Classification and Regression
From Linear and Logisitc Regression to Neural Networks

Bendik Nyheim, Marcus Moen & Mira Mors
https://github.com/benyhh/FYS-STK4155

November 2021

Abstract

The goal of this work is to study both classification and regression problems by developing our own feed-forward neural network and regression code. First, the values of the Franke function are predicted using the stochastic gradient descent method, which will be a fundamental part of the other methods used. The mean square error obtained is 0.045. Results show that the stochastic gradient descent method converges faster by adding momentum and scheduling the learning rate. Additionally, the stochastic gradient descent method proves to be a good alternative for large data sets. Using a neural network with one hidden layer and 15 neurons, setting the learning rate to $\eta = 0.1$ and shrinking parameter to $\lambda = 0.0001$, the mean squared error can be reduced to 0.039. That is a 10% improvement from our previous work when using regular linear regression.
With the Wisconsin breast cancer data, our neural network for classification achieves 97% accuracy for predicting whether a tumor is malignant or benign. The recall score is one and the F1 score is 0.97. For benchmarking, we also implement logistic regression. Here, the accuracy is 95% and the recall and F1 scores are 0.99 and 0.96, respectively. All in all, our own neural network obtains the best results.

Contents

1 Introduction

We live in an information age where we are confronted with challenges concerning data storage, organization and searching. Hence a new type of "data mining" must be used in order to be able to analyse the vast amounts of data that are being generated. The aim is to extract important patterns and trends such that we can learn from the data. There are two types of machine learning problems, supervised and unsupervised. In this paper we focus on supervised learning where input measures are used to predict the value of an outcome measure.

An important aspect of machine learning is optimization. Learning algorithms are based on solving a mathematical problem through an iterative process. In doing so, suitable methods must be chosen that update the estimates of the solution. This paper presents stochastic gradient descent as an optimization operation.

The way in which data is represented has a tremendous impact on the performance of machine learning algorithms. Different pieces of information are grouped into so called features. It proves difficult to know which features to extract. Machine learning allows to not only map from representation to output, but find the representation itself. More precise, deep learning deals with expressing representation in terms of other, simper representations. One can think of deep learning as a tool to represent the world as a nested hierarchy of concepts. Each concept is defined in relation to simpler concepts. This makes the system very flexible. For this report we will use a feed-forward deep network.

Since machine learning is about finding the right method and benchmarking we will also use logistic regression. Logistic regression is another method to learn how certain features correlate with various outcomes.

In detail, we will first predict the values of the Franke function using stochastic gradient descent (with momentum), adding a learning schedule to the learning rate. Next, we perform regression using a feed-forward neural network to predict the values of the Franke function.
Then we move to classification using Wisconsin breast cancer data as input. We will make predictions using a feed-forward neural network and logistic regression. For all methods, we perform a grid search and benchmarking is also done using Scikit Learn.

2 Methods

2.1 Gradient Based Optimization

Optimization is often about minimizing or maximizing a function depending on its input parameters. In our case, we are interested in minimizing a type of error, a cost function. In the following, we present the basic principles of the gradient descent and stochastic gradient descent methods that allow us to find the values of the input parameters that minimize the cost.

2.1.1 Gradient Descent

Let the cost function be defined as $C(\beta)$, which we want to minimize by altering β. The derivative of the cost function tells us how to change β to reduce the error. By shifting β in small increments with the opposite sign of the derivative, we can lower the cost. In other words, the error is minimized by moving in the direction of the negative gradient. With multiple inputs, partial derivatives, the gradient, must be used. Given a starting value β_j, one gradient descent step is given by

$$\beta_{j+1} = \beta_j \underbrace{-\eta\nabla_\beta C(\beta)}_{\Delta\beta} \tag{1}$$

where $\Delta\beta$ represents the optimizing shift. η is the learning rate, a positive scalar which allows to scale the size of the step. (Ref. [2])

Cost Function Two cost functions used in the gradient descent method are presented below. (Ref. [1])

Linear Regression The cost for linear regression is given by

$$C(\boldsymbol{\beta}) = \frac{1}{2}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) \tag{2}$$

where $\mathbf{y}$ are the target values and $\mathbf{X}$ is the input data (design matrix). $\boldsymbol{\beta}$ is a vector of weights, where each parameter is associated with a particular feature in the design matrix. Rewriting in terms of sums instead of matrices

$$C(\boldsymbol{\beta}) = \frac{1}{2}\sum_{i=1}^{N}\left[y_i - \sum_{j=0}^{M}\beta_j x_{ij}\right]^2 \tag{3}$$

Given there are M different parameters to be optimized $(\beta_0, \ldots, \beta_M)$, the gradient to be computed comprises M different partial derivative components. One partial different component looks like

$$\frac{\partial C}{\partial \beta_j} = -\sum_{i=1}^{N} x_{ij}\left[y_i \sum_{k=0}^{M}\beta_k x_{ik}\right] \tag{4}$$

Ridge Regression The cost for ridge regression is given by

$$C(\boldsymbol{\beta}) = \frac{1}{2}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) + \frac{1}{2}\lambda\boldsymbol{\beta}^T\boldsymbol{\beta} \tag{5}$$

where λ is a shrinking parameter. Similar to the linear regression case, we write the cost function in terms of sums

$$C = \frac{1}{2}\sum_{i=1}^{N}\left[y_i - \sum_{j=0}^{M}\beta_j x_{ij}\right]^2 + \frac{1}{2}\lambda\sum_{j=0}^{M} w_j^2 \tag{6}$$

The partial derivative is now given by

$$\frac{\partial C}{\partial \beta_j} = -\sum_{i=1}^{N} x_{ij}\left[y_i - \sum_{k=0}^{M}\beta_k x_{ik}\right] + \lambda\beta_j \tag{7}$$

2.1.2 Stochastic Gradient Descent

The stochastic gradient descent method is based on the fact that the cost function consists of a sum over the training data

$$C(\beta) = \sum_{i=1}^{n} c_i(\mathbf{x}_i, \beta) \tag{8}$$

where $\{\mathbf{x}_i\}_{i=1}^{n}$ is the set of training points and β are the parameters to be optimized. Consequently, the gradient of the cost function is a sum over the i-gradients

$$\nabla_\beta C(\beta) = \sum_{i}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta) \tag{9}$$

To reduce computational power, the gradient, in each gradient descent step, is computed only for a subset of the given training data. The subset is commonly referred to as a minibatch, denoted B_k.

The data for a minibatch is uniformly drawn from the training set and its size is denoted by M. For n datapoints there will be n/M minibatches, hence $k = [1, n/M]$. Summarized, the gradient of the cost function in equation (9) is replaced by a sum over the randomly selected minibatches B_k

$$\nabla_\beta C(\beta) = \sum_{i=1}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta) \to \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta) \tag{10}$$

The gradient descent can now be written as (compare with equation (1))

$$\beta_{j+1} = \beta_j - \eta_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta) \tag{11}$$

where η is the learning rate. The number of iterations n is determined by a specific number called the epoch.
The randomness that the gradient descent method entails reduces the likelihood to optimize β according to a local (not a global) minimum. Given that the number of datapoints is significantly greater than the size of the minibatches, the computational cost is reduced. (Ref. [2])

Improving Performance The performance of the method depends strongly on the setting of the hyperparameters such as the learning rate. Usually, a grid search is performed testing different hyperparameters that differ by several orders of magnitude. Tuning the size of the minibatches and the number of epochs also affects the performance.

Scheduling the Learning Rate The step length η can be varied to depend on the number of epochs. The idea is that η should become very small after a reasonable time such that the descent stops. The formula is given by

$$\eta(t; t_0, t_1) = \frac{t_0}{t + t_1}, \qquad t = n \cdot m + i \tag{12}$$

where n denotes the current epoch, m the number of minibatches and $i = 0, \ldots, m-1$. t_0 and $t_1 > 0$ are two fixed numbers. (Ref. [4])

2.1.3 Momentum Stochastic Gradient Descent

Usually, the stochastic gradient decent method is done using a momentum based approach on the learning rate η. SGD with momentum is meant to increase the gradient in the right directions, which in turn leads to faster convergence. The simplest way of implementing this is

$$v_t = \gamma v_{t-1} + \eta \nabla_\beta C(\beta) \tag{13}$$
$$\beta_{t+1} = \beta_t - v_t, \tag{14}$$

where γ is the momentum parameter. This parameter takes a value between 0 and 1. This method has the effect of increasing the steps when the gradient is big (steep cost function). One of the reason the momentum approach is useful, is because the size of the step depends on the last step, which means the method is more likely to avoid stopping in a local minima. (Ref. [4])

Pseudocode The following algorithm shows how stochastic gradient descent with momentum can be implemented. For the actual implementation visit GitHub.

Algorithm 1: Algorithm for stochastic gradient descent with momentum

```
Result: Optimized parameter $\boldsymbol{\beta}$ for minimizing a cost described by $C(\boldsymbol{\beta})$
Require: Initial guess of $\boldsymbol{\beta}$, initial velocity $\mathbf{v}$
Require: Learning rate $\eta$, momentum parameter $\gamma$
while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\mathbf{x^{(1)}}, \dots, \mathbf{x^{(m)}}\}$ with
      corresponding targets $\mathbf{y}^{(i)}$
    Compute gradient estimate $\mathbf{g} \leftarrow \nabla_{\beta} c_i(\mathbf{x}^{(i)})$
    Compute velocity update $\mathbf{v} \leftarrow \gamma\mathbf{v} - \eta\mathbf{g}$
    Apply update: $\boldsymbol{\beta}_{j+1} \leftarrow \boldsymbol{\beta}_j + \mathbf{v}$
end
```

2.2 Neural Networks

An artificial neural network (ANN) consists of layers of interconnected neurons and thus mimics a biological nervous system in structure. The given data is fed into an input layer, processed and finally returned to the output layer. In this paper we will use an artificial feed-forward neural network (FFNN) to solve a regression and classification problem. But first, let us start by explaining the basic principles of a FFNN.

2.2.1 Feed-Forward Neural Network

The Feed-Forward Neural Network is a simple ANN where information always travels forward through the layers. The input for each neuron is given by the output of the neurons in the preceding layer. In principle, the number of neurons per layer and the number of layers can vary. Figure 1 a) shows the basic principle of a node. The node is represented by a circle and its input is a weighted sum of signals $x_i, \dots, x_n$ produced by the preceding nodes. The sum serves as an argument for an activation function, f, which gives the nodes's output y[1]. The layers between the input and output layers are called hidden layers. A set of multiple layers is shown in Figure 1 b). Note that a node receives input from all the nodes in the previous layer. (Ref. [4])

[1]Later in the paper the output y is denoted as a, the activation value

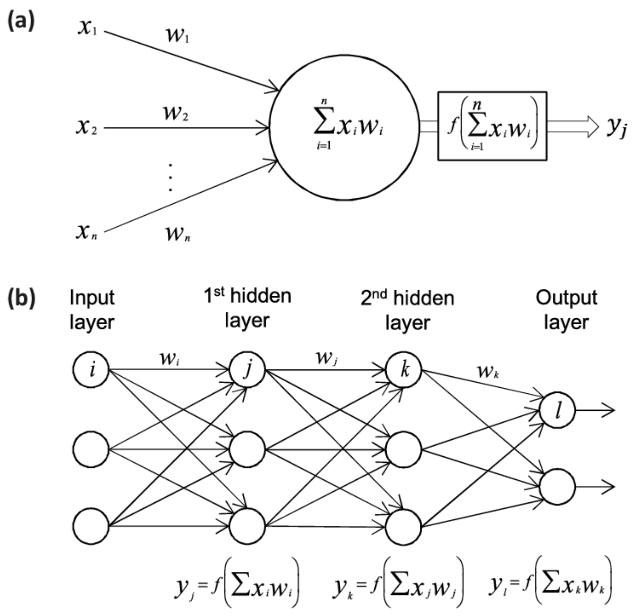


Figure 1: View of a feedforward artificial neural network where the artifical neurons are depicted as circular nodes. An arrow represents a connection from the output of one artificial neuron to the input of another. a) A node which takes in a weighted input $\{x_1, \ldots, x_n\}$ producing an output y. b) Network of multiple hidden layers with multiple nodes per layer where each node uses so called activation functions to compute the hidden layer values (Ref. [4])

The following is the mathematical explanation. Given there are N neurons in the previous layer $l-1$, the output a_i^l of a node i in layer l is defined as

$$a_i^l = f\underbrace{\left(\sum_{j=1}^{N_{l-1}} w_{ij} x_j + b_i\right)}_{z_i^l} = f(z_i^l) \tag{15}$$

where w_{ij} is the corresponding weight for node i of input x_j. b_i denotes the bias and ensures that the argument of the activation function is never zero. The activation function can be different for different layers and this is denoted by the index l

$$a_i^l = f^l(z_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l\right) \tag{16}$$

A typical activation functions is the sigmoid function. More on activation function in section 2.2.6. After the output is calculated for all nodes in layer l, it serves as a new input in the following layer until the last output layer i is reached.

