FYS 4155 Project 2 - Classification and Regression, from linear and logistic regression to neural networks

Hakan Abediy

November 19, 2023

Abstract

This project report focuses on using supervised learning to solve regression and classification problems. It discusses various algorithms such as Gradient Descent (GD)/Stochastic Gradient Descent (SGD), Logistic regression and neural networks.It is vital to configure the parameters such as hyperparameters, activation functions and adaptive optimization methods for good model performance. Complexity was added to the plain SGD and GD by adding momentum and adaptive optimization. For a polynomial problem plain SGD with momentum gave the lowest cost score compared to other GD and SGD algorithms. FFNN performed greatly in a regression porblem compared to OLS/Ridge regression. It also performed welll in classficaition analysis for a simple classficaition problem and with the Wisconsin Breast Cancer dataset. Logistic regression performed the best compared to the FFNN for the simple classification problem. In summary, FFNN showed itself to be a versatile tool that gave great results for the problems studied in this project.

1 Introduction

This project is an example of using supervised learning to solve classification and regression problems. There are many methods to solve supervised learning problems. In Project 1 linear regression was used and in this project we will dive deeper into the world of machine learning by studying algorithms like gradient descent, logistic regression and neural networks. Each algorithm will start out simple and become progressively more advanced and refined. Lastly, the different algorithms will be compared and discussed to see their strengths and weaknesses.

Gradient descent and Stochastic Gradient Descent will be studied with a fixed learning rate and then momentum will be added. The results will be given as a function of hyperparameters. Then Adagrad will be added to these models to tune the learning rate. Thereafter the option of using RMSprop and ADAM will be added to the models. Then we replace the analytical gradient with Autograd.

Thereafter we will create our own Feedforward Neural Network (FFNN) using the sigmoid function as the activation function. We will train our network and compare the solutions with with OLS and Ridge regression. Then we will add further activation functions as ReLU and ReLU-Leaky and use them on the output and hidden layer. We started by using the FFNN on a polynomial regression problem, and then we used it on the classification problem of analyzing the Wiscosnin Breast Cancer data set where the goal is to predict whether a tumor is malignant or benign based on various features.Then, we write our own logistic regression code to solve a simple classification problem as a function of hyperparameters and we compare its results with the FFNN. Lastly, we do critical evaluation of the different algorithms.

This project report will is in four chapters. Chapter 1 is an introduction, Chapter 2 is about theory of the different algorithms and concepts and method which include how the theory is applied and setup for this specific project. Chapter 4 is the results and discussion which compares the different algorithms tested. Lastly, chapter 5 is a conclusion of the findings in the discussion.

All the results and code are available on the GitHub repository: https://github.com/hakanoa/FYS-4155—Project-2

2 Theory and method

2.1 Gradient descent and Stochastic Gradient descent

In linear regression, the goal is to find the beta coefficients that minimizes the difference between the predicted values and the actual values of the target variable. Gradient descent (GD) is a numerical optimization method which main idea is to iteratively adjust the model's parameters in the direction of the steepest descent of the cost function. This process continues until the algorithm converges to a local or global minimum. This is an effective way of optimizing the model's performance. GD uses the entire dataset or training set to compute the gradient and find the optimal solution. In each iteration, it runs through all the samples in the training set to update a parameter.

Chapters 2.1.1 and 2.1.2 explains the algorithm behind GD and SGD.

2.1.1 Gradient Descent

OLS has the cost function:

$$C(\beta) = \frac{1}{n}||X\beta - \mathbf{y}||_2^2 = \frac{1}{n}\sum_{i=1}^{100}\left[(\beta_0 + \beta_1 x_i)^2 - 2y_i(\beta_0 + \beta_1 x_i) + y_i^2\right] \tag{1}$$

Where X is the design matrix, β is the coefficient matrix.

We want to find the β value that minimizes $C(\beta)$. We find the minimum by finding the gradient by calculating $\partial C(\beta)/\partial\beta_0$ and $\partial C(\beta)/\partial\beta_1$.

This gives us:

$$\nabla_\beta C(\beta) = \frac{2}{n}\begin{bmatrix} \sum_{i=1}^{100}(\beta_0 + \beta_1 x_i - y_i) \\ \sum_{i=1}^{100}(x_i(\beta_0 + \beta_1 x_i) - y_i x_i) \end{bmatrix} = \frac{2}{n}X^T(X\beta - \mathbf{y}), \tag{2}$$

The update rule for β can be given as:

$$\beta(t+1) = \beta(t) - \eta\nabla_\beta C(\beta), \tag{3}$$

where η is the learning rate.

