FYS-STK4155 - Project 2

Kvalvaag, Tom-R. T. Odorczuk, Michał Jan Ho, William

November 20, 2021

Abstract

As an extension of project 1, we explored different methods for solving linear regression on the Franke Function. We developed and employed both OLS and ridge using SGD, and also a feed forward neural network for the regression task. The results of the gradient methods were not quite up to par compared to the OLS and Ridge from project 1 which utilized matrix inversion. Furthermore, we also explored the problem of classification on the Wisconsin Breast Cancer dataset using two different methods. The first of these methods was a modification of the aforementioned feed forward neural network and the second being logistic regression, both of which yielded exceptionally good results. This project showed that choosing the most complex model does not automatically yield the best results, and that the quality of the data is just as important as the model itself.

The source code for this project can be found at Github[1]. For generations of results in this report, run the scripts located in the solutions folder in the repository.

1 Introduction

Machine learning is a field which is in constant development, and more complex methods are being developed every year to try to solve problems in a more efficient way. However, more complex models does not necessarily lead to better performance. Most of the times, striving for good data quality is much more important than having a complex model when it comes to producing great results. In addition, complex models are often more computationally heavy and also less interpretable.

We will in this project explore different methods to solve the problem of regression and classification. For the problem of regression, we will expand upon project 1 [1] and apply and compare new methods to solve the problem of estimating the Franke Function. These methods include linear regression using gradient optimization, and also more complex neural networks. For classification, we will tackle the Wisconsin Breast Cancer dataset. This is a heavily studied dataset, and contains a lot of great features which are very valuable when it comes to separating the samples into two different classes. We will adapt our neural network used for regression to the problem of classification, and employ the modified neural network on the Wisconsin Breast Cancer dataset. Furthermore, we will also look at a simpler model for classification, namely logistic regression, and see how it stacks up against the more complex neural networks.

This report mainly consists of three sections. Firstly, we will presents the theory behind the methods and expound upon how the methods are applied in this project. In the second section we will present our results of the experiments conducted. Afterwards, we will discuss the results, and finally end the report with a conclusion of our work.

2 Method

2.1 Problems and Datasets

As regression and classification are two separate problems, we will provide different datasets for each of the two.

2.1.1 Regression - Franke Function

One of the main goals of this project is to explore different machine learning methods for regression. Regression tackles the problem of fitting a continuous function based on sets of input and output data. In our case, we have chosen to reuse the widely used The Franke Function discussed in project 1 [1].

[1] `https://github.com/MJOdorczuk/MachineLearning/tree/master/Project2`, visited 12.11.2021

The Franke Function is defined for $x, y \in [0, 1]$ and is a weighted sum of exponentials

$$f(x,y) = \frac{3}{4}\exp\left\{\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right)\right\} + \frac{3}{4}\exp\left\{\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right)\right\}$$
$$+ \frac{1}{2}\exp\left\{\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right)\right\} - \frac{1}{5}\exp\left\{\left(-(9x-4)^2 - (9y-7)^2\right)\right\} \quad (1)$$

and the surface form of the function can be seen in Figure 1.

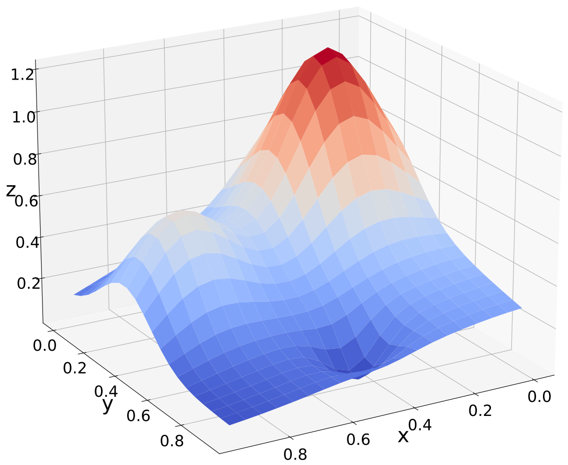


Figure 1: Visualization of Franke Function

To setup our input data, we sampled a set of points x and y from a uniform distribution. To generate our target values, we used a straight forward implementation of the Franke Function, and applied the function on our set of x and y to produce targets z. The generated data is then normalized by subtracting the sample mean value (standard scaler) as in Equation (2). Typically, the standard scaler includes dividing by the standard deviation of the samples, but due to the data already being uniformly distributed this would result in dividing by a number close to zero, resulting in larger values. Therefore, we decided to not include scaling by the variance.

$$x_j^{(i)} = x_j^{(i)} - \bar{x_j} \quad (2)$$

We will try to approximate this function in two different ways. Firstly, we will apply linear regression methods. We will compare our results of applying the ordinary least squares (OLS) and Ridge regression methods using matrix inversion from project 1 with the same methods, but this time, replacing the matrix inversion with a gradient based optimization algorithm in Stochastic Gradient Descent. Here we use a polynomial design matrix from the x and y coordinates as input to the model.

Secondly, we will try to approximate the function by apply a Feed Forward Neural Network (FFNN) using back propagation and stochastic gradient descent to optimize our regression model. As input to the FFNN we will not use the same input as for linear regression as discussed above, but a simple coordinate pair input $= [x, y]$.

More details of the architectures follows, except the matrix inversion regression methods, as they were discussed in project 1[1].

As this is a generated dataset, we will stick to only 100 data points to show how well our models will perform on a limited dataset, which we are more likely to encounter in real life compared to a large dataset.

2.1.2 Classification - Wisconsin Breast Cancer

The other main goal of this project is to explore the problem of classification using machine learning. The classification problem aims to classify a data sample into one of several predefined categories, also known as classes. To do this, we need to approximate a function which outputs the probability of a sample being in a given class, and then assign the sample to the class with the highest probability. In this project, we will study the Breast Cancer Wisconsin (Diagnostic) data set, which consists of 569 data samples. Each of the samples consists of 30 different numerical features which are computed from a digitized image of a breast mass, and a label of one of two classes 0 and 1, which represents malignant and benign samples respectively.

The data is easily accessible through scikit-learn. The input data is setup as a matrix of dimensions $[569, 30]$, where each row represents a single sample and the columns are the features. The targets is a one dimensional vector of size 569, where each of the entries corresponds to the respective sample in the input data. As we have 30 different features, which is quite a lot, we will scale our data using the standard scaler

$$x_j^{(i)} = \frac{x_j^{(i)} - \bar{x_j}}{\sigma(x_j)} \tag{3}$$

We will only scale the input data, as the the output labels are discrete values 0 or 1, thus scaling the outputs will ruin the discretization.

We will tackle this classification problem in two ways. Firstly, we will modify the neural network used for our regression problem by changing the output layer, and also changing the optimization criterion.

Secondly, we will implement another classification algorithm in logistic regression, and optimize it using gradient descent.

More details of the architectures follows.

2.1.3 Splitting our data

For the task of comparing linear regression using matrix inversion and SGD, we will be splitting our dataset into only training and testing consisting of 80% and 20% of our data as we did in project 1 [1] to match the experiments performed there.

However, for the rest of the tasks, we will be splitting our data further into training, evaluation and testing sets consisting of 60%, 20% and 20% of the total dataset respectively. The training dataset will be used for tuning the model parameters through gradient descent discussed in Section 2.2, whilst the evaluation set will be used to tune hyperparameters which will be discussed in Section 2.8. The test set will be "locked in a vault" during training, and only be used a single time during the final testing when the model is done training. This is to prevent us from overfitting to the testing dataset, which would give us a biased evaluation of the generalization of our model.