Matrix-vector notation The arguments of the activation function can be written in matrix-vector notation. Thus, the output of a whole layer can be represented as an output vector

$$\mathbf{a}^l = f_l(\mathbf{W}_l \mathbf{a}_{l-1} + \mathbf{b}_l) = f_l\left(\begin{bmatrix} w_{11}^l & w_{12}^l & \cdots \\ \vdots & \ddots & \vdots \\ w_{N_l 1}^l & \cdots & w_{N_l N_{l-1}}^l \end{bmatrix} \cdot \begin{bmatrix} a_1^{1-1} \\ \vdots \\ a_{N_{l-1}}^{l-1} \end{bmatrix} + \begin{bmatrix} b_1^l \\ \vdots \\ b_{N_l}^l \end{bmatrix}\right) \tag{17}$$

where $\mathbf{W}$ stores the weights of node i in column i.
For example, the output of node i in layer l is just the i^{th} row of $\mathbf{a}^l$

$$a_i^l = f^l\left(w_{i1}^l a_1^{l-1} + w_{i2}^l a_2^{l-1} + \cdots + w_{iN_{l-1}}^l a_{N_{l-1}}^{l-1} + b_i^l\right) \overset{eq.(16)}{=} f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l\right) \tag{18}$$

Pseudocode The following algorithm shows the implementation of the feed-forward network. For the actual implementation visit GitHub.

Algorithm 2: Algorithm for forward propagation through an artificial neural network

```
Result: Predictions $\tilde{\mathbf{y}}$ given an input $\mathbf{x}$
Require: Network's architecture (number of layers $l$, number of neurons per layer)
Require: $\mathbf{W}^{(i)}, i \in \{1, \dots, l\}$, weight matrices of the model
Require: $\mathbf{b}^{(i)}, i \in \{1, \dots, l\}$, bias parameters of the model
Require: $\mathbf{x}$ (input data)
Randomly initialize all weights and bias
$\mathbf{h}^{\mathbf{0}} = x$ // values of input layer
for $i = 1, \dots, l$ do
    /* computation in hidden layers */
    $\mathbf{z}^{(i)} = \mathbf{W}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(\mathbf{i})}$ // weighted input to the nodes in hidden layer $i$
    $\mathbf{h}^{(i)} = f(\mathbf{z}^{(i)})$ // output from all the nodes of a hidden layer
end
$\tilde{\mathbf{y}} = \mathbf{h}^{(l)}$ // prediction at output layer
```

2.2.2 Classification

The task of the computer program is to predict to which of k categories or labels an input belongs. The learning algorithm should produce a function $f : \mathbb{R}^n \to \{1, \dots, k\}$. (Ref. [2]) In our case, we look at a binary case; true or false. How do we evaluate such a model? In classification we are not using the mean squared error (see equation (61)) to calculate the fit of our model. Here we will use something called an accuracy score. The accuracy score is defined to be the number of correct labels divided by the total number of guesses.

$$Accuracy = \frac{\sum_{i=1}^{n} I(ypred_i = y_i)}{n}, \tag{19}$$

where n is the number of guesses and $I(ypred_i = y_i)$ is the indicator function, which is 0 if the statement is false and 1 if the statement is true.

In our evaluation of the classification models we will also use something called precision, recall and F1-score. Precision is the amount of true positives, classified as 1 while correct label is 1, divided by the number of times the model classified something as 1. Recall is the amount of true positives divided by the total amount of positive samples in the test set. Intuitively precision is a score of how accurate we are when we say something is 1, while recall is a score of how often we missclassify something that should have been 1. It is important to both have high precision and recall scores. For example, in cancer detection we do not want someone to believe they have cancer when in fact they do not, while we do not want to missclassify someone with cancer either. The F1-score is the harmonic mean of the precision and the recall. Mathematically these measurements can be written like this:

$$precision = \frac{\sum_{i=1}^{n} I(ypred_i = 1 \text{ and } y_i = 1)}{\sum_{i=1}^{n} I(ypred_i = 1 \text{ and } y_i = 1) + I(ypred_i = 1 \text{ and } y_i = 0)} \tag{20}$$

$$recall = \frac{\sum_{i=1}^{n} I(ypred_i = 1 \text{ and } y_i = 1)}{\sum_{i=1}^{n} I(ypred_i = 1 \text{ and } y_i = 1) + I(ypred_i = 0 \text{ and } y_i = 1)} \tag{21}$$

$$F1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} \tag{22}$$

2.2.3 Regression

For regression, the aim is to find an algorithm which predicts a numerical value given some input. That is, the learning algorithm shall output a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Classification and regression therefore differ in the form of the output. The algorithm can be evaluated using the mean squared error or the R2 score (see section 6.1.2). In regression, a linear activation function $f(x) = x$ is used in the last layer, such that the output ($a_k = z_k$) in the last layer is the predicted value. Note that this is only the case when also using the squared error (23) as cost function (see also 2.2.5). The number of outputs is also equal to the number of inputs in the neural network, unlike classification, where the number of outputs are equal to the number of classes. (Ref. [2])

2.2.4 Back Propagation algorithm

The backpropagation algorithm is used to determine the optimized values for the weights and biases of a multilayer neural network with a fixed architecture. More precise, the neural network is trained by using, iterative, gradient-based optimizers. The cost function is driven to a very low value. In contrast to linear regression, no linear equation solvers are used. In our case, we are using the stochastic gradient descent method. The goal is to minimize the sum of squared errors between the initial values of the network and the given target values. The following shows how to change the weights and the bias. (Ref. [6])

Weight Optimization We start with optimizing the weights. For simplicity, let us look at a neural network with one input (I), one hidden (J) and one output layer (K). We define

- w_{kj} as a weight from the hidden to the output layer
- w_{ji} as weight form the input to the hidden layer
- a as the activation value
- t as a target value
- z as the net input

In regression, the most common cost function is the squared error. See section 2.2.5 for further details about the cost function. The reason for the factor $\frac{1}{2}$ will be apparent soon.

$$\mathcal{C}(W) = \frac{1}{2}\sum_k (t_k - a_k)^2 \tag{23}$$

where a_k denotes the final output of the neural network and t_k the corresponding target value. To reduce the overall error, we change the network's weights according to

$$\Delta W \propto -\frac{\partial C}{\partial W} \tag{24}$$

As the name of the backpropagation algorithm suggest, we start at the output layer. The adjustment for a particular weight, w_{kj}, from the hidden to the ouput layer is

$$\Delta w_{kj} \propto -\frac{\partial C}{\partial w_{kj}} \tag{25}$$

Since the cost function is not a direct function of a weight, we expand and get

$$\Delta w_{kj} = -\eta \frac{\partial C}{\partial a_k}\frac{\partial a_k}{\partial z_k}\frac{\partial z_k}{\partial w_{kj}} \tag{26}$$

where η is the learning rate. Note that merely the term in the cost function associated with the weight under consideration has a nonzero derivative.

Now, let us look at the derivative of the cost function with respect to the activation. Inserting the expression from equation (23) we have

$$\frac{\partial C}{\partial a_k} = \frac{\partial \left(\frac{1}{2}(t_k - a_k)\right)^2}{\partial a_k} = -(t_k - a_k) \tag{27}$$

The summation from equation (23) has been removed since the only time $\frac{\partial C}{\partial a_k}$ will be non-zero is when i is equal to k. If the cost function does not depend much on a particular output neuron, the derivative will be small.
Next, the derivative of the activation with respect to the net input is the derivative of the activation function

$$\frac{\partial a_k}{\partial z_k} = a_k^{'} \tag{28}$$

where the exact expression depends on the activation function used. Finally, the derivative of the net input with respect to a weight is given by the input associated with the respective weight

$$\frac{\partial z_k}{\partial w_{kj}} = \frac{\partial (w_{kj} a_j)}{\partial w_{kj}} = a_j \tag{29}$$

Again, only the term of the input associated with the particular weight will have a non-zero derivative.
Inserting the new expressions back into equation (26), the weight change for a hidden to output weight can be written as

$$\begin{aligned}
\Delta w_{kj} &= -\eta \underbrace{\frac{\partial C}{\partial a_k} \frac{\partial a_k}{\partial z_k}}_{\delta_k} \frac{\partial z_k}{\partial w_{kj}} && (30)\\
&= -\eta \underbrace{(-(t_k - a_k)) a_k^{'}}_{\delta_k} a_j && (31)\\
&= -\eta \delta_k a_j && (32)
\end{aligned}$$

where δ is the product of the error with the derivative of the activation function. Note that the exact expression for δ will depend on the cost function.
It remains to determine the weight change for an input to hidden weight. For this weight change we must consider the error at all the nodes the weighted connection can lead to. For this network, the connection leads to all the nodes in the output layer

$$\Delta w_{ji} \propto - \left[\sum_k \frac{\partial C}{\partial z_k} \frac{\partial z_k}{\partial z_j} \right] \frac{\partial z_j}{\partial w_{ji}} \tag{33}$$

where $\frac{\partial C}{\partial z_k}$ describes how the neurons in layer K affect the cost and $\frac{\partial z_k}{\partial z_j}$ describes how the neuron j in layer J affects the neurons in layer K. The application of the chain rule leads to

$$\begin{aligned}
\Delta w_{ji} &\propto - \left[\sum_k \frac{\partial C}{\partial a_k} \frac{\partial a_k}{\partial z_k} \frac{\partial z_k}{\partial a_j} \right] \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ji}} && (34)\\
&= -\eta \left[\sum_k \underbrace{(-(t_k - a_k)) a_k^{'}}_{\delta_k} w_{kj} \right] a_j^{'} a_i && (35)\\
&= -\eta \underbrace{\left[\sum_k \delta_k w_{kj} \right] a_j^{'}}_{\delta_j} a_i && (36)\\
&= -\eta \delta_j a_i && (37)
\end{aligned}$$

We have

$$\delta_j = \left[\sum_k w_{kj}\delta_k\right] a_k^{'} \tag{38}$$

Given multiple hidden layers, the expression generalises to

$$\delta_j^l = \left[\sum_k w_{kj}^{l+1}\delta_k^{l+1}\right] (a_k^{'})^l \tag{39}$$

where l and $l+1$ represent two consecutive hidden layers.

Bias Optimization Similar to the optimization of the weights, the bias are also updated. Accordingly, starting from the equation (24) we have to find

$$\Delta \mathbf{b} \propto -\frac{\partial C}{\partial \mathbf{b}} \tag{40}$$

For a given layer l and node j, the change of the cost function with respect to the bias can be expanded as

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{z_j^l}\frac{\partial z_j^l}{\partial b_j^l} \tag{41}$$

From the previous results (equations (34) to (36)) we recognize $\frac{\partial C}{z_j^l} = \delta_j^l$. In order to find an expression for $\frac{\partial z_j^l}{\partial b_j^l}$ we remind that $z_j^l = \sum_i w_{ij}^l a_i^{l-1} + b_j^l$. Taking the derivative with respect to the bias, the only term that contributes is the bias itself which is a constant. That is $\frac{\partial z_j^l}{\partial b_j^l} = 1$. Therefore

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{42}$$

More generally, for all biases in a layer we have

$$\frac{\partial C}{\partial b^l} = \delta^l \tag{43}$$

The change in bias is in a given layer l is

$$\Delta \mathbf{b}^l = -\eta \boldsymbol{\delta}^l \tag{44}$$

Now we have an expression for both the weight and bias change that we can plug into the equation (11) in order to update the weights and bias. That is, using stochastic gradient descent.

