic regression

```
Require: List of η values
Require: List of λ values
Require: List of gradient descent methods
for each gradient descent do
    for each η do
        for each λ do
            Train logistic regression with given η, λ
            Make prediction of test data
            Calculate and store the accuracy score
```

G. Data

For regression we will use the Franke function, given by

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

We generate data by drawing points $x, y \in [0, 1)$ from a uniform distribution and evaluating the Franke function at these points.

For classification, we will use breast cancer data from Wisconsin, available through Scikit-learn. The data is described on the website [14]. It contains 569 samples with 30 features each, and each sample is classified as either benign or malign. This means that we have a binary classification problem.

III. RESULTS AND DISCUSSION

A. Gradient descent

1. *Gradient descent*

In Figure 2, we have plotted the result of a gradient descent (GD) optimisation used to fit data from the Franke function (9) to an ordinary least squares (OLS) linear regression. As an illustration of how the learning rate affects the MSE of our model, we have plotted the MSE as a function of the learning rate. We tested (GD) with and without the use of momentum. The momentum was set to a value of 0.9 after a short trial and error, where it was determined that 0.9 was a sufficient momentum rate.

We observe that the GD model with momentum generally performs better than the GD model without momentum. The only exception is at a learning rate of $\eta = 10^{-6}$, where the MSE for GD with momentum spikes a little higher, this could be an indication of the GD momentum model getting stuck in a local minimum. In general, we see that the a higher η values gives a lower MSE both with and without momentum, with the most optimal learning rate appearing to be close to $\eta = 0.1$. As expected, gradient descent with momentum overall performs the best with the lowest MSE at 0.0341 for a learning rate of 0.1. The fixed η value calculated by finding the max eigenvalues of the Hessian matrix was $\eta = 0.194$ which gave an MSE of 0.0292.

In Figure 3 and Figure 4, we have plotted the gradient descent model with and without momentum using ordinary Ridge regression on our Franke model data. We observe that for Ridge regression as well, the MSE is lower in general for the gradient descent model with momentum. We also observe the similar trend that a higher learning rate will give a lower MSE, at least up until an $\eta = 0.1$.

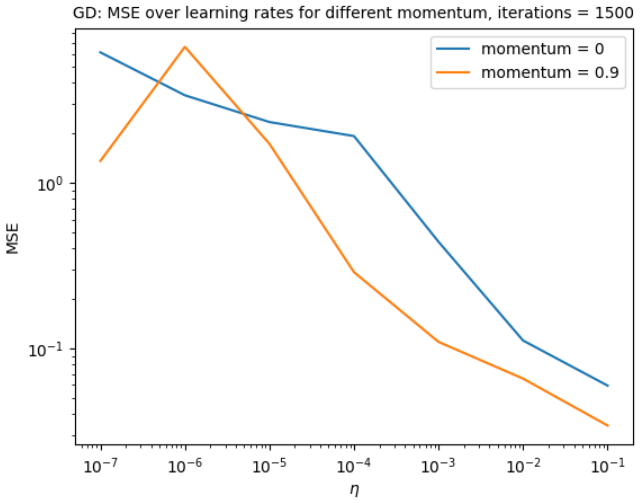

FIG. 2: Gradient descent model on Franke function data for 101×101 data points, using OLS regression. The plot shows variations in MSE for different learning rates, η, with and without the use of momentum.

2. *Stochastic gradient descent*

Figure 5 shows the mean squared error we get when using stochastic gradient descent to do linear regression on the Franke function. We see that there are several options for an optimally low MSE, so long as we pick a small enough batch size for the number of epochs we want to run through. We used the results in Figure 5 to determine an optimal batch size of $M = 35$, and used this for testing of Ridge regression used in stochastic gradient descent.

The results for ridge regression using stochastic gradient descent are presented in Figure 6. We notice that as λ approaches 0, the MSE decreases. This in addition to the results from Figure 5, indicates that the OLS method is the optimal linear regression method. This is similar to what was found in the report [11] on linear regression.

Overall, through our testing we determined that the regular gradient descent model with momentum and OLS regression outperformed the others. This lead us to continue using the GD model with momentum for our logistic regression.

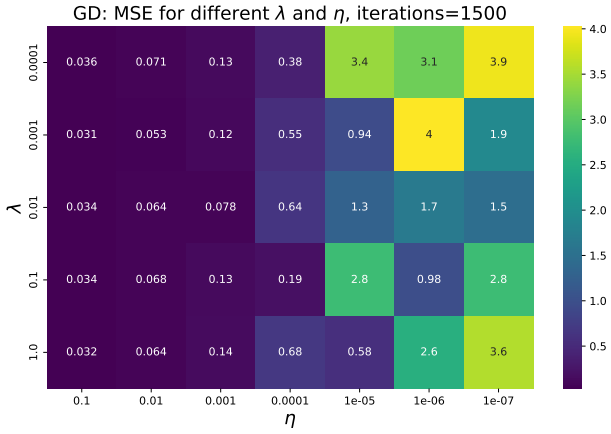

FIG. 3: Gradient descent model on Franke function data for 101×101 data points, using Ridge regression. The method used is standard gradient descent with momentum set to 0.9.