2.2 Stochastic Gradient Descent

Theory

A major part of machine learning methods is attempting to minimize a cost function, or optimize a criterion in general. For some methods, such as OLS, this can be done analytically as we discussed in project 1. However, for many applications such as neural networks, this is not the case, and a numerical method is needed to optimize our criterion.

Stochastic gradient descent (SGD) is a numerical method that has become the staple method for optimizing machine learning methods [2]. This method aims to find a local minimum of our cost function iteratively.

Given the cost function $C(\beta)$ and its gradient $\nabla C(\beta)$. SGD can be defined as the iterative process

$$\beta_{j+1} = \beta_j - \eta \nabla C(\beta_j) \tag{4}$$

where η is the learning rate or step size in the direction of the gradient, and β is our model parameters. The algorithm for computing SGD can be described as in Algorithm 1. A key point of SGD, is the use of mini-batches instead of the whole training set in each iteration. This introduces a stochasticity in our search for a local minimum, which in turn may help us jump out of a bad local minimum, as the search landscape may change a little with each mini-batch.

A variant of SGD is with an added momentum, where a momentum/inertia m term is introduced. This helps the method keep a history of direction and a regulation variable α for how much is remembered. As can be seen in Equation (6), if α is 0 it reverts back to a standard SGD.

$$m_{j+1} = \alpha\ m_j + (1-\alpha)\nabla C(\beta) \tag{5}$$
$$\beta_{j+1} = \beta_j - \eta\ m_{j+1} \tag{6}$$

Algorithm 1 Stochastic gradient descent

```
Ensure: $E$ = Number of epochs, $B$ = number of mini batches
  Randomly assign values to $\beta$
  for $epoch = 0..E$ do
      for $j = 0..B$ do
          Randomly sample training data
          $\beta_{j+1} = \beta_j - \eta\nabla C(\beta)$
      end for
  end for
```

Implementation

An implementation of SGD using mini-batches for the regression problems from project 1 was implemented to replace the analytical minimization of the parameters and also for the optimization of the neural network and logistic regression. It includes the option to use SGD with momentum. The SGD use a time based learning rate decay, where the learning rate η for a given step n and a decay parameter d is given by Equation (7).

$$\eta_{n+1} = \frac{\eta_n}{1 + d \cdot n} \tag{7}$$

For the neural network and logistic regression, we implemented a SGD class such that each set of weights would keep their own instance of the optimizer. This allowed each layer of the neural network to have their own momentum term independent of other layers.

We also implemented a simple early stopping mechanism to stop the training when it stopped improving. We simply stopped the training when the change in loss was smaller than a small value $\epsilon << 0.01$.

In our code, we have computed all the gradients with the help of the Python package, autograd [3].

2.3 Linear regression

As we have looked at extensively in project 1 [1], matrix inversion is one way of solving the problem of linear regression with both OLS and ridge regression. Another way to solve the same problem is through the use of gradient methods such as SGD to minimize our criterion. We will in this project, again, employ both OLS and ridge regression as we did in project 1, but this time replace the matrix inversion with the SGD algorithm described in Algorithm 1 to minimize the mean squared error.

To compare the method using matrix inversion and the gradient optimization, we ran a grid search to find the best hyperparameters for each complexity for each of the models, after which the best results for each of the four models were plotted as a function of complexity.

2.4 Feed Forward Neural Network

Today, Neural networks have taken a prevalent role in the the field of machine learning, and even spawned a field of its own, namely deep learning. Neural networks can be used in a plethora of different applications, and in our case, we will utilize it for both regression and classification.

Theory

A feed forward neural network (FFNN) is a directional acyclic graph which tries to approximate a function f^*. [2, p. 164]. As the graph is directional and acyclic, all inputs of the network may only go in one direction, hence the name feed forward. This property of a FFNN is important to keep in mind for our project, as other more complex architectures such as recurrent neural networks allows feeding data back to itself. Thus, to keep it simple, we will only focus on a FFNN. A simple FFNN architecture usually consists of at least three main component also called layers. These layers are the input, hidden and output layers. The number of neurons in each of the layers may vary, and additional hidden layers may also be added to make the network "deeper".

Each layer has a set of trainable parameters that we can adjust through the back propagation algorithm. These parameters consists of a weight matrix w and a bias vector b. These parameters are used to produce the output of a given layer through a process called the forward pass. Each neuron in the layer processes the input x by an inner product of x and the weights w_{jk} corresponding to the j^{th} input neuron and k^{th} output in the layer. Then, we add the bias to the inner product, and finally apply an activation function $f(\cdot)$ to produce the output y_j of the j^{th} neuron. This whole process for a single neuron can be written as

$$y_j = f(\sum_{i=0}^{n-1} w_{j,i}x_i + b_j) \tag{8}$$

and is done for each of the j neurons in the given layer. Thus, the number of inputs and number of outputs of a given layer is flexible and determined by how we define the shape of our weight matrix w and bias vector b. Furthermore, the output of a layer is given by a vector of all the $y_j's$ and fed as the input for the next layer. This process starts with the input layer which takes the input x, and feeds it through the layers until it reaches the output layer which produces the final output of the FFNN.

Looking at the FFNN more generally, a FFNN is nothing more than a composition of a set of nested functions. For a FFNN with l layers and N_0 inputs, the whole forward pass for a single output neuron y_k can be written as

$$y_k = f^l(\sum_{i=0}^{N_l} w_{k,i} f^{l-1}(\sum_{j=0}^{N_{l-1}} w_{i,j}^{l-1}(\dots f^1(\sum_{n=0}^{N_0} w_{m,n}^1 x_n + b_m^1)\dots) + b_i^{l-1}) + b_k^l) \tag{9}$$

where $k \in \{0, K-1\}$ for K output neurons. The activation functions f^l may vary, and will be discussed in section 2.4.2.

2.4.1 Implementation of Neural Network

To represent the FFNN, we first implemented a `Layer` class to represent layers. Each layer object contains an activation function, a SGD optimizer, and a weight matrix and a bias vector which again implicitly represents the neurons through the shapes of the parameters as explained above. Furthermore, to store the layer objects and nest them together, we implemented a `NeuralNetwork` class containing a list of layer objects.

The forward pass consists of iterating through the list of layers, and calculating Equation (8) and passing the output to the next layer as input.

For derivation of cost and activation functions during back propagation we use a package for performing automatic differentiation [3]. This makes the network more flexible with regard to the functions to be used.

2.4.2 Activation functions

Without an activation function, the forward pass Equation (8) solely consists of a linear function of the inputs. Thus, to be able to learn a non-linear function, we have to introduce a non-linear activation function in each layer. There are many possibilities for non-linear activation functions, but a requirement is that it should be differentiable, as we are optimizing our FFNN using gradient methods.

Although there may be many different activation functions, we will only focus on a three, namely the Sigmoid, ReLU and Leaky ReLU.

The Sigmoid function is defined as

$$\sigma(y) = \frac{1}{(1 + e^{-y})} \tag{10}$$

and maps the input to $[0, 1]$, which is a useful property for the classification problem as we will discuss later. The derivative of the sigmoid is given by

$$\sigma'(y) = \sigma(y)(1 - \sigma(y)) \tag{11}$$

A downfall of this function, is derivative approaching 0 when $y \to \pm\infty$. This leads to the vanishing gradient problem and can make learning using the sigmoid function a challenge. Nevertheless, it still serves as a good baseline for introducing non-linearity to our network.

The ReLU function is defied as:

$$f(y) = max(0, y) \tag{12}$$

with a derivative

$$f'(y) = \begin{cases} 1 & \text{if y >0} \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

Unlike the sigmoid, the derivative of the ReLU only tends towards 0 if the input is <0. This counteracts the vanishing gradient problem, as long as about half the weights and values are >0. As will be discussed later, we have in this project initialized our weights using a normal distribution with 0 mean, thus about half the weights will be non-negative, meaning only half of our neurons will die out and will not completely killing all our gradients.