2.2.5 Cost Function

There are a number of loss and cost functions to use, and the optimal functions depends on the nature of the problem. The squared error is usually used for the regression. Usually, an additional term is added to the cost function that is proportional to the size of the weights. This can be seen as a constraint on the size of the weights so that they do not get out of control. The risk of overfitting is thus reduced. The size of the weights is measured using the so-called L2 norm. The new cost function has the same form as equation (6).
For classification we use the so called cross entropy cost function at the final layer

$$C(\boldsymbol{W}) = -\sum_{i=1}^{n}\left(t_i \log a_i^L + (1-t_i)\log\left(1-a_i^L\right)\right) \tag{45}$$

where t_i are the targets. The derivative of the cost function with respect to the output a_i^L then is

$$\frac{\partial C(\boldsymbol{W})}{\partial a_i^L} = \frac{a_i^L - t_i}{a_i^L(1 - a_i^L)} \tag{46}$$

This is the same equation as for the logistic regression described below. (Ref. [5])

2.2.6 Activation Functions

In a Neural Network we need activation functions. There are multiple different activation functions to choose from. In this paper we will only look at the four most common, which are the sigmoid, softmax, ReLU, and Leaky-ReLU. (Ref. [5])

We can start by looking at the **sigmoid** activation function, also called the logistic activation function. The sigmoid activation function is used to map any value on the continuous real line to a value between 0 and 1.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{47}$$

Softmax is a way of turning a list of values into a list of probabilities. This activation function is mostly used at the output layer of a multi-class classification. Quite simply it says that the probability of any given value in the list is the exponential of that value divided by the sum over the exponential of all the values in the list. In other words,

$$\sigma(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j^K e^{z_j}}, \text{ for } i = 1, \dots, K, \tag{48}$$

where $\boldsymbol{z} = (z_1, \dots, z_K)$.

The most common activation function used now is the **Rectified Linear Unit**, or **ReLU**, activation function. This is a rather simple activation function which maps all negative values to 0 and all positive values to itself. More mathematically it is:

$$R(x) = \begin{cases} x & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{49}$$

The biggest problem with the ReLU function is that all negative values disappear (maps to 0). **Leaky-ReLU** is a way of combating this problem. As the name suggest this is a leaky version of the ReLU function. What we mean by leaky is that all negative values will now be close to 0 but not quite. The most common approach is the multiple the nagative values with a constant a, often 0.01.

$$leakyR(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases} \tag{50}$$

The last activation function to be mentioned in this paper is the **linear**. This will only be used in the last layer when using the neural network for regression. The reason of this is that it makes the derivative of the cost function with respect to the input z in the last layer easier to compute. The linear activation function is

$$f(x) = x \tag{51}$$

2.2.7 Pseudocode

The following algorithm show the implementation of the backward propagation. For the actual implementation visit GitHub.

Algorithm 3: Backward propagation for training an ANN. Starting from the output layer and working backwards to the first hidden layer, the gradient on the activations is calculated for each layer i. The gradients indicate how each layer's output should change to reduce the error.

```
Data: Backward propagation uses output from forward computation
Result: Update of weights and bias in order to reduce the final error
Require: $\tilde{\mathbf{y}}$ (predictions from forward propagation), $a_i$ (activation values at layer $i$)
Require: $\mathbf{y}$ (target output)
Require: $\eta$ (learning rate)
$Loss = C(\tilde{\mathbf{y}}, \mathbf{y})$ // compute total error
Compute $\delta_l$ // product of the error with the derivative of the activation function
$w^{(l)} \leftarrow w^{(l)} - \eta\delta_l a_{l-1}$
$b^l \leftarrow b^l - \eta\delta^l$
for $i = l, l-1, \ldots, 1$ do
    /* loop backward through network converting the gradient on the layer's output into a
       gradient on the prenonliearity activation */
    compute $\delta_i$
    $w^{(i)} \leftarrow w^{(i)} - \eta\delta_i a_{i+1}$
    $b^i \leftarrow b^i - \eta\delta_i$
end
```

2.2.8 Improving Performance

The number of layers, the neurons per layer, and the activation function used are examples of parameters that must be determined and that can affect the accuracy of the result.

Starting Values The weights are usually initialized to have values close to zero. This is done by drawing samples from the standard normal distribution, which results in the operational part of the sigmoid being approximately linear. Consequently, the neural network collapses into an approximately linear model. In backpropagation, individual weights are increased so that nonlinearities arise where they are needed. If the initial weights were set to zero, the derivatives would be zero and the algorithm would never move. Large weights, on the other hand, often lead to poor results. Therefore, we also add the l2 regularization for the weights in the backward propagation (see section 2.2.5). The bias is set to 0.01 to secure that the input into the activation function is not zero.

Scaling Scaling can strongly influence the quality of the final solution. The scaling of the inputs determines the effective scaling of the weights in the bottom layer. Normally, the mean of the input is set to zero and the standard deviation is set to one. This way, all inputs are evaluated equally in the regularization process. All weights can thus be initialized randomly but uniformly close to zero.

2.3 Logistic Regression

Another, rather common, way of doing classification is to use logistic regression. Logistic regression is most often used for classification problems with two categories. It is possible to use logistic regression for more than two categories, but this is not something we will look into here. One can look at logistic regression as a form of converting a linear model to a probability value between 0 and 1. The way this is done in is to input a linear combination of the input variables into the sigmoid function, see the subsection on activation functions 2.2.6 for more information about this function. The sigmoid function takes a real value x and turns it into a probability value between 0 and 1. This probability value is interpreted as the probability of the correct label being 1.

In logistic regression we have defined the cost function to be:

$$C(\beta) = \frac{1}{n}\sum_{i=1}^{n} -y_i \log(\sigma(\mathbf{x}_i\beta)) - (1-y_i)\log(1-\sigma(\mathbf{x}_i\beta)) \tag{52}$$

This cost function has a rather nice derivative. Without doing the whole calculation, the derivative of the cost function is:

$$\frac{\partial C}{\partial \beta_j} = \frac{1}{n}\sum_{i=1}^{n}(\sigma(\mathbf{x}_i\beta)) - y_i)\mathbf{x}_i^{(j)} \tag{53}$$

where $\mathbf{x}_i^{(j)}$ is the j'th element of the input variable vector $\mathbf{x}_i$.

By using the cost function, and its derivative, we can use stochastic gradient decent to find the optimal values for β. The β values are updated as follow until convergence:

$$\beta_j := \beta_j - \alpha\sum_{i=1}^{n}(\sigma(\mathbf{x}_i\beta)) - y_i)\mathbf{x}_i^{(j)}$$

where α is the learning rate.

After minimizing this function and obtaining the optimal β values for the given data we can make our predictions based on the σ function. This is often done by computing the value and classify the point as 1 if $\sigma(\mathbf{x}) > 0.5$ or 0 otherwise. (Ref. [3]) See GitHub for the code implementation.

3 Results

3.1 Stochastic Gradient Decent

To create the dataset, we generated $N = 500$ random x_1 values and x_2 values with the uniform distribution from 0 to 1 and determined the corresponding y values using the definition of the Franke function (see equation (63)). To the y values, we added additional noise. This noise is generated using the normal distribution, $\mu = 0$ and $\sigma = 0.2$. The design matrix was set to be a two-dimensional polynomial of degree five (see section 6.1.1). The data was then split into a training set and a testing set, with 80% of the data used for training and the rest for testing. The data was scaled using Scikit Learn's Standard Scaler where we chose the deviation to be one. This has to do with the fact that the input data are between 0 and 1 and therefore precision errors can occur for values close to zero. Predictions were done using equation (57). For the stochastic gradient descent (with momentum), the squared error cost function and the ridge regression cost function were chosen as the loss function (see 2.1.1 section). A grid search was performed to find the optimized hyperparameters. Results are given for the test set.

Table 1 shows the mean squared error between the predictions and the true Franke-values, using either the squared error or the ridge cost function for optimization. The initial estimate of β was randomly determined using a standard normal distribution. The results were calculated using 500 epochs. For the squared error case, the learning rate was $\eta = 0.01$ and the number of mini-batches 100. For the case where the ridge cost function the additional shrinkage parameter was $\lambda = 0.001$. For comparison Scikit Learn's stochastic gradient descent results are also shown. For Scikit Learn, the input parameters were the learning rate, the shrinkage parameter and the number of epochs.

Figure 2 depicts the mean squared error between the predictions and the true Franke-values as a function of epochs for different values of the momentum parameter γ (see equation (13). Computations were done with and without a learning rate schedule (see equation (12)). The loss function for optimization was the ridge cost function.

The heatmaps in figure 3 show the mean squared error as a function of different hyper parameters. For the OLS optimization, it is shown as a function of the learning rate and number of minibatches. For the ridge cost function, the error is shown as a function of the number of minibatches and shrinking parameter λ 3b. The learning rate $\eta = 0.1$. 3c shows the error as a function of the shrinkage parameter and learning rate (batch size 5). 5000 epochs were used for every calculation.

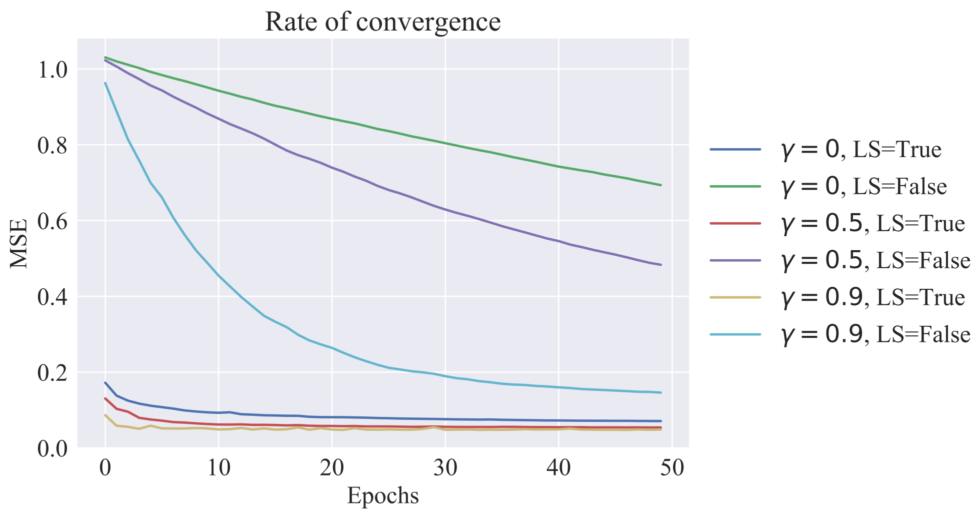
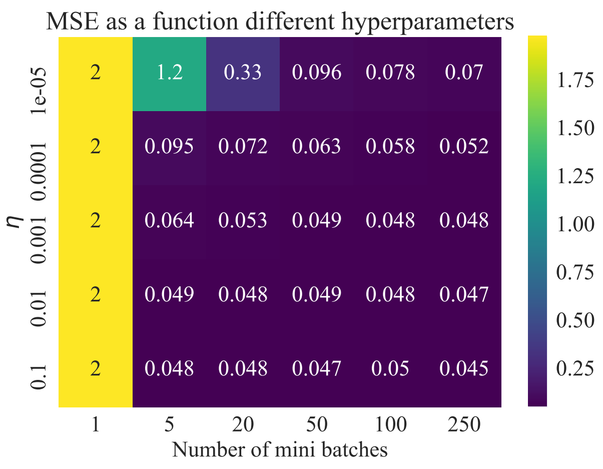


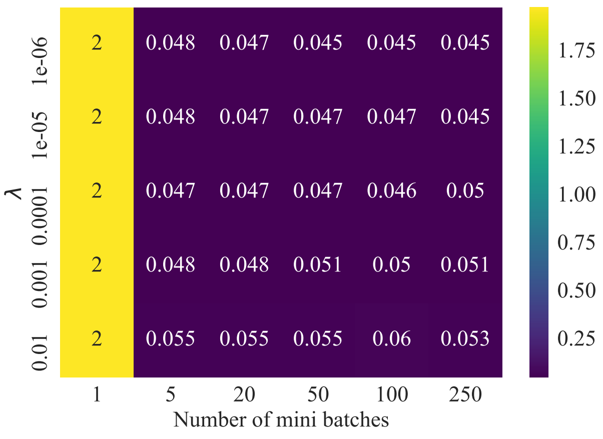
Figure 2: Prediction of values of the Franke function using stochastic gradient descent (with momentum). The figures show the mean squared error between true and predicted Franke values as a function of epochs for different values of the momentum parameter γ and both with and without a learning schedule (LS) to scale the learning rate η. The training data is split into 100 mini-batches with 5 samples each.