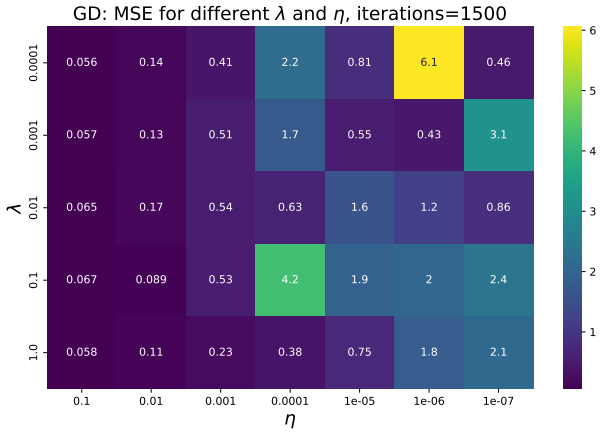

FIG. 4: Gradient descent model on Franke function data for 101×101 data points, using Ridge regression. The method used is standard gradient descent without momentum.

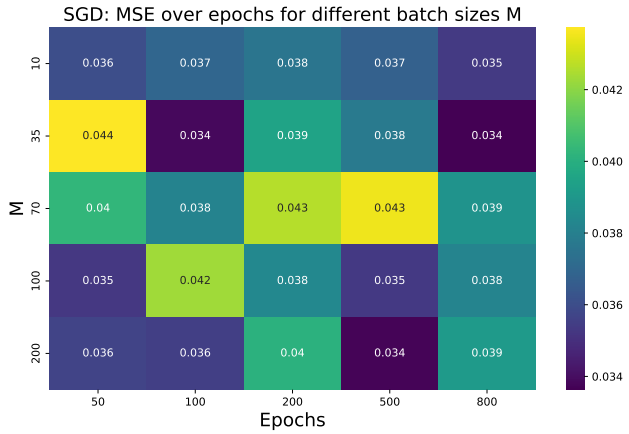

FIG. 5: Stochastic gradient descent model on Franke function data for 101×101 data points, using OLS regression. Heatmap showing MSE for variations in batch sizes, M, and epochs.

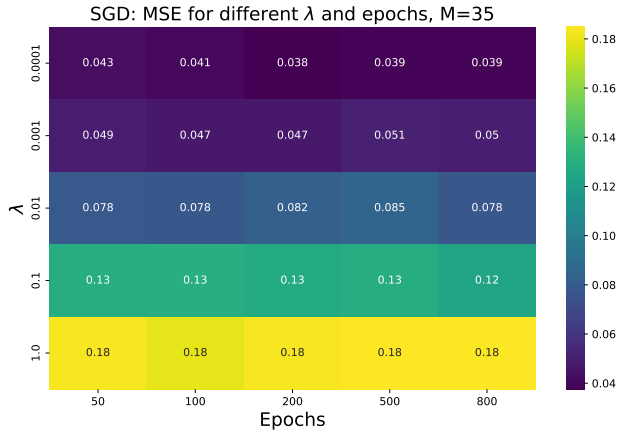

FIG. 6: Stochastic gradient descent model on Franke function data for 101×101 data points, using Ridge regression. Heatmap showing MSE for variations in, λ, and epochs for a batch size M=35.

B. Regression

After finishing the gradient descent, we are then ready to use our neural network. We train a feedforward neural network (FFNN), as described in Section II D, to fit to data generated by the Franke function (9). The data was generated on a grid of 101×101 points drawn randomly from a uniform distribution over $x, y \in [0, 1)$. We use 80 % of the data for training and 20 % for testing.

In the end we will compare the performance of our neural network with that of linear regression. Linear regression on the Franke function is detailed in the report [11]. As the data from the Franke function was normalised there, we will again normalise the Franke function in this study to get comparable results. Our chosen method of comparison is the test mean squared error.

Since we are here trying to mimic linear regression and want the mean squared error to be as low as possible, we choose the cost function to be the same as for ordinary least squares regression. This way, our network will directly try to find the lowest MSE value. As our function in the output layer we choose a function that does not change the output from the last hidden layer, namely $f(x) = x$. That is because we want continuous output, which we already have from the hidden layers. Therefore there is no need to change the output further.

We started by using the Algorithm 7 to do a broad search over many different parameters, training the net-

work with each combination and storing the test MSE. This way, we tested all combinations of parameters shown in Table I. Our gradient descent method of choice was non-stochastic Adam with the default parameters, see Section II B 4 for details on this method. We initially started the broad search with both bigger and smaller η values than those presented, but we immediately saw that these values gave very poor results and therefore abandoned them.