Ridge regression has the cost function:

$$C_{\text{ridge}}(\beta) = \frac{1}{n}||X\beta - \mathbf{y}||^2 + \lambda||\beta||^2,\ \lambda \geq 0. \tag{4}$$

We need to minimize the gradient which for Ridge regression is given as:

$$\nabla_\beta C_{\text{ridge}}(\beta) = \frac{2}{n}\begin{bmatrix} \sum_{i=1}^{100}(\beta_0 + \beta_1 x_i - y_i) \\ \sum_{i=1}^{100}(x_i(\beta_0 + \beta_1 x_i) - y_i x_i) \end{bmatrix} + 2\lambda \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} = 2(X^T(X\beta - \mathbf{y}) + \lambda\beta). \tag{5}$$

As with OLS the update function is:

$$\beta(t+1) = \beta(t) - \eta\nabla_\beta C(\beta), \tag{6}$$

Based on this, Linear regression with GD is performed by:

1. Define X and y
2. Set β as random values, but it will be updated through the process
3. Define the hyperparameters learning rate and number of iterations.
4. Iterate over the data:
   (a) Compute prediction $\hat{y}$
   (b) Compute the OLS or Ridge gradients analytically by using Equations 5 or 2 or automated by using the Autograd library.
   (c) Compute the cost function $Cost_{OLS}$
   (d) Update β using the learning rate, gradient(additionally velocity if using momentum).
   (e) Repeat the steps for a specified number of epochs

2.1.2 Stochastic Gradient descent

Stochastic Gradient Descent (SGD) is an optimization algorithm derived from the more traditional Gradient Descent (GD), where instead of utilizing the entire training dataset in each iteration, SGD randomly selects either a single data point or a small subset. This selective sampling approach introduces a level of randomness into the parameter update process, making SGD more computationally efficient and suitable for large datasets. The use of a limited sample size in each iteration introduces noise but allows for quicker updates, enabling faster convergence during the training of machine learning models. This process of updating the parameters for each data point is what makes it stochastic.

OLS with SGD was performed in the same was as with GD, but with an extra step between 3 and 4 in 2.1.1 which is to:

- Randomly shuffle X and y to ensure that the model sees the data in a different order during each epoch/iteration.

2.1.3 Limitations

Despite its simplicity and wide applicability, gradient descent has some limitations. One of the main challenges is the choice of the learning rate (η). If the learning rate is too small, the algorithm may take a long time to converge, while a too-large learning rate can cause the algorithm to overshoot the minimum and fail to converge. Therefore, selecting an appropriate learning rate is crucial for the success of the gradient descent algorithm. Additionally, gradient descent may get stuck in local minima, preventing it from finding the global minimum of the cost function. [3]

2.1.4 Learning rate

Learning rate is a hyperparameter which which controls how big a step we take towards the minimum of the cost function. The value of the learning rate is hugely influential for the performance of the neural network. A too small learning rate is computationally expensive and and a too large learning rate can lead to a suboptimal solution or the minimum point being skipped. It is therefore important to choose a an appropiate learning rate. A learning rate can be constant or variable based on the needs of the problem.

2.1.4.1 Momentum Momentum is an optimization technique used to speed up the convergence of gradient-based optimization algorithm. The basic idea behind momentum is to add a part of the previous gradient update to the current gradient update, which can help smooth out the optimization process and overcome local minima more efficiently.

The update rules for β with momentum can be given as:

$$v(t+1) = mv(t) + \eta\nabla_{\beta}C(\beta)$$

and

$$\beta(t+1) = \beta(t) - v(t+1)$$

. Where $v(t)$ and $m(t)$ is the velocity function and momentum function, respectively.

2.1.4.2 Adaptive optimization methods

2.1.4.3 Autograd Autograd is used to automatically compute gradients of mathematical functions, instead of calculating them analytically.

Autograd is implemented in Python through the library ”autograd” and gradients are calculated by using the ”grad” function.

2.1.4.4 Adagrad Adagrad adapts the learning rate for each parameter based on the historical gradient information.

2.1.4.5 RMSprop RMSprop adjusts the learning rate for each parameter individually by using a moving average of the squared gradients. This can help address issues with learning rates that are too large or too small.