Table 1: Mean squared error for the testing set from predicting the values of the Franke function using a two dimensional polynomial of degree five. Predictions were done using stochastic gradient descent (SGD) with 500 epochs, learning rate $\eta = 0.01$. For the gradient of the ridge cost function, we used shrinking parameter $\lambda = 0.001$. Both our own implementation and results from SKlearn's SGDRegressor are presented.

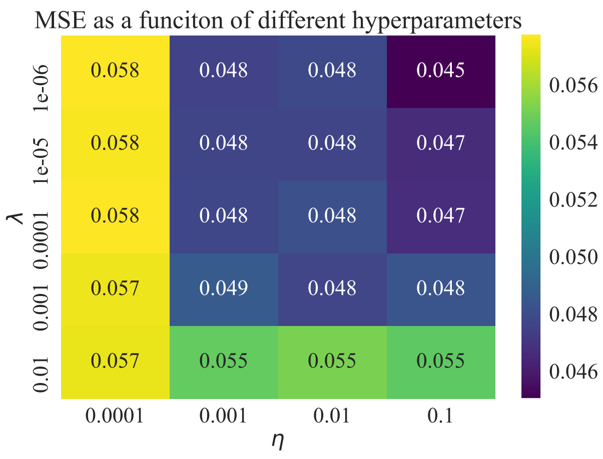
	MSE	
	OLS	Ridge
SGD	0.052	0.056
SKLearn SGD	0.053	0.053



(a) OLS



(b) Ridge - $\eta = 0.1$



(c) Ridge - number of mini batches 100

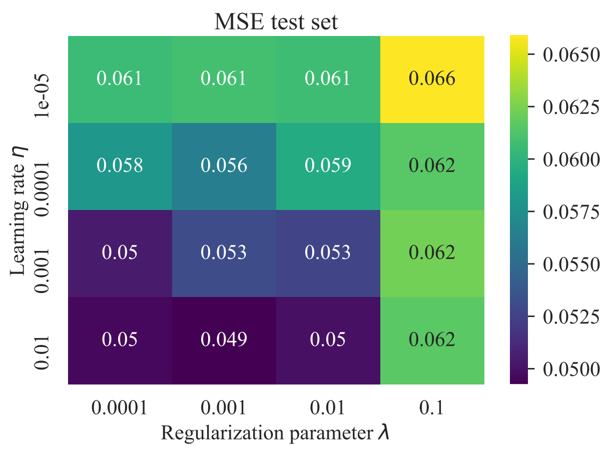
Figure 3: Mean squared error for predicting the Franke-function values using stochastic gradient descent as a function of the learning rate η, number of minibatches and shrinking paramter λ. Each heatmap uses either the squared error (OLS) or ridge cost function for optimization. The number of epochs is set to 5000.

3.2 Neural Network

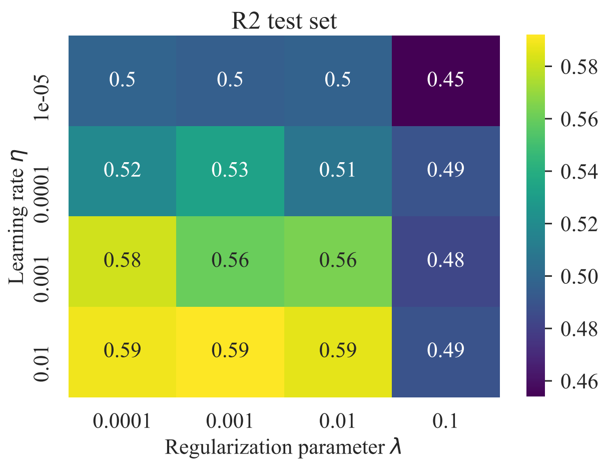
3.2.1 Regression

We used our neural network to do regression on the Franke function. The data were constructed similarly to the stochastic gradient descent case (see 3.1). The only difference is that the design matrix was a first degree polynomial. That is, no assumptions were made about the shape of the model.

Figures 4-7 present the MSE and R2 score when making prediction on test data using different activation functions in the one and only hidden layer (for the definition of MSE and R2 see 6.1.2). The heatmaps show the result for different values of learning rate η and regularization parameter λ. Other hyperparameters used are number of epochs set to 5000, batch size 50, and one hidden layer with 10 neurons.
Further grid search showed (see figure 15) that the lowest error is obtained for the relu activation function with $MSE = 0.039$ and $\eta = 0.01, \lambda = 0.0001$ using 15 neurons for one hidden layer.

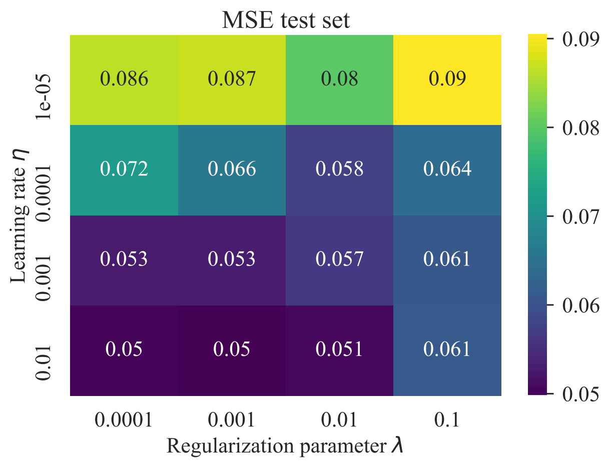


(a) One hidden layer

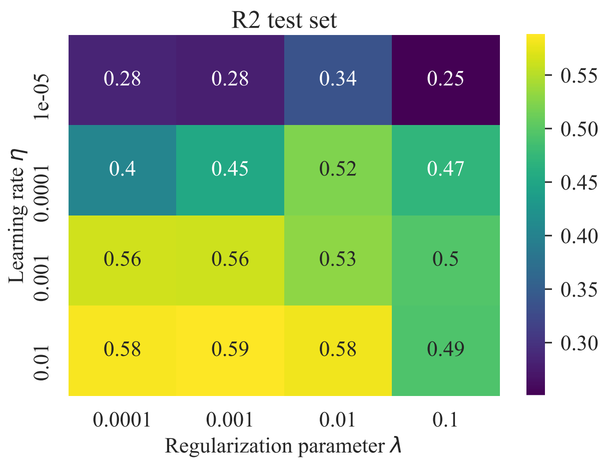


(b) One hidden layer

Figure 4: MSE and R2 score for predicting the values of the Franke function using neural network regression with stochastic gradient descent with one hidden layer containing 10 neurons. The activation function in the hidden layers is the sigmoid. Number of epochs is set to 5000 and batch size is 50.

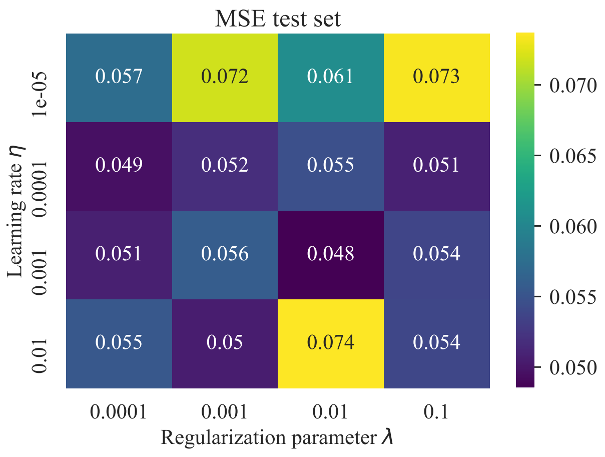


(a) One hidden layer

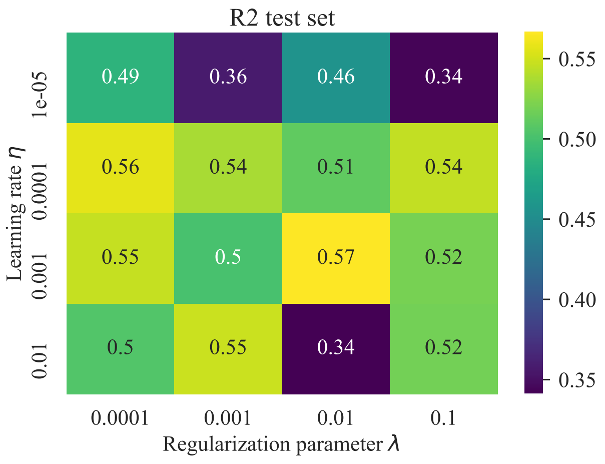


(b) One hidden layer

Figure 5: MSE and R2 score for predicting the values of the Franke function using neural network regression with stochastic gradient descent with one hidden layer containing 10 neurons. The activation function in the hidden layers is the softmax. Number of epochs is set to 5000 and batch size is 50.

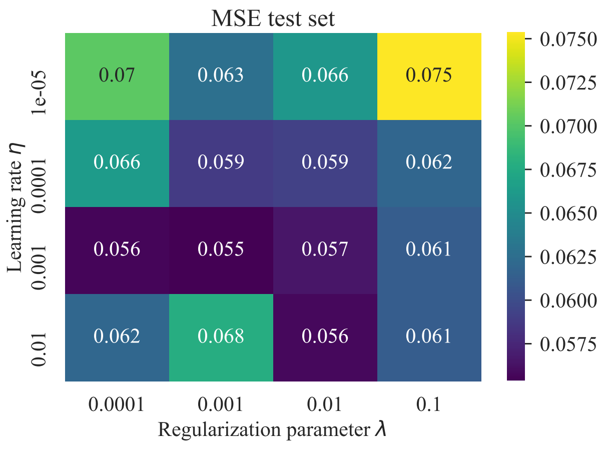


(a) One hidden layer

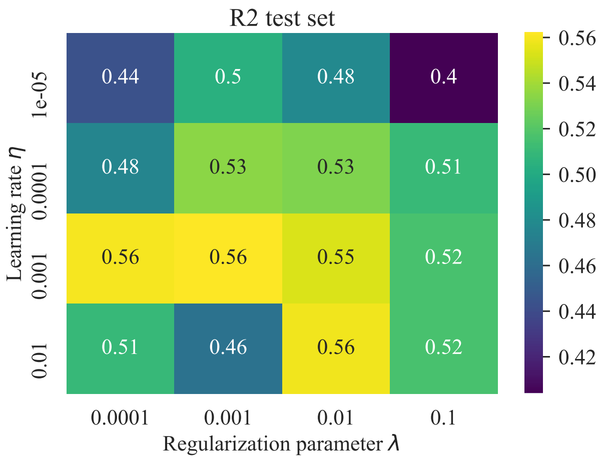


(b) One hidden layer

Figure 6: MSE and R2 score for predicting the values of the Franke function using neural network regression with stochastic gradient descent with one hidden layer containing 10 neurons. The activation function in the hidden layers is the relu. Number of epochs is set to 5000 and batch size is 50.