Parameter	Values tested
Hidden layers	(50), (100), (50, 50)
Learning rate η	$10^{-3}, 10^{-2}$
Regularisation λ	$10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1$
Activation function	Sigmoid, RELU, LRELU
Iterations	100, 200

TABLE I: All parameter used in the first training step of a neural network for the Franke function. All possible combinations of these parameters were tested. (n, m) for the hidden layers means one layer with n nodes and a second layer with m nodes, and so on.

For each combination of model structure, activation function and number of operations, we generate grid plots for the test MSE and test R^2 score over the possible combinations of the learning rate η and regularisation parameter λ. This allows for an easy way of comparing results. Examples of such plots are shown in Figure 7 for MSE and in Figure 8 for the R^2 score. These particular plots are for a model with hidden layers (50), the Sigmoid function as activation function and 200 iterations. Since we generated many such plots that are not particularly interesting, we will only include a few selected ones here.

From this search, we first observed that models with one hidden layer performed better that models with two layers. This was especially the case for the RELU and LRELU functions. We therefore decided to stick to models with one hidden layer. Additionally, we found $\eta = 10^{-3}$ to be the best learning rate.

Based on these observations we did a new round of training for the network. We used hidden layers (50) and (100), $\eta = 10^{-3}$, all activation functions and all λ values used in the first search. We increased the number of iterations to 500 and 1000, to see if that would increase the performance. We however interrupted the training with 1000 iterations, as the test MSE started to increase again.

Having thus narrowed down the candidates for best model to some combination with $\eta = 10^{-3}$, one hidden layer and 500 iterations, we conclude our testing of different parameter combinations. We find the best parameter combination among these we have tested by inspecting the grid plots. The RELU and LRELU functions with 50 nodes have the lowest test MSE, being nearly indistinguishable. But looking at the MSE with more decimals the best value for RELU is 0.1000 and the best value for LRELU is 0.1008.

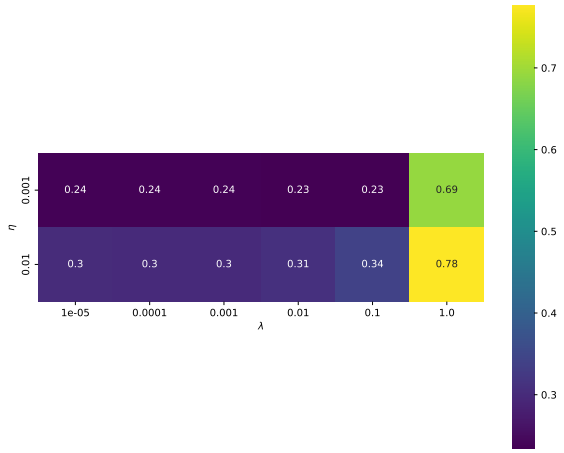

FIG. 7: Example of grid plot over learning rate η and regularisation parameter λ for the test MSE after training a feedforward neural network with a given model shape, activation function and number of iterations. See the text for details.

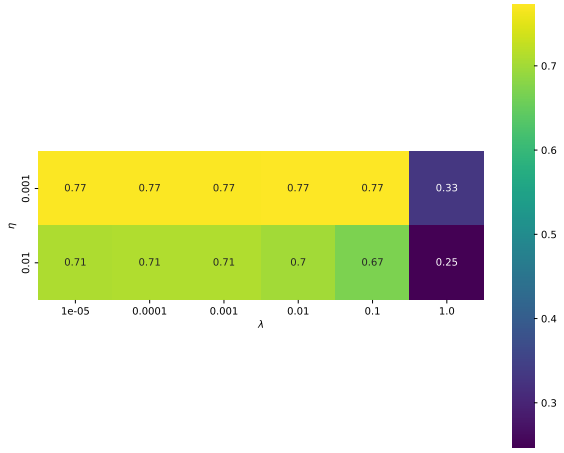

FIG. 8: Example of grid plot over learning rate η and regularisation parameter λ for the test R^2 score after training a feedforward neural network with a given model shape, activation function and number of iterations. See the text for details.

The test MSE for different values of the regularisation parameter λ for the RELU function with $\eta = 10^{-3}$, 50 nodes and 500 iterations are shown in Tables II and III, for the test MSE and test R^2 score, respectively. We see that $\lambda = 1$ gives significantly worse results that the others, and that either $\lambda = 10^{-2}$ or $\lambda = 10^{-1}$ give the

best result. Looking at more decimals we find that $\lambda = 10^{-2}$ gives a test MSE of 0.1001 and $\lambda = 10^{-1}$ gives a test MSE of 0.1000.