The Leaky ReLU extends upon the ReLU and addressed the fact that ReLU will kill quite a few of the neurons. It is defined as

$$f(y) = max(\alpha y, y) \tag{14}$$

where $\alpha \sim [10^{-2}, 10^{-3}]$. The derivative is then given by

$$f'(y) = \begin{cases} 1 & \text{if y >0} \\ \alpha & \text{otherwise} \end{cases} \tag{15}$$

Since the gradient is no longer 0, the network is less likely to die.

2.4.3 Back propagation

To optimize our FFNN, we will need to adjust the weights of the network. As we have seen, a way to do so is to use SGD. However, a challenge of applying SGD in a neural network is that the network has multiple sets of weights. Thus, it is hard to find out which component of the network contributed to the error and how much they contribute, which in turn makes it hard to calculate the gradient needed for updating the weights.

The back propagation (backprop) algorithm addresses this problem by utilizing the chain rule of calculus to propagate the loss of our outputs back to our weights. Through the chain rule, backprop computes the gradient of the loss with respect to each of the weights of our network, and does this one layer at a time moving in the opposite direction of the input of our network. These gradients are then combined with SGD to update our weights.

The derivation of error terms used in the algorithm can be found in Appendix A.1 and we will only show the main points of the algorithm here. Furthermore, as we will be using several different activation functions in this project, we will keep the algorithm general. However, to keep things a bit simpler, we will also be using the halved square error as our cost function

$$C(\hat{W}) = 1/2 \sum_{i=0}^{N-1} (y_i - t_i)^2 \tag{16}$$

but the same principles also applies for other cost functions such as the normal MSE and binary cross entropy.

The Algorithm

Since the algorithm requires all the inputs to each layer computed by Equation (35), including the inputs from the input layer x, and also the inputs to the activation functions computed by Equation (36), we will have to store them for each layer during the forward pass. After completing the forward pass, we compute the error for the output layer, which is given by

$$\delta_j^L = f'(z^L) \frac{\partial C}{\partial (a_j^L)} \tag{17}$$

where L is the last layer, j is the index of a given output neuron, a^L is the output neurons, C is our cost function, and $f'(\cdot)$ is the derivative of the activation function.

Then, we iterate all the layers backwards for each of the hidden layers until the input, and compute the error given by

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l) \tag{18}$$

where δ_k^{l+1} is the error of the previous from the side of the output, w_{kj}^{l+1} is the weights of said previous layer, and $f'(z_j^l)$ is the derivative of the current layers activation function applied on the input of the layer.

Now that we have our error terms, we can finally use them to calculate our gradients which can be used to update our weights with SGD. As shown in Equation (44), the gradient of our cost function C with respect to our weights is then given by our error term multiplied by the activation of the previous layer, which is also the input to the current layer. Thus, for each layer $l = L-1, L-2, .., 2$, we can update our weights by

$$w_{jk}^l = w_{jk}^l - \eta \frac{\partial C}{\partial w_{jk}^L} = w_{jk}^l - \eta \delta_j^l a_k^{l-1} \tag{19}$$

where η is our learning rate, and from Equation (50), we can update our bias by

$$b_j^l = b_j^l - \eta \frac{\partial C}{\partial b_j^l} = b_j^l - \eta \delta_j^l \tag{20}$$

Also, note that for the first layer, our a_j^{l-1} is the input of the network.

2.5 Cost functions

The choice of a cost function for our machine learning model is one of the most central parts of designing a machine learning system, as the cost function plays a major role in defining if the system is a regression, classification or another type of model.

For regression the cost function we will be using is the mean squared error which we studied extensively in project 1[1]. For a prediction from the model $\hat{y} = f(x)$ with a target value of y and m samples, te MSE is defined as

$$MSE = \frac{1}{m} \sum_{i=0}^{m} (\hat{y} - y)^2 \tag{21}$$

The gradient of the MSE with respect to the model weights w is defined as

$$\frac{\partial MSE}{\partial w} = \frac{2}{m} \sum_{i=0}^{m} (\hat{y} - y) \frac{\partial \hat{y}}{\partial w} \tag{22}$$

as our model $\hat{y} = f(x)$ is a general model.

For classification we will be using the binary cross entropy (BCE) cost. Given the output of a binary classifier $\hat{y} = f(x) \in [0, 1]$ and a target y, the BCE is defined as

$$BCE = -\frac{1}{m} \sum_{i=0}^{m-1} y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i) \tag{23}$$

As we can see, depending on the target value, which are either 0 or 1 in the case of classification, one of the two terms in the sum will disappear. Thus, the BCE cost of a given sample (x_i, y_i) is the negative logarithm of the predicted probability of the true class.

Furthermore, for a linear model with the sigmoid activation $\hat{y} = \sigma(wx + b)$, the gradient of the BCE with respect to the weights is

$$\frac{\partial BCE}{\partial w} = \frac{1}{m} \sum_{i=0}^{m-1} x(\hat{y}_i - y_i) \tag{24}$$

and the gradient with respect to the bias b is

$$\frac{\partial BCE}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (\hat{y}_i - y_i) \tag{25}$$

For each of the losses, we will add $L2$ regularization term of the model weights

$$\lambda \sum_{i=0}^{n-1} w_i^2 \tag{26}$$

For the linear regression and logistic regression models, this term is explicitly added to the loss function, whilst for the neural network, the regularization is baked into the back propagation algorithm.

2.6 Classification

As mentioned in Section 2.1.2, we will also be tackling the task of classification in this project by analyzing the Wisconsin Breast Cancer dataset. To do this, we will first adjust our FFNN from handling regression to classification, and then we will compare it with another classification algorithm, namely logistic regression.

Since our dataset consists of only two classes, we can treat the problem as a binary classification task. In shorts, we will first approximate a continuous function $f(x)$ much like we did in the regression problem using stochastic gradient descent. However, after we have produced an output from the approximated function, we turn it into a probability using the logistic sigmoid function $\sigma(z)$ in Equation (10), which maps a continuous value $z \in \mathrm{R}$ to $[0, 1]$, which can be interpreted as the probability of sample x coming from class 1.

$$z = f(x) \tag{27}$$

$$\hat{y} = \sigma(z) \tag{28}$$

We can then classify $\hat{y}$ into one of two classes by using a step function.

$$f(\hat{y}) = \begin{cases} 1 & \text{if y >0.5} \\ 0 & \text{otherwise} \end{cases} \tag{29}$$

which yields the final classification.

2.6.1 Training a classifier

From a training perspective, the only things that distinguishes a regression model with a single output, and a binary classifier are the loss functions used and the output activation function. As in our case, the regression model will try to minimize the MSE Equation (21), whilst a binary classifier will try to minimize the BCE Equation (23). Furthermore, as a regression model can predict any number, the output activation is simply an identity function, whilst the output activation for the binary classification is the sigmoid, as described just above.

Thus, to change our FFNN from a regression model to a classifier, we simply change the output layers activation function to a sigmoid, and change the cost function to the BCE.

2.6.2 The accuracy score

To evaluate our classifier, we will use the accuracy score which, for a binary classifier, is defined as

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(y_i = \hat{y}_i)}{n} \tag{30}$$

where I is the indicator function, meaning that if $y_i = \hat{y}_i$, then I yields 1, and I yields 0 otherwise, y is our target classes, and $\hat{y}_i$ is our predicted class. The accuracy score simply indicates how many percent of our total number of samples we predicted correctly, where the value 1 indicates a perfect prediction, and 0 indicates getting every sample wrong.