(a) One hidden layer



(b) One hidden layer

Figure 7: MSE and R2 score for predicting the values of the Franke function using neural network regression with stochastic gradient descent with one hidden layer containing 10 neurons. The activation function in the hidden layers is the leaky relu. Number of epochs is set to 5000 and batch size is 50.

3.2.2 Classification

The Wisconsin breast cancer dataset, a binary classification dataset with 30 features and a total of 589 samples, served as input. The two classes are benign and malign breast cancer tumours, which are labeled as 0 and 1 respectively. The data was again split into training and testing, with 80% of the data used for training. The standard scaler from Scikit Learn was used to scale the data. Common to all our neural network results is the number of epochs of 50 and the batch size of 10. The exception is figure 12, where the number of epochs is 1000.

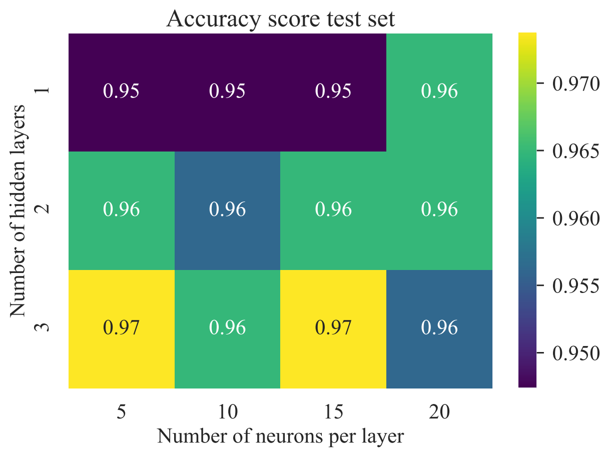
We trained our neural network with different parameters and made heatmaps to see the effect. Figure 8 shows four heatmaps using different activation functions for the hidden layers. Each of them shows the prediction accuracy on the test data for different combinations of number of hidden layers, and number of neurons per hidden layer. The other hyperparameters are learning rate $\eta = 0.01$ and regularization parameter $\lambda = 0.001$.

Figure 10 presents tests of other hyperparameters. In both heatmaps, the number of hidden layers are 3, and the number of neurons per layer varies widely. The activation function in the hidden layers is the sigmoid for both runs. The difference of the two subfigures is that one has a constant regularization parameter λ and varies the learning rate η. 10a, and the other has a constant learning rate η and varies the regularization parameter λ 10b.

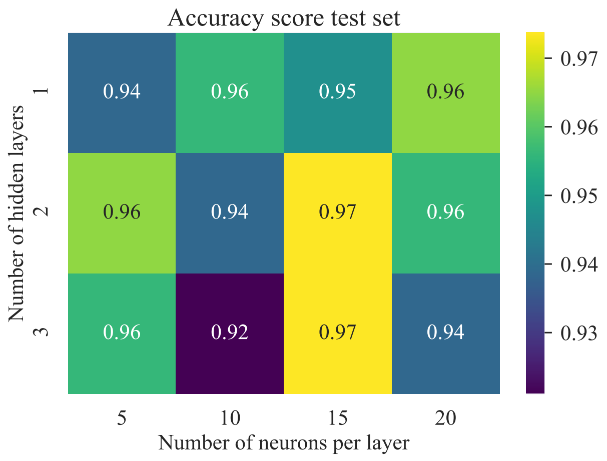
Similarly, figure 9 shows how the different activation function perform when keeping the number of hidden layers and neurons in each layer constant, while varying the learning rate η and regularization parameter λ. There are two hidden layers with 15 neurons in each layer.

A comparison of our own implementation of a feed-forward neural network and SKlearn's implementation is shown in figure 11. It shows two confusion matrices of the predictions on the test data set. We used the same parameters for SKlearn and as for our own NN: Learning rate $\eta = 0.01$, regularization parameter $\lambda = 0.001$, three hidden layers, with 50, 20 and 50 neurons respectively.

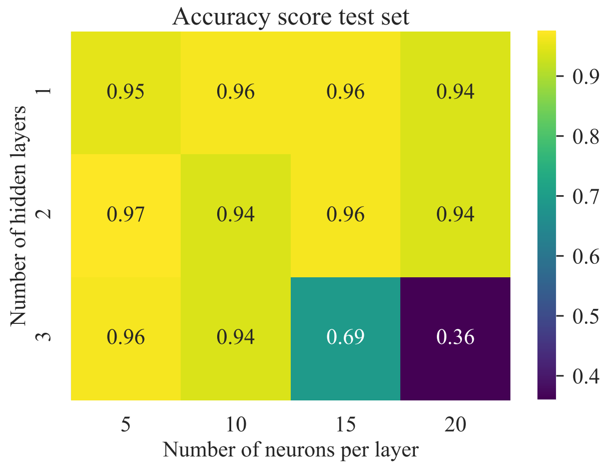
Figure 12 shows the prediction accuracy on training data as a function of epochs for two different learning rates. The sigmoid function is used as activation function, and there are three hidden layers with 50, 20 and 50 neurons respectively. The regularization parameter $\lambda = 0.001$ and the batch size is 10. As the plots show, there are 1000 epochs.



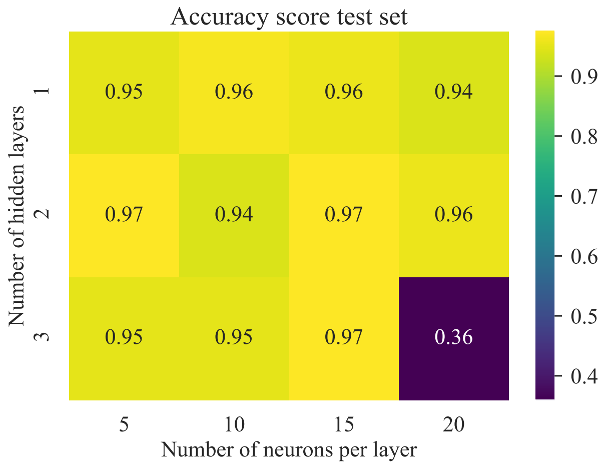
(a) Sigmoid



(b) Softmax

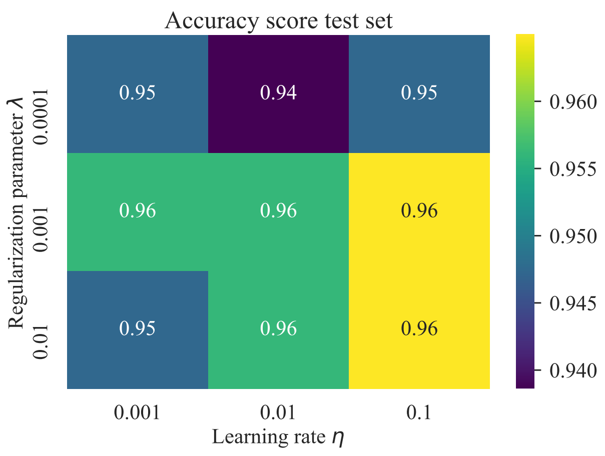


(c) Relu

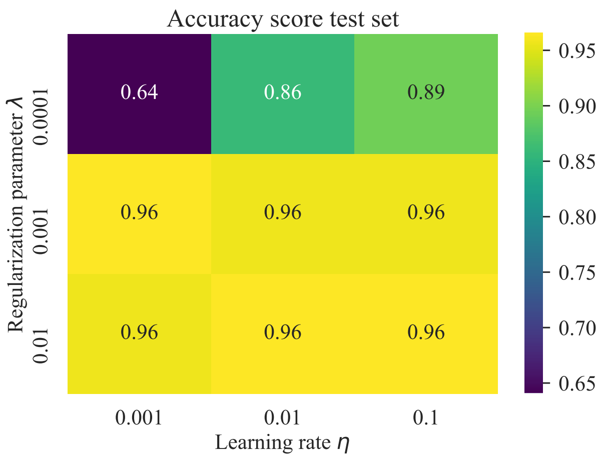


(d) Leaky Relu

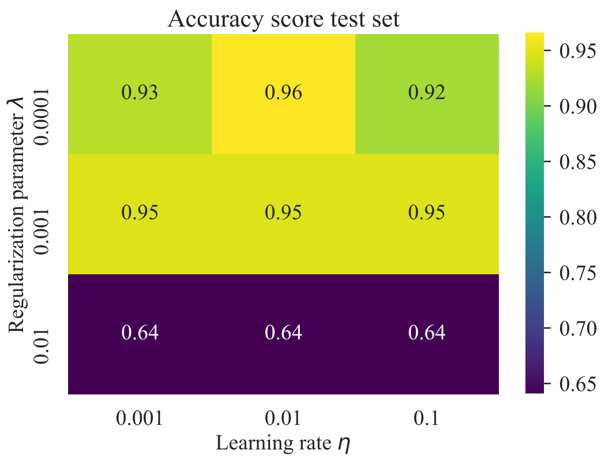
Figure 8: Prediction accuracy of neural network with stochastic gradient descent on test set using breast cancer data. The subfigures show the result for different activation functions, different number of hidden layers and number of neurons per hidden layer. Epochs is set to 50 and batch size is 10. Regularization parameter $\lambda = 0.001$ and learning rate $\eta = 0.01$.



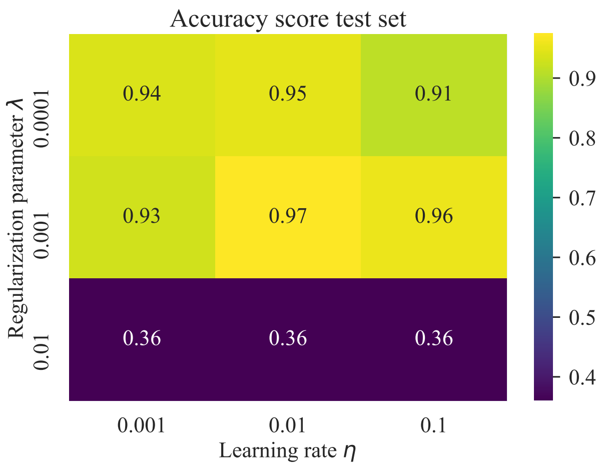
(a) Sigmoid



(b) Softmax

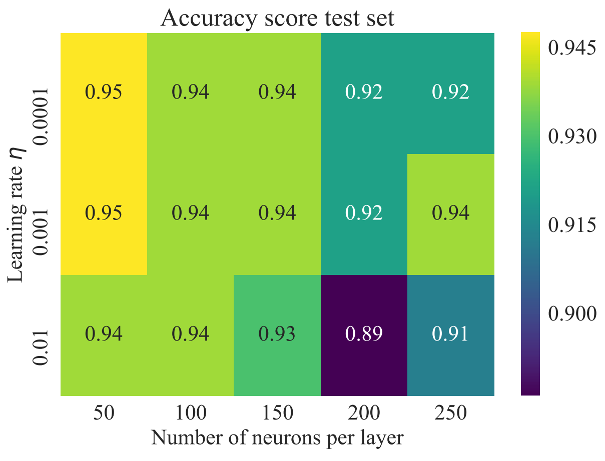


(c) Relu

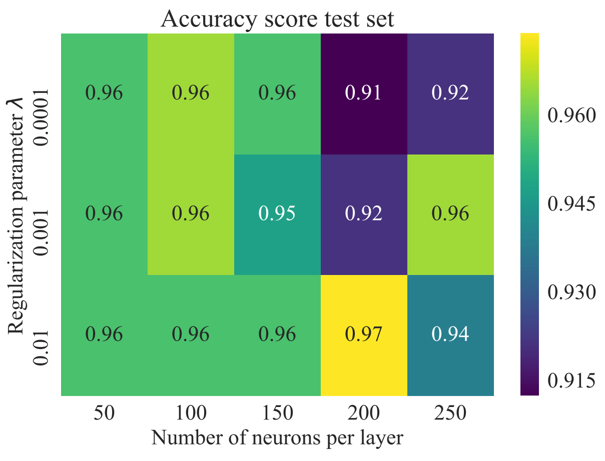


(d) Leaky Relu

Figure 9: Prediction accuracy of neural network with stochastic gradient descent on test set using breast cancer data. The subfigures show the result for different activation functions when varying the regularization parameter λ and learnign rate η. Epochs is set to 50 and batch size is 10. Two hidden layers are used with 15 neurons in each layer.