λ	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}
MSE test	0.120	0.121	0.117	0.100	0.100	0.222

TABLE II: The table shows how the test MSE depends on the regularisation parameter λ, after training a feedforward neural netwok on the Franke function. The network is trained with learning rate $\eta = 10^{-3}$, model shape (50), activation function RELU and 500 iterations.

λ	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}
R^2 score test	0.883	0.883	0.886	0.903	0.903	0.785

TABLE III: The table shows how the test R^2 score depends on the regularisation parameter λ, after training a feedforward neural netwok on the Franke function. The network is trained with learning rate $\eta = 10^{-3}$, model shape (50), activation function RELU and 500 iterations.

To sum up, best combination of parameters that we were able to find is presented in Table IV. The test MSE for this model is 0.1000 and the test R^2 score is 0.9030. Figure 12 shows a plot of the prediction we get for the Franke function with this model. For comparison, the true Franke function is shown in Figure 11. We see that the prediction is able to roughly capture the general trends of the Franke function, but it is far from perfect.

In light of the universal approximation theorem [8], we can say for certain that we have not found the best neural network for the Franke function. As the Franke function is continuous, and we have used the Sigmoid activation function, is should be possible to approximate it to arbitrary accuracy with only one hidden layer and a finite number of neurons.

Parameter	Best value
Hidden layers	(50)
Learning rate η	10^{-3}
Regularisation λ	10^{-1}
Activation function	RELU
Iterations	500

TABLE IV: The best combination of parameters we found for training our feedforward neural network on the Franke function.

Scikit-learn [10] has its own neural network implementation, and for comparison we also trained Scikit-learns neural network on the Franke function. We did a grid search over η and λ values for the model shapes (50) and (100), with 500 iterations and all three activation functions. The result for the RELU activation function are shown in Figures 9 and 10, for the test MSE and test R^2 score, respectively.

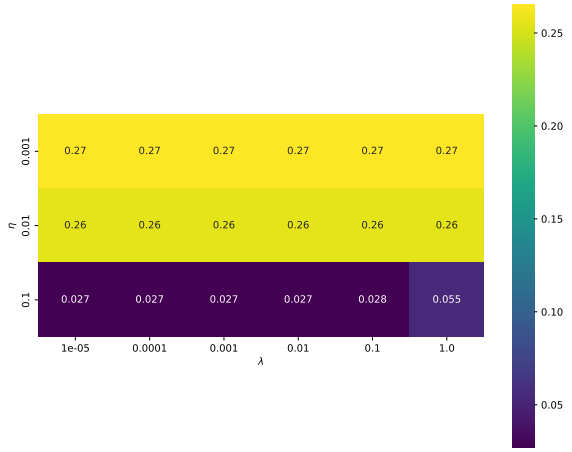

FIG. 9: Grid plot over learning rate η and regularisation parameter λ for the test MSE after training Scikit-learn's neural network for the Franke function. The network is trained with model shape (50), activation function RELU and 500 iterations.

We see that Scikit-learn's neural network performs best when $\eta = 0.1$, with one order of magnitude lower MSE and significantly better R^2 score than for the other η values. This differs greatly from how our network implementation behaves. We also see that Scikit-learn's network outperforms ours for the best parameter combinations we found. The fact that our results do not match those obtained with Scikit-learn indicate that our implementations differ in some way, but we did not have the time to investigate this further. It could for example be due to a different way of initialising weights and biases or differences in how the gradient descent or back propagation algorithm is implemented.

Lastly, we compare the performance of a neural network to that of an ordinary least squares regression. In the report [11] it was shown that an ordinary least squares regression on the Franke function could obtain a test MSE of 0.00112 and an R^2 score of 0.999. This is significantly better than our neural network performance and also better that Scikit-learn's neural network performance. Thus, it would seem that making a neural network is an unnecessary amount of work for little gain in the case of the Franke function, as a simple ordinary least squares regression easily outperforms it. Table **??** summarises the performances of the best models for the Franke function.

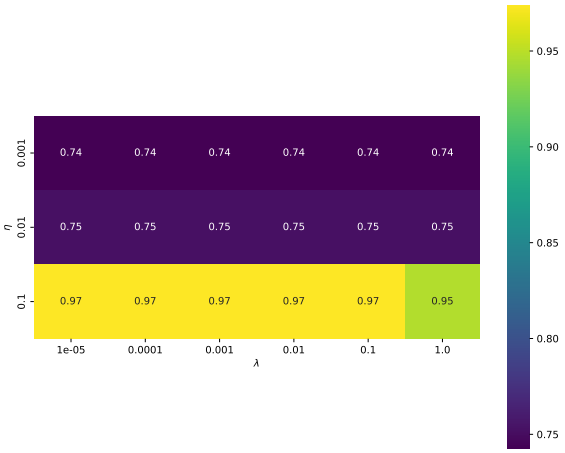

FIG. 10: Grid plot over learning rate η and regularisation parameter λ for the test R^2 score after training Scikit-learn's neural network for the Franke function. The network is trained with model shape (50), activation function RELU and 500 iterations.