2.7 Logistic Regression

Although logistic regression sounds like a regression model, it is actually a binary classifier. Logistic regression is nothing more than a neural network with a single neuron in a single layer with a sigmoid activation used to classify our data. We therefore only have a single set of weights and a bias, which we will denote as w and b respectively, which combined also defines our logistic regression model. We will then use SGD to optimize our weights in the same way we did with linear regression in Section 2.3, but this time minimizing the BCE cost function discussed in Section 2.5.

As mentioned in Section 2.1.2, our dataset of interest consists of two classes, meaning our targets will either be $y_i = 0$ or $y_i = 1$. The output of the sigmoid activation can be interpreted as the probability of our sample being in class 1 given our input x and model w, b

$$p(y_i = 1|x_i, w, b) = \frac{1}{1 + exp(-(b + w_i x_i)} \tag{31}$$

Since this is probabilty of class 1, the probabily of class 0 is then given by

$$p(y_i = 0|x_i, w, b) = 1 - p(y_i = 1|x_i, w, b) \tag{32}$$

Since the sigmoid outputs the probability of a sample being in a class, we will thus assign the predicted class based on the class with the highest probability. To do this, we will again utilize the step function Equation (29) which yields the final predicted class which we can use to calculate the accuracy score Equation (30).

Since we will attempt to minimize the BCE cost by updating our pair of parameters w and b, we will perform Algorithm 1 with the gradients given in Equation (24) and Equation (25).

2.8 Hyperparameter tuning

A hyperparameter is a setting that controls the methods behavior which we can customize externally. As an example, for OLS the order of the polynomial in the design matrix is a hyperparameter. A way to select hyperparameters is to perform a grid search over a set of candidates to see which combination of hyperparameters performs best. This is done with a separate validation dataset, that the training of the method does not use [2, p. 118-119]. An important hyperparameter is the learning rate of the optimization method used. If you only have time to tune one hyperparameter Goodfellow *et al.* [2, p. 424] recommend tuning the learning rate.

For our neural network, we perform a grid search for the best parameters over the learning rate, regularisation term λ, number of hidden layers, number of neurons in each layer and activation functions for the hidden layers. As the logistic regression only has a single neuron, we can stick to only search for the regularization term and learning rate. For both the mentioned methods, we kept the batch size constant at 5 to save time. As for the linear regression, we performed a grid search over the batch size, learning rate, momentum term m, and regularization term. We are, however, only able to do a limited search with in the hyperparameter space with a smaller set of discrete values due to time constraints.

During the grid search, the model and parameters are saved when a new best evaluation score is found. The R^2 score is used as the evaluation score for regression, and the accuracy score is used for classification.

2.9 Initialization of weights

An important step of the training process is initializing the weights of our models, especially when it comes to the neural network with several neurons in the same layer. As we want each neuron to learn different features, we don't want them to have the same weights, as having the same weights means that the output of each neuron will be the same. Thus, the output of a given layer is just a scaled sum of the activation of a single neuron. Thus, we want to avoid symmetry in our initialized weights.

Furthermore, we would also want our weights not to be too small or too large, as our gradients depends on our weights as seen in Equation (18). Thus, having too large weights will lead to exploding gradients, and having too small weights will lead to vanishing gradients.

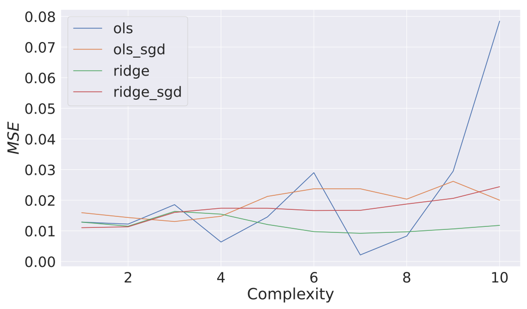
Thus, a good choice for initialization of our weights will be using a standard normal distribution $N(0, 1)$. This ensures that our weights will not have the same values, and also that our weights are in a suitable range for not causing our gradients to vanish or explode. We would also simply initialize our biases to 0, as the biases could be interpreted as the intercept which we would adjust during training. We will initialize the weights for all our models using this method.

3 Results

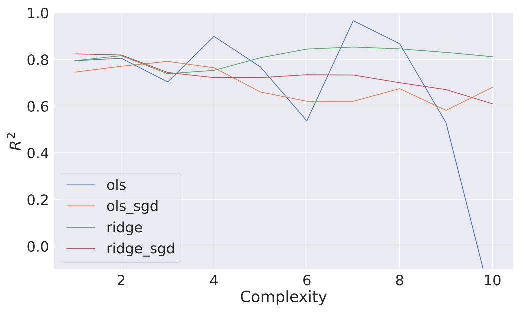
3.1 Regression

3.1.1 Linear regression using SGD

For ordinary least square and ridge regression with both analytical optimization and through the use of SGD, a grid search was performed to find the best hyperparameters. For our models, the hyperparameters were the learning rate, batch size, regularization term λ, use of SGD with or without momentum, and its regularisation variable α. Our final choices of hyperparameters for this search can be found in appendix B. From each of the methods, the best model of each complexity were used to calculate the MSE and R^2 score and plotted as a function of the model complexity. See Figures 2a and 2b. All results for the methods with and without SGD can be seen in Appendix C. Hyperparameters for the best models for each method can be seen in Table 1. Regression with or without SGD do not give a large difference in score or loss. However OLS do perform poorly at a complexity of 10.



(a) MSE as a function of model complexity.



(b) R^2 as a function of model complexity.

Figure 2: Comparison of MSE and R^2 between using matrix inversion and SGD for both ordinary least square and ridge regression as a function of model complexity.

Table 1: Hyperparameters and results for best model for OLS and ridge with SGD from grid search

	OLS	Ridge
Complexity	3	1
Learning rate	0.5	0.25
Lambda	–	0.001
Batch size	1	1
Momentum	True	True
Alpha	–	0.1
Eval MSE	0.0286	0.0357
Eval R^2	0.7616	0.7015
Test MSE	0.0130	0.0110
Test R^2	0.7915	0.8236

3.1.2 Feed Forward Neural Network

The results of the best model's training and validation performance using our FFNN for regression on the Franke Function data can be seen in Figures 3 and 4. The models are the result of a grid search for optimal hyperparameters based on the best R^2 score. The dataset used is generated as discussed in Section 2.1.1 with 100 points of data for x and y, which were split according to section 2.1.3. The hyperparameters and results from the best model is also given in Table 2. The best model was chosen through a grid search, it was tested on an unseen test dataset, where it achieved an MSE of 0.0122, and an R^2-score of 0.836.