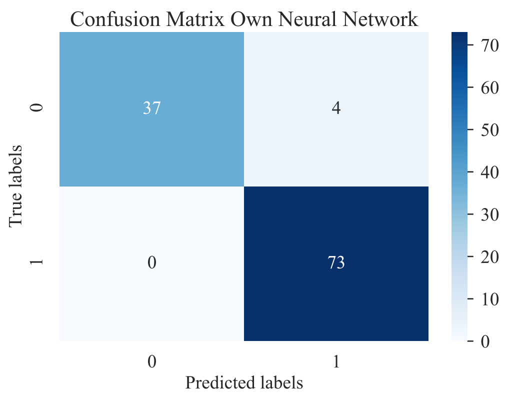


(a) $\lambda = 0.0001$

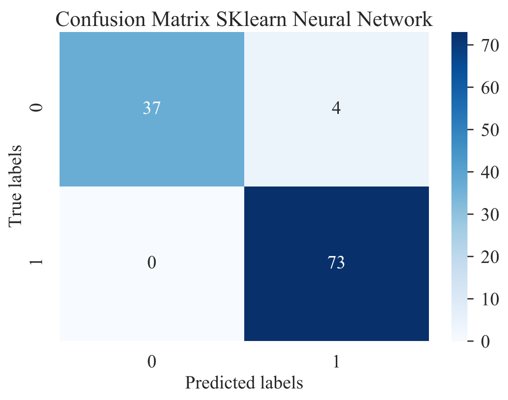


(b) $\eta = 0.01$

Figure 10: Prediction accuracy of test set based on breast cancer data using a neural network with stochastic gradient descent. The sigmoid is used as activation function, batch size is 10, and the number of epochs are 50. There also three hidden layers and the middle layer contains $20, 50, 70, 120, 150$ neurons.

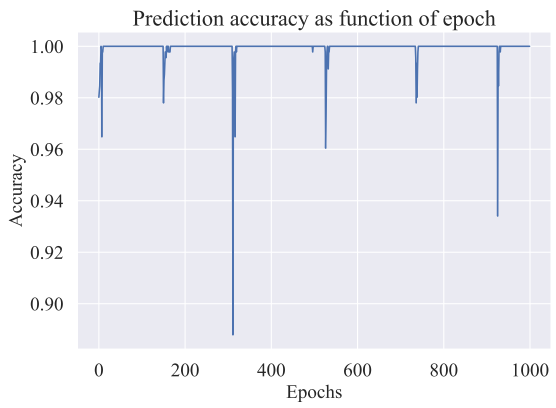


(a) Own implementation

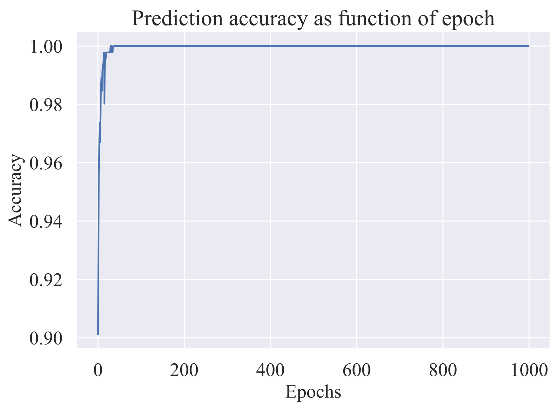


(b) SKLearn implementation

Figure 11: Confusion matrix for the predictions on the breast cancer test data. Predictions were done using both our own and SKlearn's implementation of a neural network with stochastic gradient descent. Batch size is 10, number of epochs is 50, learning rate $\eta = 0.01$, and there are three hidden layers with 50, 20 and 50 neurons respectively. Regularization parameter $\lambda = 0.001$ and the activation function in hidden layers is the sigmoid function.



(a) Learning rate $\eta = 0.1$



(b) Learning rate $\eta = 0.01$

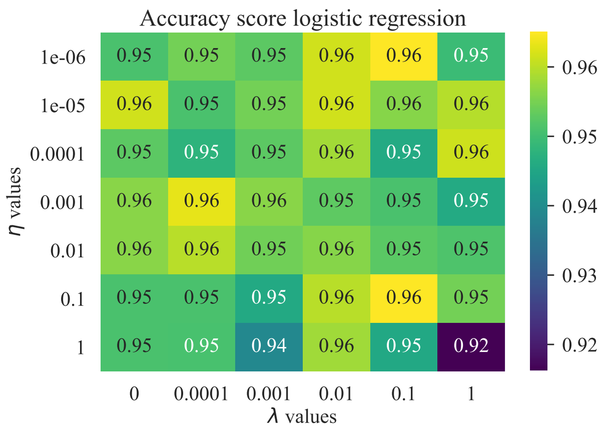
Figure 12: Prediction accuracy using a neural network on breast cancer training data as a function of epochs. The learning rate η is the only difference between the two plots.

3.3 Logistic Regression

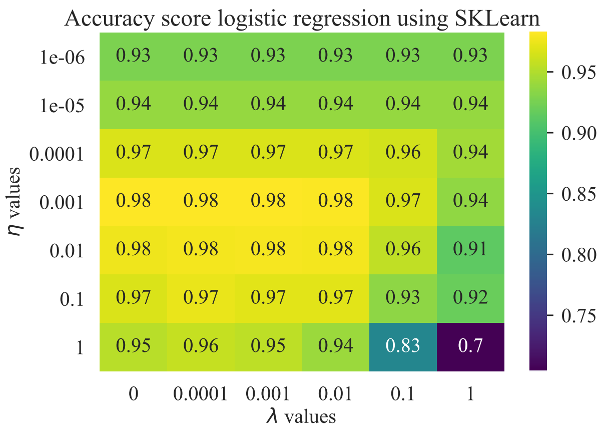
Finally we can move over to the results from logistic regression. First we have our own implementation using stochastic gradient decent. We have used the same breast cancer data as we did in feed-forward neural network. We are using all the features as the design matrix. In figure 13 we can see a heatmap over different values of learning rate η and regularization parameter λ for both our own implementation and the SKLearn implementation. The results have been generated using k-fold cross validation with $k = 5$ folds.

After doing this k-fold cross validation, we used the best results on the test data. This gave us the two confusion matrices in figure 14. Parameters used here are $\lambda = 0.0001$ and $\eta = 0.001$.

Table 2 shows the measurements after testing the best models on the test data. We can see the accuracy, precision, recall, and F1-scores for both implementations. The results here has been inferred from figure 10 and 13.

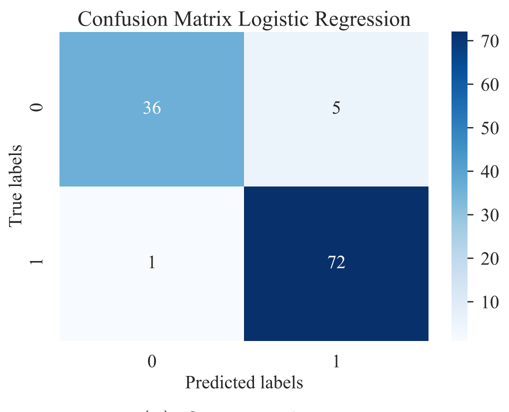


(a) Own implementation



(b) SKLearn implementation

Figure 13: Heatmap of prediction accuracy for different shrinking parameter λ and learning rates η values using own logistic regression (with stochastic gradient descent) and Scikit Learn on breast cancer test data



(a) Own implementation

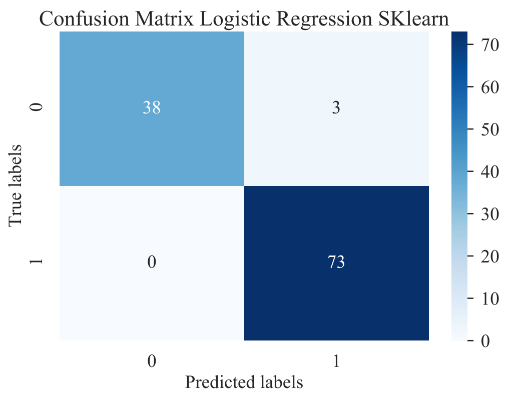


Figure 14: Confusion matrix for the test set with breast cancer data. Predictions are done using own implementation of logistic regression (with stochastic gradient descent) and with SKlearn's implementation. Shrinking parameter $\lambda = 0.0001$ and learning rate $\eta = 0.001$

Table 2: Performance on breast cancer test data using logistic regression and feed-forward neural network (FFNN). Results are the confusion matrices in figure 10 and 13.

	Logistic Own	Logistic SKLearn	FFNN Own	FFNN SKLearn
Accuracy	0.947	0.974	0.965	0.965
Precision	0.935	0.961	0.948	0.960
Recall	0.986	1.00	1.00	0.986
F1 Score	0.960	0.980	0.973	0.973

4 Discussion

4.1 Stochastic Gradient Descent

From the table 1 we see that our own implementation of SGD matches relatively well with the implementation of Scikit Learn. This is an indication that we have implemented the method correctly. When using OLS, our SGD method has a smaller MSE by about 2%. For ridge, Scikit Learn is better by about 6%. For Scikit Learn, there is no difference whether we use OLS or Ridge. However, it should be mentioned that the results were computed without a grid search for the learning rate or the shrinkage parameter, since the purpose was only to see if the implementation was correct. Since the methods are stochastic, the results may vary, and to make accurate statements about the MSE, one would need to collect an average for different runs.

Figure 2 clearly shows that the error becomes smaller the higher the epoch. This makes sense because higher epochs ensure that you converge and you are in a minimum. However, we do not know whether we are in a local or global minimum. Scheduling the learning rate helps to make larger steps at the beginning, which become smaller over time. This is to achieve faster convergence. When the learning schedule is set to true, the MSE for the same number of epochs is significantly lower than for simple stochastic gradient descent. Nevertheless, care must be taken that the learning rate does not become too small too soon such that the algorithm stops. The figure also shows the advantage of using momentum stochastic gradient descent. In this case, the higher the momentum parameter, the faster the convergence. The main advantage with momentum is that it ensures not to get stuck in a local minima. Summarized, best results are obtained for using the learning schedule and momentum stochastic gradient descent with γ close to one.

From the figure 3a, it can be seen that the mean square error using OLS is the lowest with a value of 0.045 for a high number of mini batches ($m = 250$) and a learning rate of $\eta = 0.1$. Note that grid search helped to reduce the error compared to the results in Table 1, where the error was 0.052. That is a difference of 16%. Looking at figure 3b and 3c there are multiple parameter combinations which yield the lowest mean squared error of 0.045. Ridge is therefore not better than OLS. There is a trend that the number of mini batches should be higher than 50 and the learning rate set to 0.1 while lambda is small $\lambda \sim 10^{-6}$. Thus, the optimal learning rate is the same for OLS and ridge.

Of course, an even more comprehensive and computationally intensive grid search could be performed to find the optimal parameters that reduce the error. This is the core problem of machine learning, to find the optimal parameters.

4.2 Regression

Studying figures 4-7, the best results concerning the mean squared error are obtained for the relu activation function in the hidden layer with and MSE of 0.048 where $\eta = 0.001$ and $\lambda = 0.01$. The leaky relu performs worst with a 15% increase in the error rate ($\eta = 0.001, \lambda = 0.001$). For the sigmoid and softmax the error increases by 2% and 6% respectively. The best R2 score of 0.59 is obtained when using the sigmoid or softmax as activation functions where $\eta = 0.01, \lambda = 0.001$. However, the best R2 score for the different activation functions does not vary more than $\sim 5\%$. It is generally evident that an incorrect choice of parameters can have a major impact and leads to varying results regardless of which activation function is used. In general, all runs gave the best results with $\eta = \{0.001, 0.01\}$ and $\lambda = \{0.001, 0.01\}$.