The RMS update rule for θ is given by:

$$g_t = \nabla_\theta E(\theta)$$

$$s_t = \beta s_{t-1} + (1-\beta) g_t^2$$
$$\theta_{t+1} = \theta_t - \eta_t \frac{g_t}{\sqrt{s_t + \epsilon}}$$

β controls the averaging time of the second moment, η_t is a learning rate typically chosen to be 10^{-3} and $\epsilon \sim 10^{-8}$ is a small regularization constant to prevent divergences. [3]

2.1.4.6 Adam Adam combines the concepts of momentum and RMSprop and adapts the learning rate for each parameter. Adam is known for its effectiveness and is widely used in practice.

The Adam update rule for θ is given by [3]:

$$\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta})$$

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t$$
$$\mathbf{s}_t = \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2$$
$$\mathbf{m}_t = \frac{\mathbf{m}_t}{1-\beta_1^t}$$
$$\mathbf{s}_t = \frac{\mathbf{s}_t}{1-\beta_2^t}$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{m}_t}{\sqrt{\mathbf{s}_t} + \epsilon}$$

β_1 and β_2 set the memory lifetime of the first and second moment and are typically taken to be 0.9 and 0.99. [3]

2.1.5 Initilization

The input polynomial data and hyperparameters when running OLS and Ridge with SGD and GD was the following:

```

# Define the number of data points
number of data points = 100

# Create a column vector x
x = column vector(n)

y = 2.0 + 3 * x + 4 * x^2

beta = random normalized array
v = array with zeros

```

```
# Create a design matrix X with columns of 1s, x, and x^2
X = design matrix(x)
learning_rates : 0.01, 0.08, 0.1, 0.5, 0.6


```

2.2 Neural Network

Neural Networks (NNs) are computational models that emulate the biological neural networks of the human brain. They can solve both classification and regression problems. They learn tasks from examples without any task-specific programming. The Universal approximation theorem states that a NN with a hidden layer can approximate any continous function to a certain degree of accuracy. An NN consists of an input layer, an hidden layer and an output layer. Each layer consist of one or more nodes. The configuration of the network and its hyperparamters impacts the results of the network. The number of layers in the NN controls the complexity of the NN, but it doesn't necessarily improve the accuracy of the solutions. The number of nodes also plays a role. Too many nodes can lead to underfitting and high bias while too few nodes can lead to overfitting and high variance. The way to configure the network is through experimentation. The final layer is the output layer, and during training, the goal is to make the network's output match the desired output. The hidden layer is important in how the NN achieves the desired output.

The Feed-Forward Neural Network (FFNN) is the earliest and simplest form of NNs. In FFNN, information only travels in one direction, which is forward, from input to output. Each node in the NN calculates its output based on the weighted sum of its inputs and bias thereafter the activation is applied. Often MSE is used to calculate the cost for regression problems and logistic loss is often used in classification problems.

See Figure 1 for an example of an FFNN.

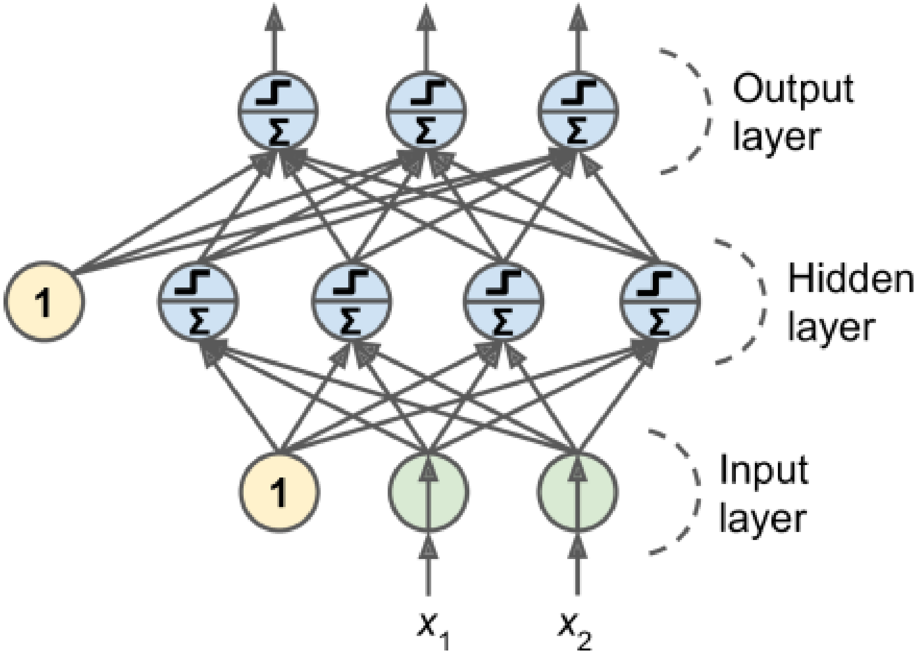


Figure 1: An example of an FFNN network [2]

