Classification and Regression, from linear and logistic regression to neural networks

Qian J Oscar[1]

[1]University of Oslo, Problemveien 11, 0313 Oslo, Norway

Abstract

This project explores both regression and classification tasks by developing a custom Feed-Forward Neural Network (FFNN) alongside logistic regression and classic linear regression models. The FFNN model, utilizing a backpropagation algorithm, is applied to a second-order polynomial function for regression task, and the Wisconsin Breast Cancer dataset for classification task. This project builds on project 1 by integrating gradient descent (GD) and stochastic gradient descent (SGD) optimizations, with enhancements through methods like momentum, Adagrad, RMSprop, and Adam. Comparative analyses are conducted with Scikit-Learn's implementations, examining model performance across mean-squared error (MSE), R^2 scores, and accuracy metrics. Results highlight the strengths and limitations of each approach, particularly with respect to hyperparameter selection, model stability, and computational efficiency, contributing insights into optimal model choice for specific problem types.

Keywords

Feed-Forward Neural Network, Gradient Descent, Stochastic Gradient Descent, Hyperparameter Selection

1. Introduction

Classification and regression are core challenges in predictive modeling, with applications spanning finance, healthcare, and engineering. Recent advances in machine learning, especially neural networks, have introduced powerful tools for these tasks, yet traditional models like logistic regression and linear regression remain competitive in certain contexts. This project aims to bridge classic and modern approaches by developing an FFNN for both regression and classification, comparing its performance to traditional methods and established Scikit-Learn models. We studied various methods of optimization, implementing gradient descent (GD) and stochastic gradient descent (SGD) with enhancements such as momentum, Adagrad, RMSprop, and Adam, while analyzing their effects on model accuracy and efficiency.

Starting with regression, we revisited methods from project 1 to implement GD and SGD optimizations in the context of ordinary least squares (OLS) and Ridge regression. The classification section leverages logistic regression and neural networks to classify tumor characteristics in the Wisconsin Breast Cancer dataset, using accuracy scores to evaluate performance. Through a systematic comparison of these models and a review of related literature, we discuss critical considerations in selecting and tuning machine learning models, particularly the role of regularization, learning rates, and activation functions. This analysis ultimately offers insights into optimizing machine learning architectures for both accuracy and computational efficiency.

jianhenq@uio.no (Q. J. Oscar)
https://github.com/oscarqjh (Q. J. Oscar)

2. Theory

2.1. Ordinary Least Squares (OLS)

In the Ordinary least Squares (OLS), we want to estimate the parameters β of a linear model that minimizes the sum of squared differences between the observed values y_i and those predicted by the linear model. The model assumes the form:

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \epsilon$$

where $\boldsymbol{y}$ is the target vector, $\boldsymbol{X}$ is the design matrix of predictors, $\boldsymbol{\beta}$ is the vector of coefficients, and ϵ represents the error terms. In OLS, we aim to minimize the cost function, which is also known as the Mean Squared Error (MSE), defined as:

$$C(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y_i})^2$$

or in a more compact matrix-vector notation as

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

From the cost function, we can quickly formulate the optimization problem as

$$\min_{\beta\in\mathbb{R}^p} \frac{1}{n}\{(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})\}$$

With some calculus and linear algebra, we will arrive at an analytical solution for the parameter β. This solution is also implied to be unique [1].

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}^T} = 0 = \boldsymbol{X}^T\left(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\right),$$

$$\boldsymbol{X}^T\boldsymbol{y} = \boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\beta},$$

$$\boldsymbol{\beta} = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}.$$

We will realise that β will depend on finding the inverse of $\boldsymbol{X^T X}$. This is a computationally expensive procedure with a time complexity of $O(nd^2 + d^3)$, where n is the number of data and d is the number of features [1]. Hence, in this project, we explore using iterative optimization techniques, such as GD as an alternative, which we will go into more details in the Gradient Methods section.

2.2. Ridge Regression

Ridge regression, originally developed to address issues of collinearity, has gained renewed relevance with the rise of high-dimensional data. In high-dimensional settings, the covariates in the design matrix $\boldsymbol{X}$ are highly collinear—meaning they exhibit strong linear relationships. This collinearity causes the covariates to span a subspace of reduced dimensionality within the

overall parameter space, leading to a nearly rank-deficient design matrix. As a result, it becomes challenging to distinguish the individual contributions of these covariates to the response variable Y. This ambiguity regarding the specific covariates driving the variation in Y influences the linear regression model's fit to the data. Collinearity presents itself in the model fit through large estimation errors for the regression coefficients of collinear covariates, which often result in inflated coefficient values [2]. We define the Ridge regression cost function as:

$$C(\boldsymbol{X}, \boldsymbol{\beta}) = \{(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})\} + \lambda \boldsymbol{\beta}^T\boldsymbol{\beta},$$

where λ is the regularization parameter which controls the amount of shrinkage applied to the coefficients. From the cost function, we can then derive the analytical solution as:

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X^T X} + \lambda I)^{-1}\boldsymbol{X^T y}$$

We can see that similar to OLS, derivation of the analytical solution requires matrix inversion, which can be computationally expensive for high dimensional and large datasets.