	MSE test	R^2 test
FFNN	0.100	0.903
Scikit-learn	0.0273	0.974
Linear regression	0.00112	0.999

TABLE V: The performances of our best feedforward neural network (FFNN), the best feedforward neural network found with Scikit-learn and the best linear regression model from [11], when fit to the Franke function. See the text for details.

C. Classification

We will now use our feedforward neural network for binary classification, namely to classify the breast cancer data described in II G. Again the data is split into 80 % for training and 20 % for testing. We also normalise the data. Normalising is important in this case to ensure that the different features have approximately the same order of magnitude.

In the classification case we use the accuracy score (5) to determine performance of the models, or more specifically the accuracy score on the test data. The accuracy score is however not suited as a cost function, so we instead use the cost function (4), which is the standard cost function when doing binary classification. We use the Sigmoid function as our output function, as this will give us output between 0 and 1. Each output is then classified as 1 if the output is ≥ 0.5 and 0 if the output is < 0.5.

For training the neural network on classification data,

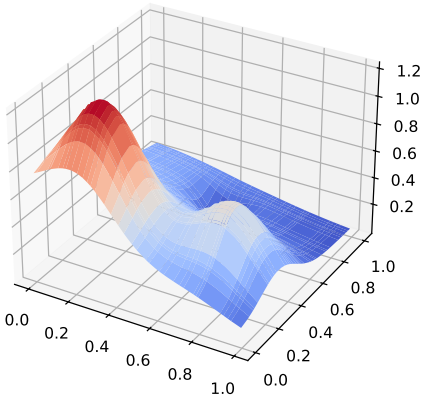

FIG. 11: The Franke function.

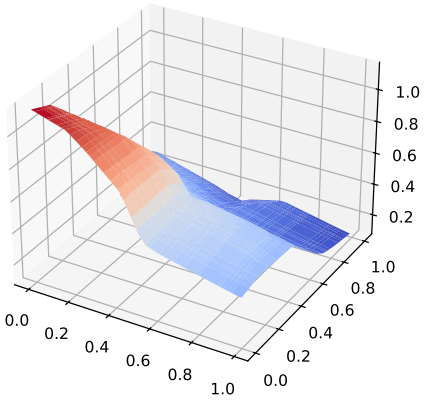

FIG. 12: Prediction of Franke function using a feedforward neural network trained with the best combination of parameters we were able to find, specified in Table IV.

we used the same approach as in the regression case. We searched Algorithm 7 over all combinations of the parameters given in Table VI. Bigger and smaller η values were disregarded because they gave larger MSE values across the board. The gradient descent method used is still Adam with default parameters.

We first observe that the network does not seem to train at all when $\eta = 0.01$, independent of the other parameters. The training accuracy score starts at 0.622 and stays 0.622 through all iterations. And increasing from 100 to 200 iterations makes the models worse for all λ values except $\lambda = 1$. Overall, only models with one hidden layer and 50 nodes reaches a performance that is

Parameter	Values tested
Hidden layers	(50), (100), (50, 50)
Learning rate η	$10^{-4}, 10^{-3}, 10^{-2}$
Regularisation λ	$10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1$
Activation function	Sigmoid, RELU, LRELU
Iterations	100, 200

TABLE VI: All parameter used in the first training step of a neural network for the Wisconsin breast cancer data. All possible combinations of these parameters were tested. (n, m) for the hidden layers means one layer with n nodes and a second layer with m nodes, and so on.

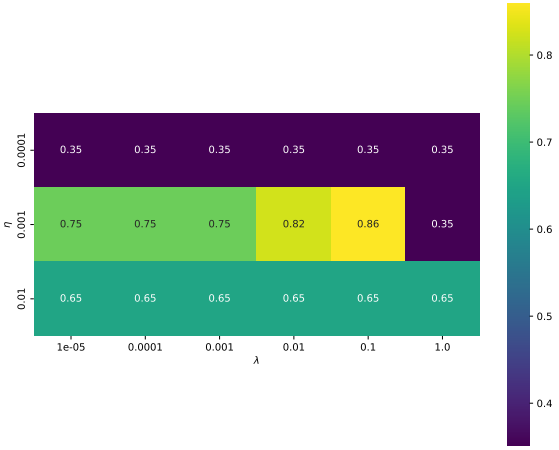

FIG. 13: Grid plot over learning rate η and regularisation parameter λ for the test accuracy score after training a feedforward neural network on breast cancer classification data. The network is trained with the Sigmoid activation function, model shape (50) and 100 iterations.

significantly better than the accuracy score of 0.622 that we get when the network does not train at all. Figure 13 shows the test accuracy of a neural network with one hidden layer and 50 nodes, 100 iterations and the Sigmoid function as activation function. Figures 14 and 15 shows the same results when the activation function is RELU and LRELU, respectively. These are our best results from the initial search, and we see that $\eta = 0.001$ gives the highest accuracy.