2.2.1 Back propogation

First the input is passed though the network forward to achieve a prediction. We then need to train our network. The backpropogation training algorithm's goal is to minimize the cost by adjusting the weights and biases of the network.It works by iterative adjusting weights and biases in the network to minimize the cost. The minimized cost is found the finding by adjusting the gradient of cost with respect to the weights and biases. The method is based on the Gradient Descent method. [1]

2.2.2 Activation function

An activation function is a key feature of a NN.They define how the weighted sum of the input is transformed into an output from nodes in a layer of a NN. The choice of activation functions have a signifcant impact on the performance of the NN.

In our FFNN three different activation functions were used; sigmoid, ReLU and Leaky ReLU.

Sigmoid function is often used in the output layer for classification analysis and the function is given by

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

ReLU is given by:

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

It's output is the input value if the input is positive and otherwise zero. Relu gives non-linearity to the NN.

Leaky ReLU is given by:

$$f(x) = \begin{cases} x & x > 0, \\ \alpha x & x \leq 0. \end{cases}$$

α is a small positive constant that determines the slope of the activation function for negative input values.

2.2.3 Algorithm

This is the algorithm used to receive the results for FFNN in Chapters 3.2 and 3.3.

1. Define X and y
2. Split the dataset into training and testing sets.
3. Define the FFNN configuration by selecting the number of layers, nodes in each layer, and the activation functions. Set β as a random array.
4. Feedforward the input data through the FFNN and compute the predicted output. Apply the activation function at the desired nodes.
5. Compute the cost function (like accuracy score or logistic loss/cross entropy).
6. Update the weights and biases of the FFNN by calculating the gradients and adjusting the parameters using, for example, SGD.
7. Iterate for a fixed number of epochs or until convergence.
8. Evaluate the model and, if necessary, tune hyperparameters by experimenting.

2.3 Logistic regression

Logistic regression is one common algorithm that is especially suitable for binary classification problems for supervised learning. SGD can be used to train models with logisitc regression. Logisitc regression with SGD is performed by:

1. Define X and y.

```
    x = random array from x with shape (n,1)
    y = convert_to_binary(x, threshold = 0.5)
    X = add_ones_to_column(x) # Design matrix. Add ones to the right and
    ↪  left of the column x
```

2. Initialize weights and biases, which will be updated later. In this case they were initialized as random values.
3. Define the hyperparameters learning rate and number of iterations.

4. Define the logistic function/sigmoid function. In logistic regression, the likelihood that a data point x_i belongs to a category $y_i = 0$ or 1 is expressed through the Sigmoid function. This function gives the probability of a certain event occurring [3]: $\sigma(t) = \frac{1}{1+e^{-t}}$

5. Compute the logistic loss function, see Chapter 2.4.1.1.

6. Iterate (until convergence)

   (a) Calculate the gradient of the cost function with respect to weights to bias using the current parameters

   ```python
   # Function to predict probabilities
   def predict_prob(X, beta):
       return some_function_to_predict(X, beta)

   # Calculate predicted probabilities
   y_prediction = predict_prob(X, beta)

   # Calculate error
   error = y_prediction - y_actual

   # Calculate gradient
   gradient = (1.0 / X.shape[0]) * X.T @ error
   ```

   (b) Update the weights and bias with the updated gradient and learning rate as: $w = w - \alpha \nabla C(w, b)$ and $b = b - \alpha \nabla C(w, b)$

7. Predict new data based on the updated weights and bias

8. Convert the predicted probabilites and convert them to binary classification (0 or 1).

2.4 Classification analysis

Classification analysis deal with outcomes that are discerete values rather than continuous values from regression problems.Most often there are only two outcomes like binary or true/false.

2.4.1 Simple classification analysis - intilization

First, we conducted a classification analysis with a simple input as seen in 2.4.1.2 and 2.4.1.3, and compared the results from the FFNN and Logisitc Regression. 2.4.1.1 describes the cost function used in this analysis.