3.2 Classification

The goal of our classification task is to predict whether a given data sample from the Wisconsin Brest Cancer dataset is a sample indicating a benign or a malingering case. This is a dataset which has been

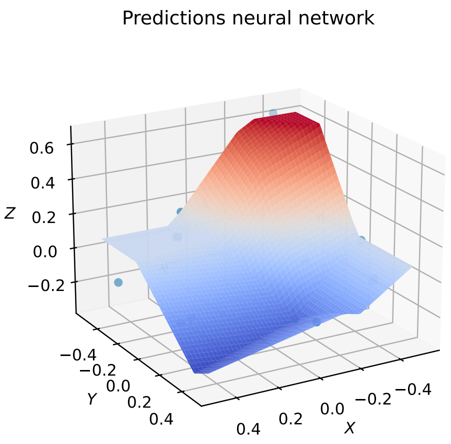
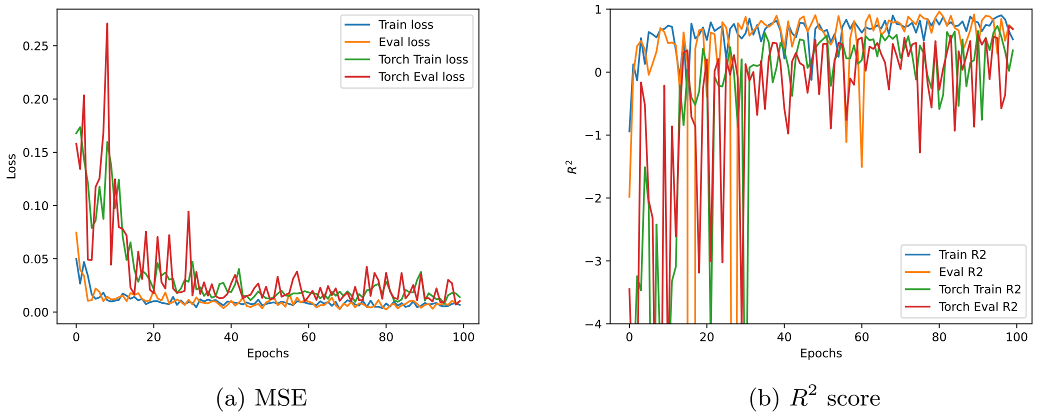


Figure 3: Model visualisation. The surface plot is the prediction from the model and scatter points are test data.



(a) MSE (b) R^2 score

Figure 4: Cost and R^2 score from training and validation of best FFNN for regression

Table 2: Hyperparameters, loss and R^2 for best FFNN regression classifiers

hyperparams	model
LR (η)	0.316
Reg term (λ)	0.001
Hidden layers	3
Nodes in layers	(5,2,4)
Hidden activation	Leaky ReLU
Batch size	5
Eval MSE	0.0023
Eval R^2	0.962
Test MSE	0.0122
Test R^2	0.837

extensively researched, and many others have achieved perfect accuracy on the dataset [4]. With this in mind, we will hope for great results ourselves.

We used all 30 features present in the dataset without doing any form of feature selection or dimensionality reduction. We also scaled our features using the standard scaler as we did in project 1, where we fit the scaler on our training data.

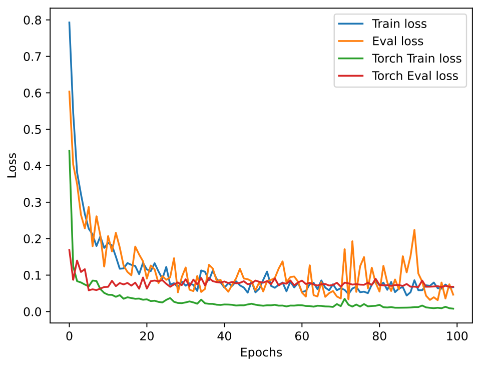
3.2.1 Feed Forward Neural Network

The neural network performed exceptionally well on this task. The accuracy on our validation set would reach 100% with a lot of different configurations. Two of which can be seen in Table 3.

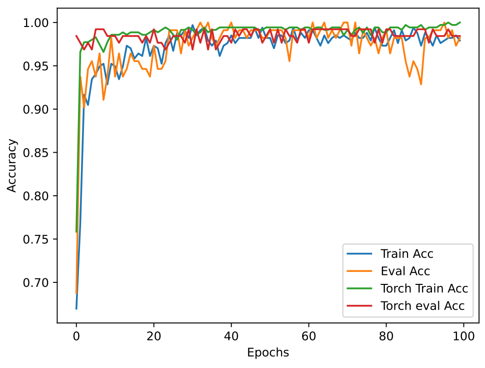
Table 3: hyperparameters for two FFNN classifiers, both with 100% accuracy on evaluation set.

hyperparams	model 1	model 2
LR (η)	1.0	0.316
Reg term (λ)	0.001	0.001
Hidden layers	1	1
Nodes in layers	1	2
Hidden activation	Sigmoid	Sigmoid
Batch size	16	16

However, as we can only test the model on our final test set a single time to prevent overfitting to our test set, we chose to go further with the model with the lowest cost. Of the two presented model, model 1 had a evaluation BCE loss of 0.059, whilst model 2 had 0.056, thus we went with model 2 for the final test. The training and evaluation loss of two models are presented in Figure 5.



(a) Loss



(b) Accuracy

Figure 5: Training and evaluation loss & accuracy for FFNN classification model 2 as in Table 3

As seen in Figure 5, we also compared our model to a neural network built with PyTorch. For this comparison, we took all the hyperparameters we found in our hyperparameter search, and simply transfered them over to the PyTorch model. However, looking at Figure 5 we notice that the learning is not identical. These discrepancies will be discussed in Section 4.3.

We can also see from Figure 5, especially from the PyTorch models, that the models converge quite quickly at around epoch 25. After this point, the loss and accuracy starts oscillating quite a bit without learning much more.

Finally, when testing our model on a separate unseen test set, the results were also great. Our custom FFNN achieved an accuracy of 98.3%, whilst the PyTorch version achieved 97.4%. As seen in Figure 6, our model is able to classify all but two samples correctly.

When it comes to hyperparameters, the learning rate had a clear effect on the model. Figure 7 shows the accuracy and loss as function of the learning rate for different number of neurons and a single layer. Here, we see that a too small or too large learning rate will reduce the performance. Meaning, that there is a sweet spot for the learning rate, which our models in Table 3 hit.

This also shows that other hyperparameters such as the number of neurons in a layer may not have had that much of an effect on the model's performance as seen in Figure 7 where we can see the same trend for all the different number of neurons stays the same - they perform bad for too small or large learning rates, and there is a sweet spot. We also extended the model to include two hidden layers, and once again checked the loss and accuracy as a function of the learning rate as seen in Figure 8. Again, the same trends can be seen here with a sweet spot for the learning rate. Furthermore, the model did not seem to need much help from the $L2$ regularization, as we were able achieve great performances with $\eta = 0.001$.

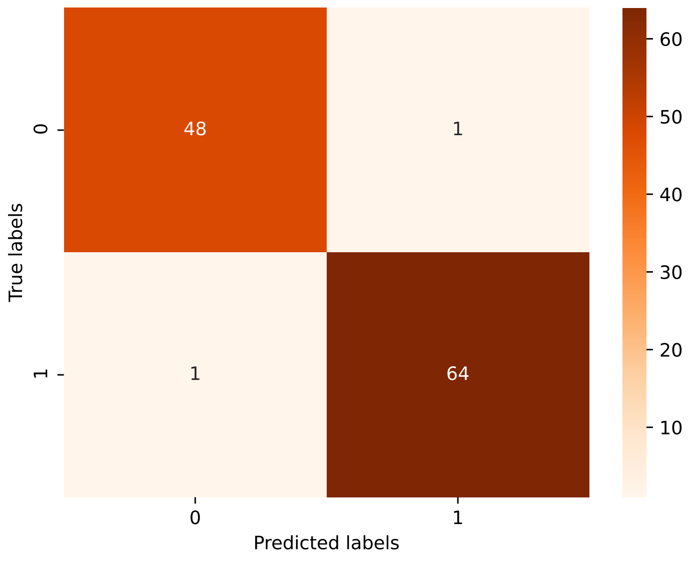
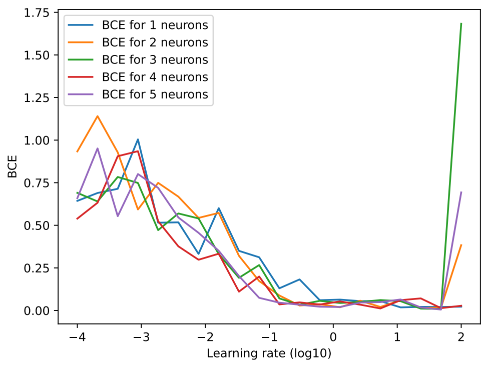
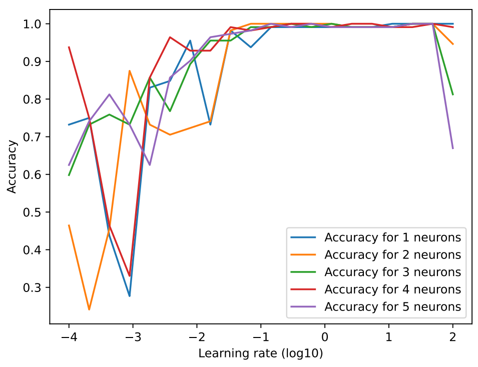