Therefore we trained the network again, this time only with model shape (50) and $\eta = 0.001$. We decreased the number of iterations to 50 to see if that would give a better performance, but it to the contrary became significantly worse. The results shown in Figures 13, 14 and 15 thus contain our best results. Upon inspection we see that we get the highest accuracy score when we have the parameter combination presented in Table VII. The test

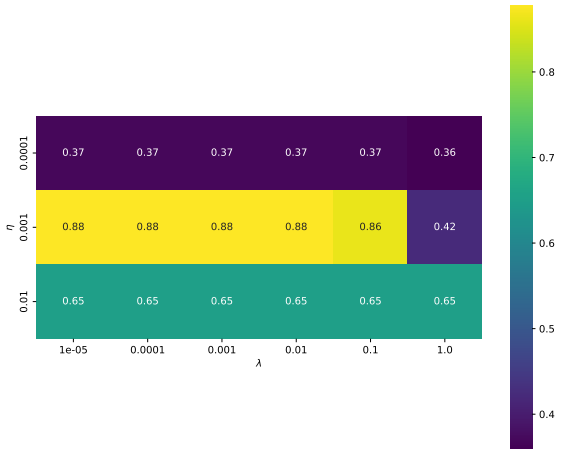

FIG. 14: Grid plot over learning rate η and regularisation parameter λ for the test accuracy score after training a feedforward neural network on breast cancer classification data. The network is trained with the RELU activation function, model shape (50) and 100 iterations.

accuracy is then 0.886.

Parameter	Best value
Hidden layers	(50)
Learning rate η	10^{-3}
Regularisation λ	$[10^{-5}, 10^{-4}, 10^{-3}]$
Activation function	LRELU
Iterations	100

TABLE VII: The best combination of parameters we found for training our feedforward neural network on the breast cancer classification data.

For comparison with Scikit-learn, a grid plot obtained with Scikit-learn's neural network is shown in Figure 16, corresponding to the same parameter combinations as in Figure 15. We see that skikit-learn's accuracy score generally is better. Specifically, we see no sign the that the network stops training the way we did for our own code, and $\eta = 0.01$ gives the best results. We also notice that $\lambda = 1$ gave significantly poorer results with our network, but not with Scikit-learn's.

Again, the mismatch of results indicate that there are some significant differences between our implementation and Scikit-learn's implementation, but we did not find out what. The best accuracy score we get with Scikit-learn is 0.921, for a model shape (50), $\eta = 10^{-2}$, $\lambda = [10^{-5}, 10^{-4}, 10^{-1}$, LRELU as activation function and 100 iterations.

We finally compare the neural network performance to the performance of standard logistic regression, as ex-

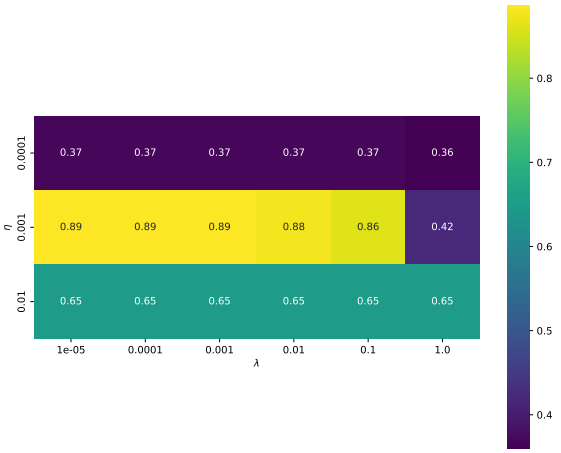

FIG. 15: Grid plot over learning rate η and regularisation parameter λ for the test accuracy score after training a feedforward neural network on breast cancer classification data. The network is trained with the LRELU activation function, model shape (50) and 100 iterations.

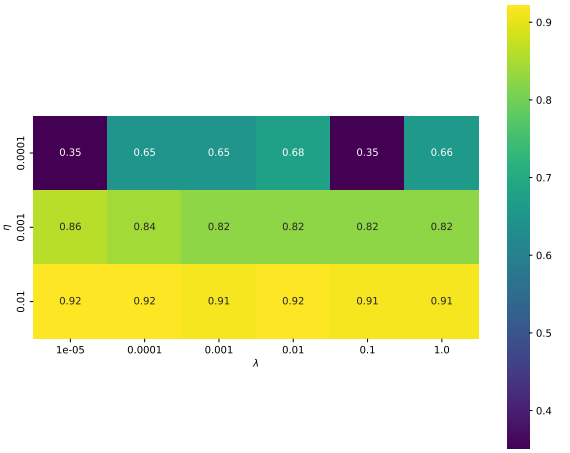

FIG. 16: Grid plot over learning rate η and regularisation parameter λ for the test accuracy score after training Scikit-learn's feedforward neural network on breast cancer classification data. The network is trained with the LRELU activation function, model shape (50) and 100 iterations.

plained in Section II C 1. We stick to the Adam scheme for gradient descent, as that is what we have done for the classification network as well. Performing a grid search over the learning rate η and regularisation parameter λ the same way as before yields the result presented in Figure 17. Here we have used 10^4 iteration, which is 2 order of magnitudes higher that the iterations needed for the neural network. Such a high number of iterations was necessary to achieve good performance.