2.4.1.1 Logsitic loss Logsitic loss/cross-entropy loss is a cost function often used in binary classification problems when using NN and logistic regression. The cost function can be written as:

$$C = -mean(y\log(p) + (1-y) * \log(1-p))$$

p is the predicted probability that the instance is 1. y is the true value, either 0 or 1.

2.4.1.2 FFNN

$$X = \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

$$\begin{aligned} \text{n\_neurons\_hidden: } & 5{,}10{,}15 \\ \text{n\_neurons\_output} &= 1 \\ \text{learning rate: } & 0.001,\ 0.01,\ 0.1 \\ \text{number of epochs: } & 500,\ 1000,\ 1500 \end{aligned}$$

Weights are initialized with random values, and biases are initialized with small non-zero values.

2.4.1.3 Logisitc regression

$$X = \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

$$\begin{aligned} \text{learning rate: } & 0.001,\ 0.01,\ 0.1 \\ \text{number of epochs: } & 500,\ 1000,\ 1500 \\ \text{beta} &= \text{random normal distribution(X.number of columns)} \end{aligned}$$

2.4.2 Cancer classification analysis - initialisation

2.4.2.1 FFNN Then, we studied the Wisconsin Breast Cancer dataset where the goal is to predict whether a tumor is malignant or benign based on various features using our FFNN.

We measured the performance of our data set by using accuracy score. The accuracy score is the number of correctly guessed targets t_i divided the number of targets n. It can be written as:

$$Accuracy = \frac{\sum I(tt_i = y_i)}{n}$$

Where I is the indicator function and is 1 if $y_i = t_i$ and 0 otherwise. y_i is the output of the FFNN.

```
load breast cancer data
X = extract features from data
y = extract target labels from data
n_neurons_hidden: 5,10,15
n_neurons_output = 1
learning rate: 0.001, 0.01, 0.1
number of epochs: 500, 1000, 1500
```

Then the data is splitted into training and testing data. The number of features and inputs are defined by the shape of the design matrix X. The number of hidden layers are defined as between 5 and 15. The number of epochs are defined as between 500 and 1500. The learning rate is set between 0.001 and 0.1. The values are based on experimenting with different configurations of hyperparameters.

3 Results and Discussions

3.1 Polynomial function - OLS with GD and SGD

The lowest cost scores from the all the GD and SGD runs are seen in Table 1. These results highlights the importance of selecting the right optimization methods for a given problem. The choice of learning rate and learning rate technique is also important for the algorithm.

The lowest cost score is achieved using plain SGD with momentum, followed by GD with momentum. Using automated differentiation with SGD with ADAM yields the third lowest cost score.

The highest cost score is from GD with no momentum. This is not unexpected as gradient descent without momentum tends to converge more slowly.

Using momentum significantly reduces the cost score by approximately $\approx 10^{-20}$. This can be because momentum helps the optimization process by allowing the algorithm to accumulate velocity in directions with consistent gradients

The lowest cost score using SGD from SKlearn with a learning rate $= 0.01$ gives the second highest cost score. Choosing an adaptive learning rate instead gives a 14.6 % lower cost score. The reason for the relatively poor performing cost score can be the lack of time I had to optimize the sklearn cost score to this problem. The reason why the SGD SKlearn cost score is higher than the plain SGD without momentum might be because the default settings from Sklearn are not well-suited for this particular optimization problem, highlighting the need for careful parameter tuning.

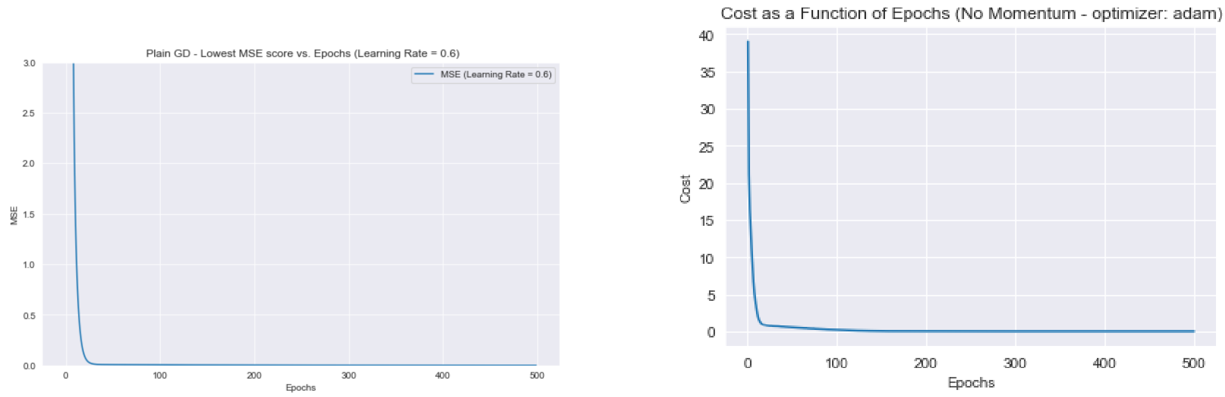
SGD with Adam might not give the lowest cost score due to high sesitivity of choice of hyperparamters.