Figure 6: Confusion matrix of final test using model 2 from Table 3



(a) Loss



(b) Accuracy

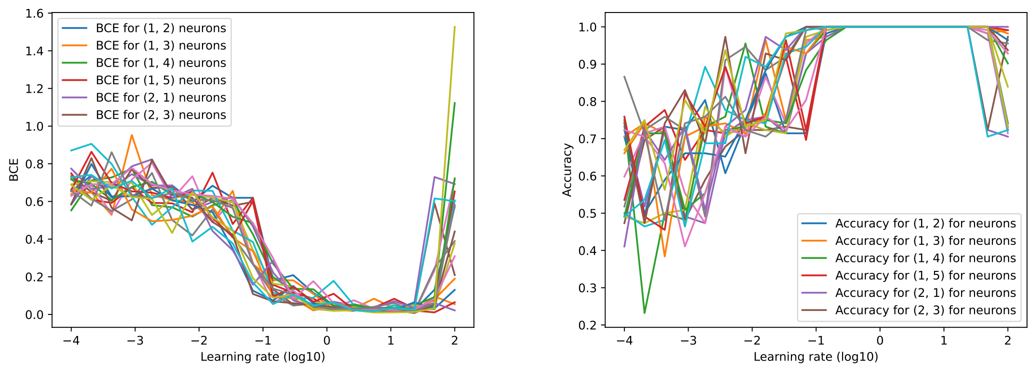
Figure 7: Evaluation loss and accuracy for varying learning rates for FFNN with a single layer and different amount of neurons.

3.2.2 Logistic Regression

As we have seen, the FFNN neural network with a single hidden layer and a single neuron performed very well on the classification task as discussed in Section 3.2.1. This architecture is very similar to the one used in logistic regression. Thus, we wanted to make sure to point out that the logistic regression differentiates itself from model 1 in Table 3 in the sense that the FFNN has an additional output layer, which the logistic regression does not have. So, if we were to remove the output layer of the FFNN model 1, we would actually have a logistic regression model, as model 1 has a sigmoid activation in the hidden layer, which maps out to a single output neuron.

Again, the logistic regression model performed exceptionally well on the problem too. The accuracy on our validation set would again reach 100% with many different configurations. Two of these models are presented in Table 4.

As we can see, the hyperparameters which can achieve 100% accuracy is vastly different. This just shows how good the dataset itself is, and from a first look, it doesn't seem like the hyperparameters matter that much. However, when looking closer at the learning rate and regularization term in Figure 9, it can be seen that the learning rate has a substantial higher impact on the performance than the regularization term. From the accuracy heatmap, a good learning rate will give us a high accuracy for almost any regularization terms (with some exceptions for $\lambda = 0.2154$).

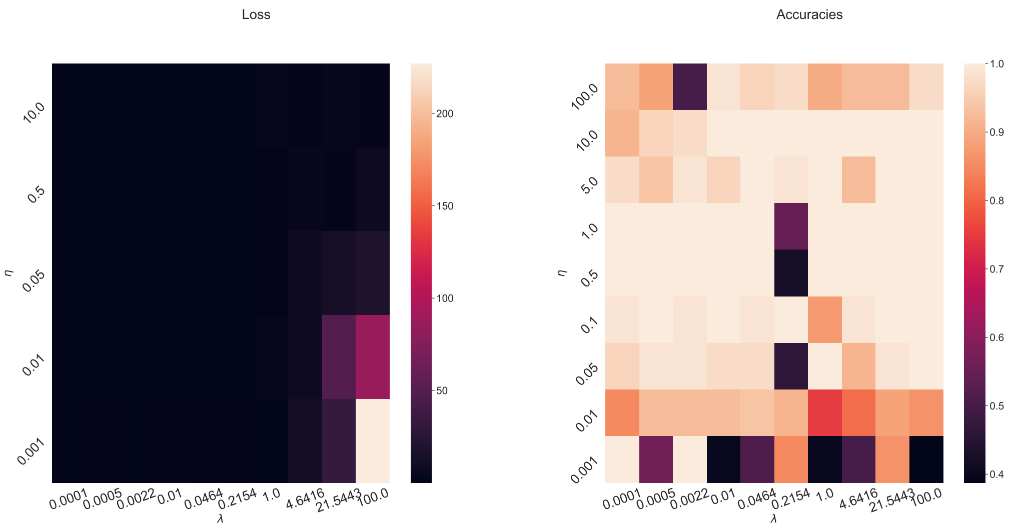


(a) Loss (b) Accuracy

Figure 8: Evaluation loss and accuracy for varying learning rates for FFNN with a two hidden layers and different amount of neurons. Note that only the first 5 combinations of neurons are labeled, but the figure is just to prove a point in Section 3.2.1

Table 4: hyperparameters for two logistic regression models, both with 100% accuracy on the evaluation set.

hyperparams	model 1	model 2
LR (η)	10	0.5
Reg term (λ)	100.0	0.00046
Batch size	16	16



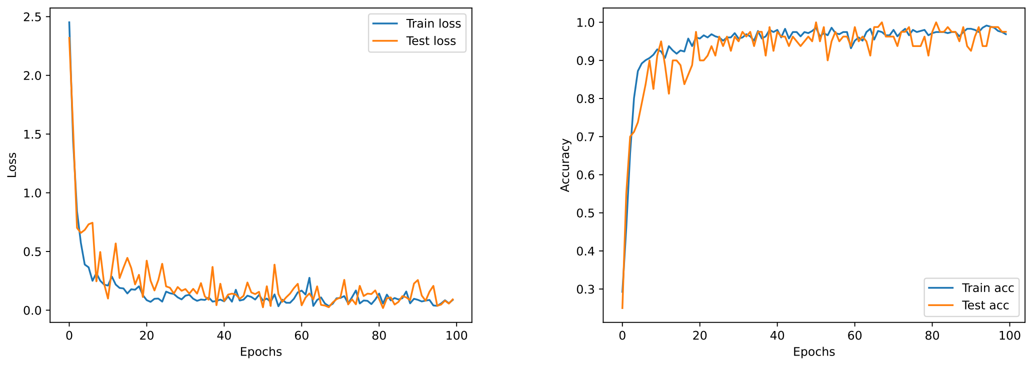
(a) Loss (b) Accuracy

Figure 9: Evaluation loss and accuracy for varying learning rates and regularization terms for logistic regression

Again, as several models achieved perfect accuracy on the evaluation set, we chose to work with the model with the best loss. For the two models in Table 4, this obviously turned out to be model 2, as model 1's regularization term would naturally increase the loss, as it is baked into the BCE loss. Thus, we chose to go with model 2 for our final test.

For model 2, the evaluation loss was 0.009, whilst the test loss turned out to be 0.06. The accuracy on the test set was a great 98.3%, and the confusion matrix was the exact same as the one we got from our FFNN seen in Figure 6. The learning for our optimal model was also very smooth, as seen in Figure 10.