Nevertheless, the lowest mean squared error is about 7% worse than what we got with the stochastic gradient descent method. Additionally, from our previous work (Ref. [7]) we know the mean square error for regular linear regression using a two-dimensional fourth degree polynomial. Using bootstrap with 100 cycles, the mean square error using OLS was 0.044. For ridge regression, the mean squared error was 0.043 with $\lambda = 0$. This is a strong indication that further grid search needs to be done.

The addition of an extra layer when using the sigmoid function did not improve the result, but made it worse (figure 16). This might be due to overfitting. Changing the number of neurons but keeping one hidden layer has a great impact. Since there are many results, the heat maps are included in the appendix (figure 15). Again, the lowest error is obtained for the relu activation function with $MSE = 0.039$ and $\eta = 0.01, \lambda = 0.0001$. That is, increasing the number of neurons per layer from 10 to 15 (or to 20, which leads to the same result, but it is best to choose the simplest architecture to avoid overfitting) helped while the learning rate stayed the same. Now our neural network is 10% better than the regular linear regression where we set the input to be a two dimensional polynomial of degree four. The advantage of a neural network is that you do not have to specify the form of the data beforehand. However, it is difficult to draw direct conclusions about the optimal parameters and their real meaning. In accordance, the R2 score improved to 0.66 when using the relu activation function.

It is important to note that the results we have obtained here might be far from optimal. The aim is to see how each parameter affects the outcome. This is particularly hard because changing one hyperparameter might change the network such that another hyperparameter has a different effect than previously. For example, when bench marking our best results using relu as activation function with sklearn's multi-layer perceptron regressor, ($\eta = 0.01, \lambda = 0.0001$, for one hidden layer with 15 neurons and a batch size of 50) we get $MSE = 0.057$ and R2$= 0.053$. Changing the learning rate to $\eta = 0.1$, Scikit Learn's result improve to a mean squared error of 0.045 and an R2 score of 0.63. Still, our network is better. This may be due to the many setting options and the fact that the stochastic gradient descent introduces a certain randomness.

4.3 Classification

Now on to a discussion on how the neural network performs when classifying malign or benign tumours based on 30 features. First off, most of the results from the neural network are pretty good. Even though the hyperparameters vary widely, most of the results show a 94%+ prediction accuracy on test data.

The main aim of figure 8 is to see how the number of hidden layers and number of neuron in each layers affects the result for each activation function. If we start with sigmoid as activation function (figure 8a), we see that a higher complexity neural network (in terms of architecture) performs better. The worst performance comes from 1 hidden layer, while the the best comes from 3 hidden layers. However, there is not a big difference on the performance, as the worst result in this subfigure is a 95% prediction accuracy while the best is 97%. The behaviour differs from softmax as activation function (figure 8b), where there it is not as clear what number of layers and neurons yields the best results. It is clear however, that 3 hidden layers may give some inferior results, probably due to overfitting, as the training results still give at least 99% prediction accuracy.

Relu (figure 8c) and leaky relu (figure 8d) show that a too high complexity can result in terrible prediction accuracy. The accuracies of 69% and 36% is probably due to the NN only predicting either only 0 or only 1. All in all, the result's behaviour is quite similar for all activation functions when varying the number of hidden layers and number of neurons in each layer.

As each activation function performed well with two hidden layers and with 15 neurons each, we tested how the neural network performs when using those values for all activation functions, while now varying the learning rate η and regularization parameter λ. When looking at figure 9, it is clear that a high value of regularization parameter $\lambda = 0.01$ does not perform well at all with relu and leaky relu, while sigmoid and softmax do not mind. On the contrary, softmax performs much worse when using a low value, $\lambda = 0.0001$. The sigmoid performs evenly regardless of both λ and η. The role of the learning rate is to ensure that the weights of the neural network converge as quick as possible, while still being stable. Therefore, a high learning rate is preferable as long as the weights are stable. A too high learning rate can cause the weights to jump out of a minimum which reduces the cost, causing a worse performance of the NN.

An example of this is shown in figure 12. It is clear that the lower learning rate in subfigure 12b yields stable weights, while the higher learning rate $\eta = 0.1$ in figure 12a is large enough for the weights to sometime jump out of the minimas. This is quite interesting, as the prediction accuracy can be 100% for over 150 epochs and then suddenly drop, which is one of the reasons careful thought is required when choosing the learning rate.

Figure 10 shows results from a high number of neurons in each of the 3 layers. The results are sometimes as good as $96 - 97\%$, but it is clear that the risk of overfitting is more significant when using a high number of neurons. And as the results never succeed those from a the less complex neural network discussed earlier, there is no point in using a neural network of that high complexity.

When comparing our neural network to SKlearn's, we see similar results (figure 11). For both networks in that particular run, there were 0 false negatives, and 4 false positives. This goes to show that our own implementation of a neural network yields similar results to other implementations.

4.4 Logistic Regression

For the logistic regression we are using the same data as in neural network. And as in neural network, we got pretty good results using logistic regression.

Figure 13 shows the results from a grid search with both our own implementation and the sklearn implementation. As we can see both methods gives us good results, but the sklearn version is slightly better. The aim of this figure is to find the best values to pick for λ and η. For our own implementation the best results was in different areas, but we choose to use $\lambda = 0.0001$ and $\eta = 0.001$ for the test calculations. The same is true for the sklearn implementation. We can also see that there is this clear trend in the sklearn data which we cannot see in our own implementation. This could be because the are not implemented in the exact same way.

Figure 14 gives us a confusion matrix of the results from the test data. We see the same trends here as in the heatmap, sklearn is slightly better then our own. Table 2 gives us a good comparison of the results. We can see that especially recall is excellent for all the different methods. This means that if we care a lot about finding everyone with malign cancer we have pretty good models.

4.5 Pros and Cons of the Different Models

We have throughout this paper looked at multiple different models for doing both regression and classification. We have looked at ordinary least squares and ridge regression using stochastic gradient descent, feed-forward neural network for both regression and classification, and logistic regression for binary classification. All of these models have their own pros and cons.

We can first look at the regression models. The OLS and ridge regression models are quite similar, here the only different is that we have a $l2$ regularization parameter λ in the ridge model to penalize complexity. Both of these models are linear models and relatively easy to understand. The FFNN regression model is on the other hand way more complex. This complexity gives the model more flexibility to understand complex patterns and can fit non-linear data as well. On the other hand, higher complexity can lead to overfitting and is more difficult to understand. It is not always easy to interpret how a FFNN got its answer. In conclusion, stochastic gradient descent doesn't give better results than regular ordinary least squares or ridge regression, but SGD can arrive at a desired model with less computation in cases where there are large amounts of data. A neural network however, can easily obtain lower error than regular regression, so the nature of the problem decides which method is desireable. If we were to choose between regular regression or neural network on the terrain data, a neural network would probably be better, because the terrain is very complex, and using regular regression would be very computationally heavy for such complexities, as we saw in the previous project (Ref. [7]).

In the case where we have a classification problem we have looked at two different models. FFNN for classification and logistic regression. Most of the same argument as in the regression case holds here. The logistic regression is relatively simple and easy to understand, while the FFNN can be very complex. It important to have multiple methods in your repertoire, such that you have a better chance of finding the optimal method for a certain problem. If a simple model from logistic regression gives the desired result, there is not point in training a neural network. However, if simple models can't capture the complexity of the problem, a neural network might help. What model to choose can be difficult, especially when it is to be used in medicine. A high prediction accuracy is of course desirable, but it is harder to trust a model that can't tell us why it predicts the way it does. Despite this, we would choose the neural network, as we with further tuning could perhaps increase the prediction accuracy even more.

5 Conclusion

In stochastic gradient descent, we do not expect to achieve the analytical results. However, we have shown that stochastic gradient descent is a very good alternative, especially when the data set is large. A good grid search is essential. In our case, it reduced the mean square error for predicting values of the Franke function with $\sim 16\%$ to a value of 0.045. For a data set of 500 data points, a higher number of minibatches ($m > 50$) and a small shrinkage parameter ($\lambda \sim 10^{-6}$) proved successful, while the learning rate $\eta = 0.1$. There was no difference in using the square error or ridge cost function for optimization. However, the hyperparameters must be determined anew for each problem and generalizations are difficult. Scaling the learning rate and adding momentum help to further improve the results.

Using the same data set but a neural method to predict the values of the Franke function, the mean squared error was 0.039, which is an improvement of 10% over our previous work using regular linear regression with a two-dimensional polynomial of degree four as input. The optimized parameters were the learning rate $\eta = 0.01$ and the shrinkage parameter $\lambda = 0.0001$. Only one hidden layer with 15 neurons and relu as activation function was used. Once again, it was realized that a good gridsearch is indispensable.

With some small modifications to the neural network, we were able to use it for classification as well. With the Wisconsin breast cancer data set, we trained the neural network to correctly predict if a tumour is malign or benign 97% of the time. Lots of different choices of hyperparameters yield a 97% prediction accuracy. The least complex network to obtain that result was a neural network with two hidden layers of 5 neurons each. The activation function could be either relu or leaky relu, and the other hyperparameters were regularization parameter $\lambda = 0.001$, learning rate $\eta = 0.01$ and batch size 10. Since there is randomness involved in splitting data and initializing the weights of the network, these results could of course vary. There is little doubt that with further tuning and testing, a 99% accuracy on test data is obtainable.

Compared to the neural network methods, logistic regression does at least an equally good job. The best model on our test data was in fact sklearn logistic regression, giving us 98% prediction accuracy. Hence, for this dataset one might prefer to use logistic regression since it is (i) more understandable and (ii) needs less computational power.

As we have seen in this project, there are huge amounts of tuning that can be done to further improve the results from the neural networks. Also, testing the neural network on other data sets would be interesting. For instance, using the neural network on the terrain data which we used regression on in the previous project (Ref. [7]), might give very good results. And instead of a binary problem, testing the neural network on a classification problem with more classes could and should be done in the future. Additionally, in order to reduce the amount of data, a principal component analysis could be done in order to select the dominant and thus important features.

6 Appendix

6.1 Method

6.1.1 Two-dimensional Polynomial

We define a two dimensional polynomial of degree d to be

$$f(x, y) = \sum_{i=0}^{d} \sum_{j=0}^{i} \beta_{i-j,j} x^{i-j} y^{j} \tag{54}$$

where β are coefficients.
In matrix notation we have

$$f(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\beta} \tag{55}$$

where $\mathbf{x} = [1, x, y, x^2, xy, y^2, \cdots, x^d, x^{d-1}, \cdots, y^d]$ and $\boldsymbol{\beta} = [\beta_1, \beta_2, \cdots, \beta_p]^T$. p denotes the number of features of the model

$$p = \frac{(d+1)(d+2)}{2} \tag{56}$$

6.1.2 Linear Regression

The equation for the approximated $\tilde{y}$-values is

$$\tilde{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{\beta} \tag{57}$$

where $\boldsymbol{X}$ is the design matrix. $\boldsymbol{\beta}$ are the parameters to be determined.