The extremely low cost scores in most cases suggest that the optimization process is converging very well.

Optimization Method	Lowest Cost Score	Learning Rate
Plain SGD with Momentum	1.46×10^{-29}	0.6
Plain GD with Momentum	1.21×10^{-21}	0.5
SGD with Adam (No Momentum)	2.29×10^{-21}	-
Plain SGD (No Momentum)	3.19×10^{-6}	0.6
SGD with RMSprop (Momentum)	1.89×10^{-9}	-
SGD with RMSprop (No Momentum)	1.60×10^{-9}	-
Minimum Cost Sklearn (SGD - Adaptive)	0.0025	Adaptive
Minimum Cost Sklearn (SGD)	0.0029	0.01
Plain GD (No Momentum)	0.0169	0.6

Table 1: Sorted comparison of hyperparameter configurations that gives the lowest cost score for different GD methods with different activation functions. $Nmax = 500$

Figure 2 shows plot of two configurations as seen in 1. The scale of the y-axis is different for the two subfigures. The plain GD converges to the minimum faster, but the converged value is higher than the cost score of SGD with ADAM.



(a) Plot of lowest MSE score for plain GD as a function of iterations(Nmax = 500), see 1

(b) Plot of lowest MSE score for SGD with ADAM as a function of iterations(Nmax = 500), see 1

Figure 2: Plots of lowest MSE for different GD/SGD configurations

3.2 Other polynomial function - FFNN and OLS/Ridge Regression

Table 2 shows the lowest MSE scores for combinations of activation functions on the hidden and output layers and for OLS/Ridge regression. The combinations which are not part of this selection were not able to run due to NaN values or overflow issues. The combination of activation functions in the neural network has a significant impact on the performance, as seen from the diverse MSE scores. The combination with the lowest MSE score for my data is using sigmoid on the hidden layer and relu on the output layer. Using leakyrely on the hidden layer and sigmoid on the output layer also performed well.

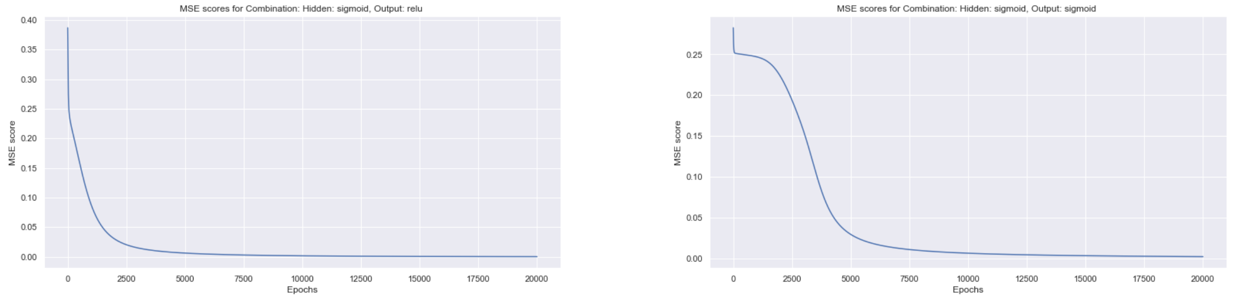
OLS and Ridge regression performed okay and performed the 3rd and 4th worst.It is unexpected that OLS performed worse better than ridge regression.

Method	Hidden Activation	Output Activation	Lowest MSE Score
FFNN	sigmoid	relu	0.0003
FFNN	sigmoid	sigmoid	0.0021
FFNN	relu	sigmoid	0.1204
FFNN	leaky_relu	sigmoid	0.0667
OLS	-	-	0.125
Ridge	-	-	0.1303
FFNN	relu	leaky_relu	0.2452
FFNN	leaky_relu	relu	0.5

Table 2: Sorted comparison of lowest MSE score for FFNN using different combinations of activation functions and OLS/Ridge Regression

Different activation functions have different properties, and how well they performed depend on the data. Also, it's possible that the combination with the lowest MSE benefited from a more appropriate set of hyperparameters.