We also ran Scikit-learn's LogisticRegression model for comparison. We fed all our training data



(a) Loss (b) Accuracy

Figure 10: Training and evaluation loss & accuracy for logistic regression model 2 as in Table 4

without splitting it any further into evaluating into the model. The results were, as expected, also phenomenal, with a test accuracy of 98.3% with the exact same confusion matrix as fig. 6. However, after running training and testing for the Scikit-learn implementation a couple of times, it was able to reach 100% accuracy. However, our implementation is still phenomenal, considering we only used our final test set once and not many times like we did with Scikit-learn.

4 Discussion

4.1 Linear regression using stochastic gradient decent

During the grid search for best *hyperparameters*, a choice had to be made for what range and how many values were to be used for each of the hyperparameters. A large interval and many values would be computational expensive and take a long time to complete, but it could find hyperparameters which are more optimal for the task. We made a compromise in the selections of interval and number of samples within the interval. This was done to make the calculations finish within a reasonable time on a laptop computer.

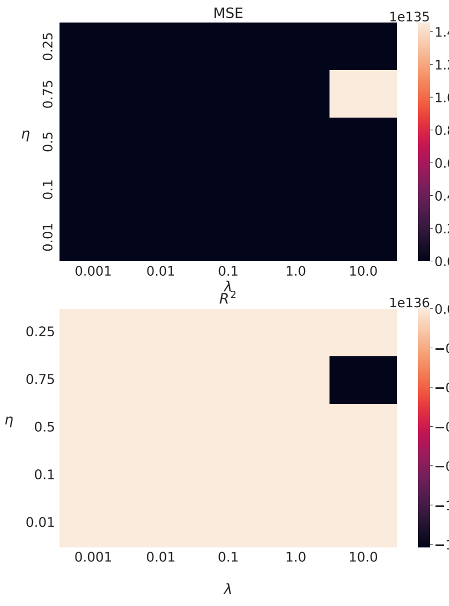
From results in Figures 2a and 2b we see that there is s small difference in the MSE between the use of SGD and without. The R^2 scores do show similar performance on OLS and ridge when using SGD, but on some model complexities with SGD preform slightly better. The grid search for best model does also present issues with OLS on complexity 10 where the R^2 score is negative. This is the case without SGD. A reason could be that the design matrix is ill conditioned for this complexity, but further investigation would be needed to conclude on the reason for this, which we did not manage to make time for.

Using SGD compared with using an analytical optimization through matrix inversion with NumPy is a lot less optimized and takes more time. For one, it iterates the number of epochs needed to optimize the parameters. Secondly, our SGD code is implemented in pure Python code which is drastically slower than NumPy, which is compiled in *C-code* for performing the matrix inversion.

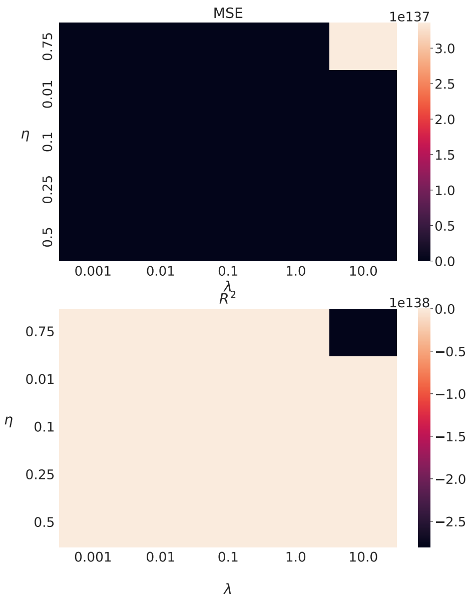
For ridge regression with SGD, we looked at the relationship between learning rate and the regularization term λ. The heatmap in Figure 11 shows the relation between learning rate, regularisation term and MSE/R^2 score. Over models from complexity 1 to 10, shows as in Figure 11 that it is not much different in choice of learning rate and regularization term, except at higher values of both.

4.2 Regression using FFNN

The results presented in Section 3.1.2 were quite good. The best R^2 score was 0.96 on the validation dataset and 0.84 on test dataset, which are decent result. Compared to the results of linear regression in Section 3.1.1, where the best R^2 score of ridge with SDG was 0.82, the FFNN achieved the best score of all our graident based models. Comparing the FFNN with the analytical solution from OLS, where the best $R^2 = 0.95$, the FFNN did not quite reach up. The FFNN will also be able to model more complex functions then a linear regression can.



(a) Complexity 1



(b) Complexity 5

Figure 11: Relationship between learning rate η and λ for two different complexities

4.3 Our implementation vs. PyTorch

As discussed when comparing our FFNN with the same network in PyTorch in Section 3, the results were very similar, but not exactly the same. This is due to several reasons.

First of all, as stated in Section 2.9, we initialized our weights from a normal distribution. On the other hand, PyTorch initalizes their linear layers based on a uniform distribution [2]. Already from this point, the search landscape will be a bit different, as the models will start their searches from different starting points, which may lead them to different local minimum. This can clearly be seen in Figure 5. Here, we have initialized our hyperparameters exactly the same, but the initial training losses at epoch 0 for the two models are quite different, with our custom model having an initial loss around 0.8, and the PyTorch implementation has a loss of 0.45.

Secondly, non of our experiments were performed with a specific random seed. Thus, PyTorch and NumPy could have had different seeds, which yields different results.

4.4 Classification: FFNN vs. Logistic regression

The classification results of our two models were very impressive, both achieving 100% accuracy on the evaluation dataset, and also 98.3% on the unseen test set. Thus, from a pure performance standpoint, neither of these two models are preferred over the other. However, there are a lot of other factors that may be relevant for choosing a model for our classification problem using the Wisconsin Breast Cancer dataset. It is also important to note that our dataset has very good features which makes the classification problem quite a bit easier than a typical classification problem.

As already discussed in Section 3.2.2, our optimal FFNN was very similar to the logistic regression model. Furthermore, Figure 7 and Figure 8 show that the number of neurons and layers does not matter a whole lot when trying to find a good model, but the learning rate is by far the most important hyperparameter. Thus, from an architectural perspective, we gain close to nothing from having extra hidden layers or neurons when it comes to performance for our given problem. The extra hyperparameters of the FFNN just makes the grid search longer to perform compared to our logistic regression model.

Furthermore, as the FFNN has more layers, it also computes the output slower than logistic regression. This also applies to the learning process, as the back propagation algorithm is blatantly more complex and more computationally heavy than the simple SGD used for the logistic regression. This amplifies the inefficiency of the FFNN compared to the logistic regression.

[2]https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/linear.py

In general, we prefer simple models over complex ones if the choice is present. This is because simpler models are, amongst other things, faster to run and easier to interpret. From this perspective, the logistic regression's simple architecture is preferred over the FFNN, as the logistic regression model is both simpler and easier to interpret than a FFNN.

All in all, for the simple problem of classification of the Wisconsin Breast Cancer dataset, the logistic regression model is by far prefered over a FFNN.

4.5 Further work

As this project was performed with limited time, we were not able to perform a grid search on all the hyerparameters we would have hoped to do. Further searches could be done on batch size, more activation functions, a finer set of learning rates and regularization terms, and also the momentum term for the neural networks. Another thing to test, is also how to initalize our weights. As seen in section 4.3, frameworks such as PyTorch initializes their weights using a uniform distribution, while limited ourselfs to only look at the Gaussian distribution. Therefore, looking at different ways of initializing the weights is definitely an interresting experiment to conduct.