Ordinary Least squares In view of equation (2), the derivative of the cost function is defined as

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \boldsymbol{X}^T (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) \tag{58}$$

it can be shown that the optimal parameters $\hat{\beta}$ is

$$\hat{\boldsymbol{\beta}}_{OLS} = \left(\boldsymbol{X}^T \boldsymbol{X}\right)^{-1} \boldsymbol{X}^T \boldsymbol{y} \tag{59}$$

Ridge Regression For ridge regression the optimal parameter can be shown to be

$$\hat{\boldsymbol{\beta}}_{\text{Ridge}} = \left(\boldsymbol{X}^T \boldsymbol{X} + \lambda \boldsymbol{I}\right)^{-1} \boldsymbol{X}^T \boldsymbol{y} \tag{60}$$

Evaluation Methods can be evaluated by looking at the Mean Squared Error (MSE) and the coefficient of determination (R^2). A definition of these values can be found in equation (61) and (62).

$$MSE(y, \tilde{y}) = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2 \tag{61}$$

$$R^2 = 1 - \frac{\sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2}{\sum_{i=0}^{n-1} (y_i - \bar{y})^2} \tag{62}$$

where the mean values are defined as

$$\bar{y} = \frac{1}{n} \sum_{i=0}^{n-1} y_i$$

6.1.3 Franke Function

The Franke function, which is a weighted sum of four exponentials reads as follows

$$f(x, y) = \frac{3}{4} \exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + \frac{3}{4} \exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \tag{63}$$

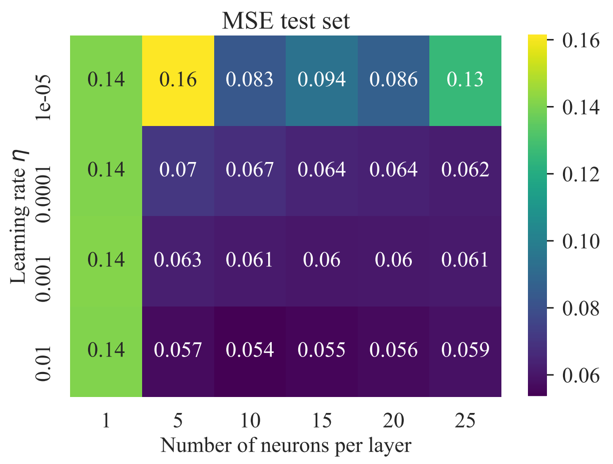
$$+ \frac{1}{2} \exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) - \frac{1}{5} \exp\left(-(9x-4)^2 - (9y-7)^2\right). \tag{64}$$

The function will be defined for $x, y \in [0, 1]$.

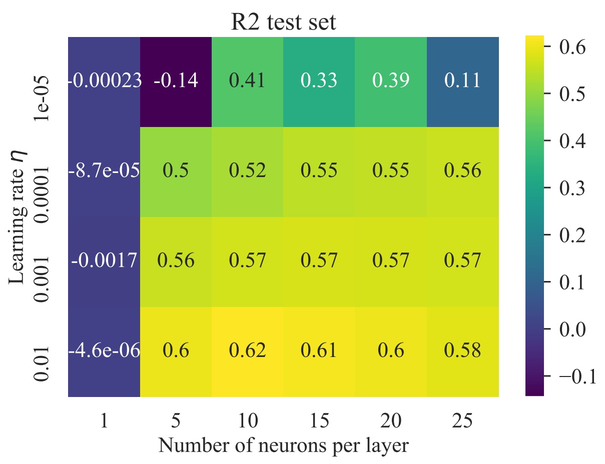
6.2 Results

6.2.1 Regression for Neural Network

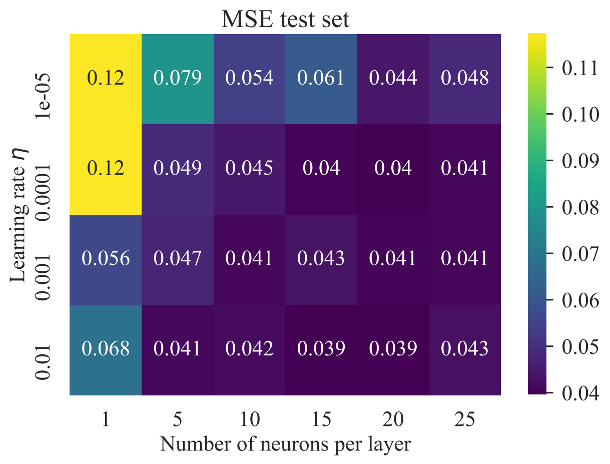
The following figures show additional results for predicting values of the Franke function (see section 3.2.1).



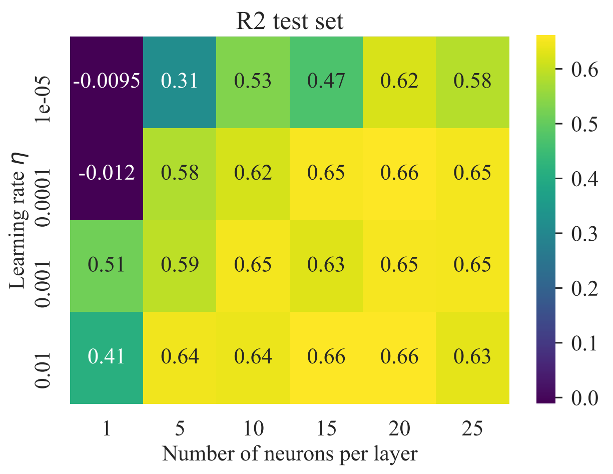
(a) Softmax



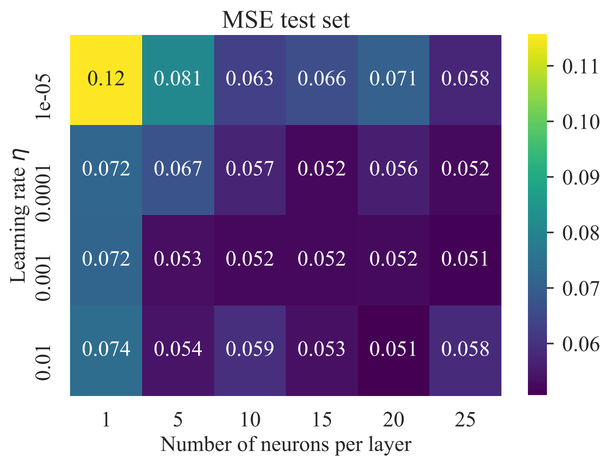
(b) Softmax



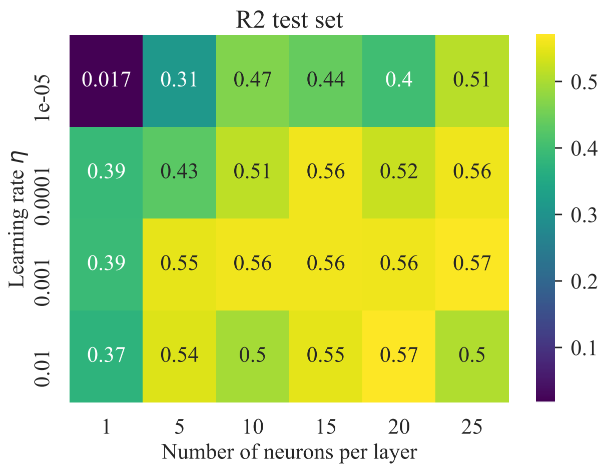
(c) Relu



(d) Relu

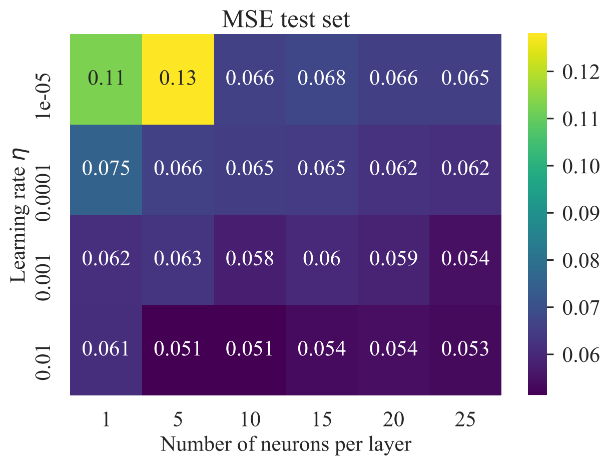


(e) Leaky Relu

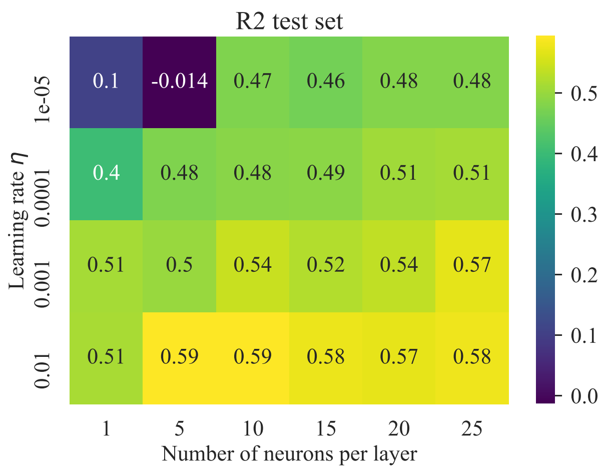


(f) Leaky Relu

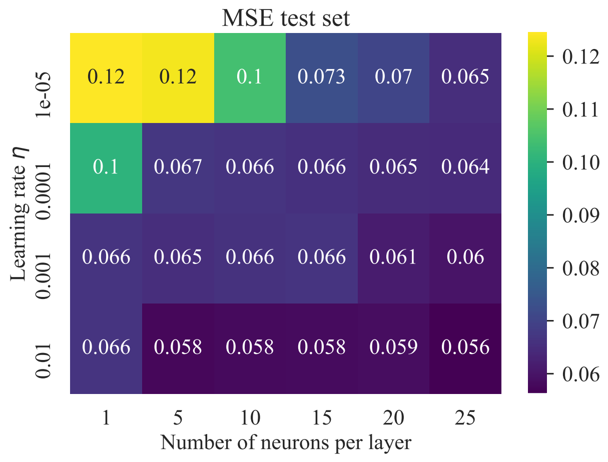
Figure 15: Heatmaps with MSE and R2 for predicting values of the Franke function as function as functions of learning rate and neurons per hidden layer for different activation functions using a feed-forward neural network with stochastic gradient descent. There is one hidden layer for all of the heatmaps. Number of epochs is set to 5000, $\lambda = 0.0001$ and batch size is 50.



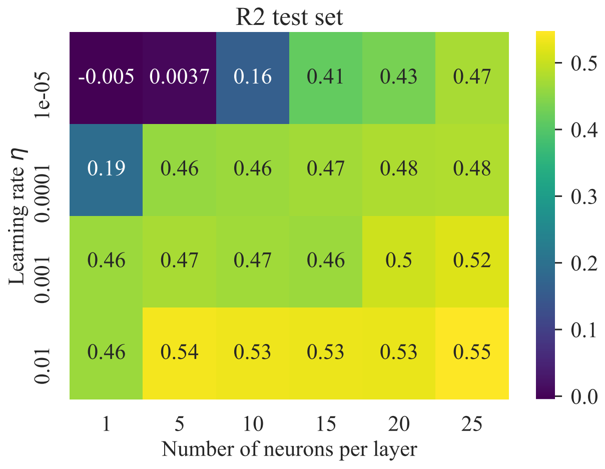
(a) One hidden layer



(b) One hidden layer



(c) Two hidden layers



(d) Two hidden layers

Figure 16: Heatmaps with MSE and R2 for predicting values of the Franke function as functions of learning rate and neurons per hidden layer using a forward neural network with stochastic gradient descent. Number of epochs is set to 5000, $\lambda = 0.0001$ and batch size is 50 and the activation function in the hidden layer is the sigmoid.

References

[1] Aleksey Bilogur. Ridge regression cost function, 2017. https://www.kaggle.com/residentmario/ridge-regression-cost-function.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] Morten Hjorth-Jensen. Data analysis and machine learning: Logistic regression, October 2021. https://compphysics.github.io/MachineLearning/doc/pub/week38/html/week38.html.

[4] Morten Hjorth-Jensen. Week 40: From stochastic gradient descent to neural networks, November 2021. https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40.html.

[5] Morten Hjorth-Jensen. Week 41 constructing a neural network code, tensor flow and start convolutional neural networks, October 2021. https://compphysics.github.io/MachineLearning/doc/pub/week41/html/week41.html.

[6] Lisa Meeden. Derivation of backpropagation, 2010. https://www.cs.swarthmore.edu/~meeden/cs81/s10/BackPropDeriv.pdf.

[7] Bendik Nyheim, Marcus Moen, and Mira Mors. Regression analysis and resampling methods, using ordinary least squares, ridge regression and lasso regression on the franke function and real terrain data, September 2021. https://github.com/benyhh/FYS-STK4155/tree/main/Project1/Report.