Figure 3 shows two of the plots of MSE scores as a function of epochs from Table 2. It shows that using relu on the output layer more quickly converges compared to using sigmoid on the output layer for this problem.



(a) Plot of lowest MSE score FFNN with sigmoid and relu as activation functions as a function of epochs, see 1

(b) Plot of lowest MSE score FFNN with sigmoid as activation function as a function of epochs, see 1

Figure 3: FFNN - Plots of lowest MSE scores depending on the activation functions

3.3 Classification analysis - Breast Cancer dataset - FFNN

The test accuracy was calculated for combinations of hidden activation layer as the sigmoid function, output activation layer, hidden layers,learning rate and epochs.

Only the sigmoid function was used as the hidden layer as it gave the best results in previus exercises. Increasing the number of hidden layers had no effect on the test accuracy for certain configurations. A learning rate of 0.001 seems to perform the best for the different configurations. A learning rate equal to 0.1 performs the worst for most of the configurations. The best performing configurations have relu or leaky-relu on the output layer, a low learning rate and a significant number of epochs. Lower learning rates than 0.001 were not tested.

The best test accuracy of ≈ 0.96 were achieved with the combinations in Table 3:

Hidden Activation	Output Activation	Hidden Neurons	Learning Rate	Epochs	Test Accuracy
sigmoid	relu	5	0.001	1000	0.9649
sigmoid	relu	10	0.001	1000	0.9649
sigmoid	relu	15	0.001	1000	0.9649
sigmoid	leaky_relu	5	0.001	1000	0.9649
sigmoid	leaky_relu	15	0.001	1000	0.9649
sigmoid	leaky_relu	15	0.001	1500	0.9649

Table 3: FFNN - Best test accuracy as a function of hyperparameters and activation functions

The worst test accuracy of ≈ 0.35 were achieved with the combinations in Table 4:

Hidden Activation	Output Activation	Hidden Neurons	Learning Rate	Epochs	Test Accuracy
sigmoid	leaky_relu	10	0.1	1500	0.3596
sigmoid	relu	15	0.1	1500	0.3509
sigmoid	leaky_relu	10	0.01	1500	0.3509

Table 4: FFNN - Worst test accuracy as a function of hyperparameters and activation function

In total, the best test accuracy was with the MLPClassifier. This is not unexpected as the MLPClassifier is a powerful algorithm and when ran with a a configuration based on extensive experimentation with hyperparamters is to be expected to give great results. It's best test was ≈ 0.97 by using relu as the actication functioion as seen in Table 5 :

Hidden Layer Size	Activation Function	Learning Rate	Max Iterations	Test Accuracy
15	relu	0.001	500	0.9736842105

Table 5: FFNN - Best test accuracy with SKlearn's MLPClassifier as a function of hyperparameters and activation functions

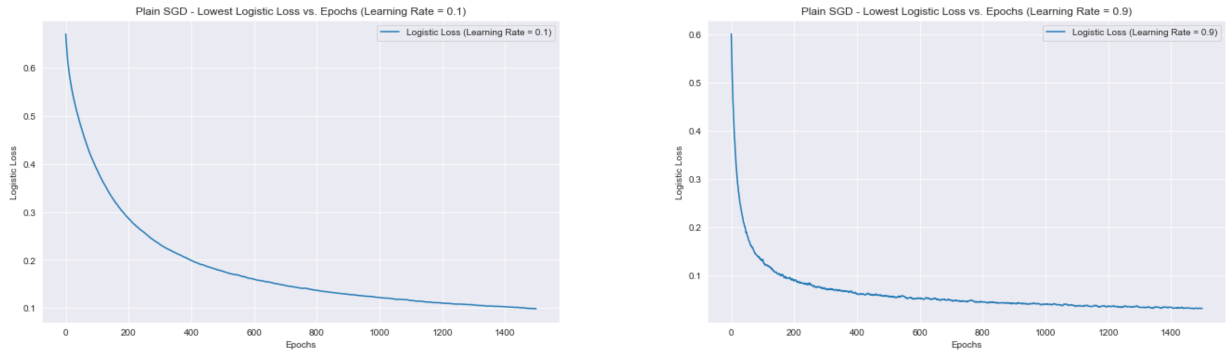
3.4 Simple classification analysis - FFNN and Logistic Regression

The logistic regression model was trained using Stochastic Gradient Descent (SGD) with varying learning rates, and gave the cost scores given in Table 6 . The lowest logistic loss score achieved with learning rates between 0.001 and 0.1 was 0.0986 (SGD, eta = 0.1).