To further test the models, other datasets can be tested, such as the terrain data we looked at in project 1 or more complex classification datasets with less established features.

5 Conclusion

We have in this project looked at two different problems in regression and classification. In both of these problems, we saw that having a more complex model does not necessarily provide better results than having a simpler one. In the regression case, we explored the Franke Function, and it turned out that the simplest model possible was also the best for this problem, which brings with it several benefits such as interpretability and fast computation. These benefits are also present in the simpler model used for classification of the Wisconsin Breast Cancer dataset, namely logistic regression, compared to the more complex neural network. These added benefits of the logistic regression makes it more desirable compared to the neural network for our given dataset, as the performance for both these models were near perfect. This great performance was much attributed to the phenomenal features of the classification dataset. All in all, this project has shown the importance of having good data quality and also that more complexity does not always yield better results. Having these points in mind, you may not only get better results, but also save a lot of time.

References

[1] T. Kvalvaag, M. Odorczuk, and W. Ho, "Fys-stk4155 - project 1," Project report from UiO FYS-STK4155 project 1., 2021.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[3] Dougal Maclaurin, David Duvenaud, Matt Johnson, Jamie Townsend, *Autograd*, version 1.3, Jun. 25, 2019. [Online]. Available: `https://github.com/HIPS/autograd`.

[4] R. Patgiri, S. Nayak, T. Akutota, and B. Paul, "Machine learning: A dark side of cancer computing," 2019. arXiv: `1903.07167 [cs.LG]`.

A Appendix

A.1 Deriving Back Propagation

Building on Section 2.4.3, to simplify the calculations, we choose the cost function to be half the squared error, which is defined as

$$C(\hat{W}) = 1/2 \sum_{i=0}^{N-1} (y_i - t_i)^2 \tag{33}$$

where $\hat{W}$ is our model, y_i is the output of our forward pass as defined in Equation (9), N is the number of outputs, and t is our target values. Again, any other cost function will also do. Furthermore, we can substitute y_i in Equation (33) by a_i^L, where L is the last layer of our network. Thus, a_i^l denotes the activation for layer $l \in [1, .., L]$, and we get

$$C(\hat{W}) = 1/2 \sum_{i=0}^{N-1} (a_i^L - t_i)^2 \tag{34}$$

As discussed in Section 2.4, we can write the activation of a layer l as

$$a_j^l = f^l(\sum_{i=0}^{M-1} w_{j,i}^l a_i^{l-1} + b_j^l) \tag{35}$$

where a^{l-1} is the activation of the previous layer, M is the number of neurons in the previous layer, and f^l is the activation function of layer l. We will also denote the sum of the dot product and the bias as

$$z_j^l = \sum_{i=0}^{M-1} w_{j,i}^l a_i^{l-1} + b_j^l \tag{36}$$

which will come in handy for the calculations.

As we want the gradient of the loss with respect to each of the weights in our network, we will apply the the chain rule on our cost function with respect to each of the weights. Starting at the last layer L, we get

$$\frac{\partial C(\hat{W})}{\partial w_{jk}^L} = (a_j^L - t_j) \frac{\partial a_j^L}{\partial w_{jk}^L} \tag{37}$$

And from Equation (36), the last partial derivative can then be further written as

$$\frac{\partial a_j^L}{\partial w_{jk}^L} = \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial w_{jk}^L} = \frac{\partial a_j^L}{\partial z_j^L} a_i^{l-1} \tag{38}$$

as z_j^L is a linear function of w^L, we get

$$\frac{\partial z_j^L}{\partial w_{jk}^L} = a_i^{l-1} \tag{39}$$

which yields

$$\frac{\partial C(\hat{W})}{\partial w_{jk}^L} = (a_j^L - t_j) \frac{\partial a_j^L}{\partial z_j^L} a_i^{l-1} \tag{40}$$

As we can see,

$$(a_j^L - t_j) \tag{41}$$

is simply the derivative of our cost function applied on a^L, and

$$\frac{\partial a_j^L}{\partial z_j^L} \tag{42}$$

is the the derivative of our activation in Equation (35) with respect to the input of the activation Equation (36). Thus, as f is our activation function, we can define our output error as

$$\delta_j^L = (a_j^L - t_j)\frac{\partial a_j^L}{\partial z_j^L} = f'(z^L)\frac{\partial C}{\partial (a_j^L)} \tag{43}$$

With this, we now have a compact expression for the gradient of our cost function with respect to our output weights

$$\frac{\partial C(\hat{W})}{\partial w_{jk}^L} = \delta_j^L a_i^{l-1} \tag{44}$$

which can be used as the gradients in SGD to compute the updated weights in the output layer.

Furthermore, for a general layer l, we can write the error as

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{45}$$

Since we are propagating backwards, we want to express this in terms of the next layer $l+1$. Thus, since in a FFNN, the output of a single neuron becomes the input for each of the neurons in the next layer, we will have to sum up the each of the outputs z_j^l contributes to, giving us

$$\delta_j^l = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{46}$$

And from Equation (45), we can write

$$\frac{\partial C}{\partial z_k^{l+1}} = \delta_k^{l+1} \tag{47}$$

yielding

$$\delta_j^l = \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{48}$$

Furthermore, from Equation (36), for $l = l+1$ we finally get

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l) \tag{49}$$

One last note is that Equation (43) can also be interpreted as the partial derivative of the cost function with respect to the biases b_j^L, which yields

$$\delta_j^L = \frac{\partial C}{\partial b_j^L}\frac{\partial b_j^L}{\partial z_j^L} = \frac{\partial C}{\partial b_j^L} \tag{50}$$

And with these error terms, we can now finally apply our gradients to update our weights using SGD through the back propagation algorithm in Section 2.4.3

B Hyperparameter grid search linear regression

The hyperparameters and candidate values in Table 5 were used during grid search for best model.

Table 5: Hyperparameter values used in linear regression with SGD

Hyper parameter	Values
Learning rate	$\{0.01, 0.1, 0.25, 0.5, 0.75\}$
Batch size	$\{1, 2, 4, 8\}$
Lambda	$\{0.001, 0.01, 0.1, 1, 10\}$
Momentum	{ True, False }
Alpha	$\{0.1, 0.5, 0.9\}$

C Linear regression results

Tables 6 and 7 show MSE and R^2 for each of the best models found in grid search for each model complexity.

Table 6: R^2 for best model at each complexity for each of methods tested.

complexity	ols	ols with sgd	ridge	ridge wwith sgd
1	0.794431	0.745330	0.794510	0.823643
2	0.804827	0.770621	0.815885	0.819012
3	0.703422	0.791546	0.738660	0.744076
4	0.898457	0.764059	0.753027	0.721808
5	0.767559	0.660341	0.807062	0.722013
6	0.536463	0.620492	0.844232	0.734233
7	0.965841	0.620535	0.852955	0.733028
8	0.867233	0.674632	0.845182	0.700165
9	0.528669	0.581664	0.829971	0.670498
10	-0.254508	0.680122	0.811595	0.609800

Table 7: MSE for best model at each complexity for each of methods tested.

complexity	ols	ols with sgd	ridge	ridge with sgd
1	0.012855	0.015926	0.012851	0.011029
2	0.012205	0.014344	0.011514	0.011318
3	0.018547	0.013036	0.016343	0.016004
4	0.006350	0.014755	0.015445	0.017397
5	0.014536	0.021241	0.012066	0.017384
6	0.028988	0.023733	0.009741	0.016620
7	0.002136	0.023730	0.009196	0.016695
8	0.008303	0.020347	0.009682	0.018750
9	0.029475	0.026161	0.010633	0.020606
10	0.078452	0.020004	0.011782	0.024402