As we can see, we only get good results for a few specific parameter combinations. The best combination by far is $\eta = 10^{-5}$ and $\lambda = 10^{-3}$, which gives a test accuracy score of 0.930. For almost all other combinations the optimisations appears to get stuck without ever getting close to the minimum cost function.

The same grid search using Scikit-learn's logistic regression is presented in Figure 18. Note that Scikit-learn's logistic classifier uses stochastic gradient descent, while our code uses Adam. Here we have used the same η and λ values, but achieved good results with only 100 iterations. Notably, Scikit-learn's implementation overall gives good results for λ values greater than or equal to 10^{-4}, which is opposite to our implementation. The best result is obtained with $\lambda = 10^{-3}$ and $\eta = 10^{-2}$. The test accuracy is then 0.939. The highest accuracy score obtained with Scikit-learn is slightly better than ours, but perhaps more important is the fact that Scikit-learn gives better models over a broader range of parameters. With our code the one parameter giving good performance could easily be missed.

A summary of the performance of the different classifications methods is presented in Table VIII. Both logistic regression methods are better than the neural network methods. This suggests that using a neural network for classification is unnecessary for the breast cancer data, as the neural network is both computationally more expensive and does not improve our results. However, for more complicated classification data sets the neural network might be able to outperform logistic regression, as one would expect the neural network to be more flexible.

	Accuracy score test
FFNN	0.886
Scikit-learn NN	0.921
Logistic regression	0.930
Logistic regression NN	0.939

TABLE VIII: The performances of our best feedforward neural network (FFNN), the best feedforward neural network found with Scikit-learn, our best logistic regression model and Scikit-learns best logistic regression model, when fit to the breast cancer classification data. See the text for details.

Another important observation is that we have not been able to implement a neural network or logistic regression that performs as well as their respective Scikit-learn counterparts. Exactly what causes these differences we cannot say, but there are many possibilities for tweaking both methods and hence many details that can differ

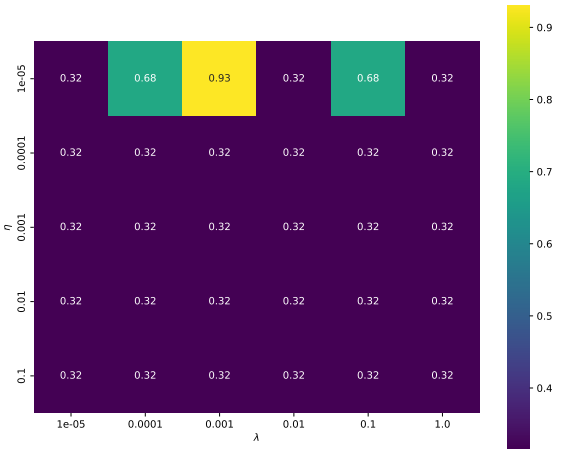

FIG. 17: Grid plot over learning rate η and regularisation parameter λ for the test accuracy score when using our implementation logistic regression on breast cancer classification data. Note that code is run with 10000 iterations, significantly more than all other models in this report.

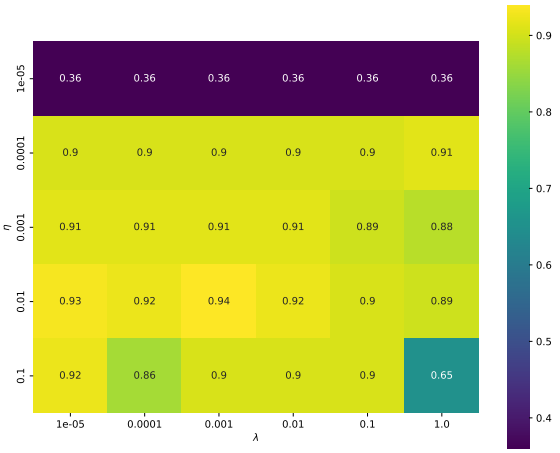

FIG. 18: Grid plot over learning rate η and regularisation parameter λ for the test accuracy score when using Scikit-learn's implementation of logistic regression on breast cancer classification data. The code is run with 100 iterations.

between the two. Based on what we have seen, our implementations are more prone to not train at all for many parameter combinations.