Algorithm	**Lowest Cost Score**
scikit-learn Logistic Regression	0.287
Logistic Regression with SGD (eta = 0.1)	0.0986
Logistic Regression with SGD (eta = 0.9)	0.0023

Table 6: Lowest Logistic/cross entropy cost scores compared to Scikit's Logistic Regression

When learning rates were increased to the range of 0.01 to 0.9, the lowest logistic loss score reached was 0.0023 (SGD, eta = 0.9).Further increases in the learning rate led to issues with division by zero in the logistic loss function. See Figure 4 for plots of the logistic score as function of learning rate. The learning rate that gave the lowest cost score in the two instances are 0.1 and 0.9.



(a) Plots of lowest logistic loss score as a function of learning rate, see 6

(b) Plots of lowest logistic loss score as a function of learning rate, see 6

Figure 4: Plots of lowest logisitc loss for different learning rates

For the other algorithms, Logistic loss score from Scikit-learn's Logistic Regression is 0.287.The lowest logistic loss scores from the FFNN using a sigmoid activation function are provided in Table 7.

Hidden Layer	**Output Layer**	**N Layers**	η	**Epochs**	**Cost Func.**	**Cost Score**
sigmoid	sigmoid	10	0.01	1500	logistic_loss	0.0235
sigmoid	sigmoid	15	0.01	1500	logistic_loss	0.0247

Table 7: Lowest Logistic Loss Scores for FFNN as a function of hyperparameters and activation functions

The FFNN with sigmoid activation function, based on specific hyperparameters, achieved the lowest logistic loss scores. Logistic regression with SGD outperformed Scikit-learn's Logistic Regression, especially with optimized hyperparameters. Limited time was allocated to optimizing Scikit-learn's logistic regression function. Logistic loss scores for logsitic regression and from the FFNN were almost similar when with the same learning rate. The different algortihms in this project have different sensitivity to learning rates. When the learning rate was increased to 0.9 for logistic regression the lowest logistic loss score was significantly lower with logistic regression compared to FFNN.

4 Conclusions

In conclusion, the results and discussions presented in this report emphasize the significance of hyperparameter selection in the results given by the different algorithms. Different models have different sensitivities to learning rate. It was not always time to extensively experiment with the selection of hyperparameters for all algorithms.

For polynomial function optimization the lowest cost score was achieved with plain SGD with momentum. The use of automated differentiation with SGD and ADAM gave the third lowest score. Generally, momentum played a key role in reducing the cost score. SKlearn's SGD did not perform so well and that might be to insufficient time to optimize the algorithm.

Plain GD converged faster than SGD with ADAM as seen in Figure 2.

For the polynomial function optimization, the comparison between the FFNN and OLS/ridge regression highlighted the impact of activation functions on performance. Sigmoid on the hidden layer and relu on the output layer achieved the lowest MSE score for the given data. Unexpectedly, OLS performed better than ridge regression in this scenario.

For the classification analysis on the Wisconsin Breast Cancer dataset configurations with relu or leaky-relu on the output layer, a low learning rate, and a significant number of epochs achieved the best test accuracy.The FFNN with the best test accuracy of 96 %. SKlearn's MLPclassifier performed the best overall with a test accuracy of 97 %.

In the simple classification analysis, logistic regression with stochastic gradient descent (SGD) gave great results and outperformed Sklean's Logisitc regression. This might be because of insufficient parameter tuning. FFNN with sigmoid as activation function on the output and hidden layer performed the great with a cost score of 0.0235, given the same learning rate as logsitic regression. When the learning rate was increased to 0.9 for logistic regression the cost decreased further to 0.0023.

In summary, the findings underscore the importance of thoughtful hyperparameter tuning for effective optimization and modeling based on the problem. The FFNN, especially when configured with appropriate activation functions and hyperparameters, emerged as very powerful for both regression and classification problems.

References

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

[2] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, 2019. ISBN-13: 978-1492032649.

[3] Morten Hjorth-Jensen. *Applied Data Analysis and Machine Learning*. 2021. Jupyter Book.