D. Possible improvements

There are several routes one could take to improve the work done in this report, and that would be interesting to investigate in the future. For example, we decided on the best model simply by looking at the test mean squared error or test accuracy score. It would be better to employ a more sophisticated validation technique, for instance bootstrap or cross validation.

We also did not have time to study the way we initialise weights and biases in our model, or to systematically test different gradient descent methods for the neural network. Both of these could be interesting topics for further research.

One could also include a stop criterion when training the neural network. This would make it unnecessary to try different number of iterations during training, as the training would take care of that itself. Additionally, a stop criterion could possibly save a lot of computational time as networks that have already converged would not train longer than needed, and networks that did not train at all could be stopped.

Additionally, there is room for improvement in how we search over parameter space, as our results are almost certainly not the optimal ones. While our method is easy to implement and understand, it can be computationally demanding and inefficient. It is also impossible to search the whole parameter space, so a more clever way of searching would perhaps have a better chance of finding optimal values. But the unpredictable behaviour when changing parameters and combining them in different ways also makes it difficult do an informed search.

Lastly, for future improvements we could have started with a simpler test model than the two-dimensional Franke function. The two-dimensional Franke function was useful when comparing with previous work, however, for initial testing of our gradient descent functions we struggled with a lack of intuitive understanding of the MSE plotted results. Using a one-dimensional polynomial during the early testing phase would have led to more easily comparable results, as an analytical solution would always be easily available and the plotted true model could be easily compared with the approximated model we created.

IV. CONCLUSION

We have in this study used neural networks to perform both regression and classification tasks. Gradient descent is an important part of the optimisation process, and during the testing and developing phase of our project, we found out that the gradient descent model with momentum out-performed the other tested variations of gradient descent. We then trained a feedforward neural network on data generated from the Franke function and on

Wisconsin breast cancer data. In both cases we used a variety of combinations of the parameters learning rate η, regularisation parameter λ, different activation functions for the hidden layers, different model shapes and varying number of iterations. To find the best parameter combinations we used test MSE and test accuracy score for regression and classification, respectively. The best parameter combinations are presented in Table IV for regression on the Franke function and in Table VII for classification of Wisconsin breast cancer data.

The performance of ordinary least squares regression on the Franke function was found to be significantly better than that of our neural network, but we know in light of the universal approximation theorem that we have not found the best model. Similarly for classification, a simple logistic regression outperformed the neural network, but again we might not have found the best parameter combination. In general there seems to be little to nothing to gain from using a neural network on these specific data sets. It introduces a more complex and computationally demanding model with no gain in performance. However, this might change for more complicated data that we cannot fit as well with standard regression methods.

Tables V and VIII shows the best performance of each type of model, for regression and classification, respectively. For all the methods discussed in this report, Scikit-learn's implementation performed better than our own implementation. Suggested improvements to our implementation include better validation, different initialisation of weights and biases and a better search over parameter space. However, Scikit-learn's neural networks still did not outperform the standard methods of ordinary least squares and logistic regression, further supporting that neural networks is not the best approach to fit the data we have studied.

[1] "Netflix recommendations: How netflix uses ai, data science, and ml," Last accessed: Nov. 12th 2023.
[2] M. Nielsen, *Neural Networks and Deep Learning* (Determination Press, 2015).
[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016) http://www.deeplearningbook.org.
[4] S. Boyd and L. Vandenberghe, *Convex Optimization* (Cambridge University Press, 2004).
[5] R. Kleinberg, Y. Li, and Y. Yuan, "An alternative view: When does sgd escape local minima?" (2018), arXiv:1802.06175 [cs.LG].
[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," (2017), last accessed: Nov. 12th 2023, arXiv:1412.6980 [cs.LG].
[7] X. Glorot and Y. Bengio, in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Proceedings of Machine Learning Research, Vol. 9, edited by Y. W. Teh and M. Titterington (PMLR, Chia Laguna Resort, Sardinia, Italy, 2010) pp. 249–256.
[8] G. Cybenko, Mathematics of Control, Signals, and Systems **2**, 303 (1989).
[9] D. R. Paxton Maeder-York, Adam Nitido and S. Sebbagh, "Autograd documentation," (2019), last accessed: Nov. 12th 2023.
[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, Journal of Machine Learning Research **12**, 2825 (2011).
[11] I. Monsen, V. H. Hvoslef, and L. Hansen, "An exploratory research into linear regression and resampling techniques for studying topographical data,".
[12] Morten Hjort-Jensen, "FYS-STK 4155 course material," (2023).
[13] Y. Ben, *Eigenvalues of the Hessian in Deep Learning: Singularity and Beyond* (Medium, 2023) last accessed: Nov. 16th 2023.
[14] "7.1. toy datasets," Last accessed: Nov. 18th 2023.