2.3. Logistic Regression

Logistic Regression is a classification algorithm used when the target variable is binary. Unlike linear regression, logistic regression models the probability that a given instance belongs to a particular class. The logistic sigmoid function constrains the output between 0 and 1 and is defined as:

$$logit(p) = log(\frac{p}{1-p})$$

This can be rewritten as:

$$\phi(z) = \frac{1}{1+e^{-z}}$$

where z is the linear combination of weights and the inputs, defined as:

$$z = \boldsymbol{w^T x}$$

We then aim to maximise the probability of seeing the observed data. For this purpose, we define the cost function, also known as cross entropy as:

$$\mathcal{C}(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i) - \log\left(1 + \exp\left(\beta_0 + \beta_1 x_i\right)\right)\right) \tag{2}$$

In order for us to use this cost function for gradient methods for optimisation, we have to add a regularisation term. This is to prevent the weights from exploding to infinity [3]. For this project, we will use the L2 regularisation where we add the L2 regularisation term to equation (2), hence the new cost function is defined as:

$$\mathcal{C}(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i) - \log\left(1 + \exp\left(\beta_0 + \beta_1 x_i\right)\right)\right) + \lambda \sum_{j=1}^{m} (\beta_1)^2$$

To minimise the cost function using gradient methods, we will find its first derivative:

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\boldsymbol{X}^T (\boldsymbol{y} - \boldsymbol{p}) + \lambda \cdot \beta$$

We can also generalise this to a multiclass setting, which is known as multinomial logistic regression [4].

2.4. Neural Networks (NN)

In this project, I built a feed-forward neural network (FFNN) from scratch. FFNN was the first and the simplest type of artificial neural network (ANN) that were made. It is characterised by its one-directional information flow and full-connectivity of each node in any layer.

2.4.1. Perceptron Model

The perception model is considered one of the best type of ANN and has since influenced the development of many other NNs. The perceptron model starts by multiplying each input by its corresponding weight, then sums these weighted inputs to produce a total weighted sum. This sum is then passed through an activation function, $\boldsymbol{f}$, often referred to as the step function, to generate the final output.

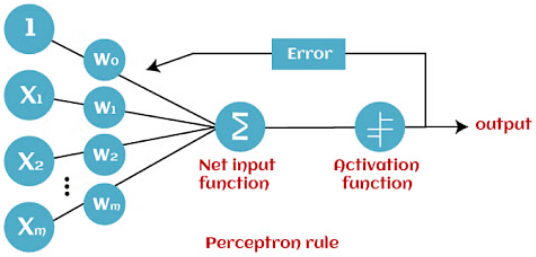

Figure 1: Illustration of a single perception model

(https://www.simplilearn.com/tutorials/deep-learning-tutorial/perceptron#:~:text=Single%20Layer%20Perceptron%20model%3A%20One,separable%20objects%20with%20binary%20outcomes.)

The figure above shows the architecture of a single perception model. The FFNN developed in this project is a much higher level derivation of the perception model, where instead of a single node, FFNN utilise multiple layers, each with multiple nodes.

2.4.2. Activation Functions (AFs)

Activation functions have to be non-constant, bounded, monotonically-increasing and continuous to fulfill the universal approximation theorem. It is indeed a very important area of research for NNs and has been studied extensively [5]. In this project, we will only explore a subset of activation functions.

Sigmoid

The sigmoid function $\sigma : \mathbb{R} \to [0,1]$ maps any real-valued number to a value between 0 and 1. It is often used as an activation function in the output layer of a binary classification neural network and is defined as:

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

The first derivative of sigmoid is defined as:

$$\sigma^{'}(x) = \sigma(x) \cdot (1 - \sigma(x))$$

Softmax

The softmax function converts a vector of K real values into another vector of K real values that add up to 1. Regardless of whether the input values are positive, negative, zero, or greater than one, the softmax function transforms them into values ranging between 0 and 1, making them interpretable as probabilities [6]. It is defined as:

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

where, z_i represents the input to the softmax function for class i, and the denominator is the sum of the exponentials of all the raw class scores in the output layer. We almost always uses the Softmax function together with the Cross Entropy Loss function in the output layer of the NN for classification task. Hence, the derivative of Softmax is often calculated together with the cross entropy loss function. Hence, we simply pass the derivative of the cost function down instead [7].

Rectified Linear Units (ReLU)

The Rectified Linear Unit (ReLU) function maps any negative input to 0 and any positive input to itself. It is often used as an activation function in the hidden layers of a neural network. It is defined as:

$$ReLU(x) = \begin{cases} x & : x \geq 0 \\ 0 & : x < 0 \end{cases}$$

The first derivative of ReLU is defined as:

$$ReLU^{'}(x) = \begin{cases} 1 & : x \geq 0 \\ 0 & : x < 0 \end{cases}$$

Leaky ReLU

Vanishing gradient is the main problem with ReLU AF which is caused due to the non-utilization of negative values. A Leaky Rectified Linear Unit (LReLU) is the extension of ReLU by utilizing the negative values [8]. The LReLU is defined as:

$$LReLU(x) = \begin{cases} x & : x \geq 0 \\ 0.01x & : x < 0 \end{cases}$$

Table 1
Advantage and disadvantage of primary AFs

AFs	Diminishing gradients	Computational inefficiency
sigmoid	Yes	Yes
softmax	Yes	Yes
ReLU	Partial	No
LReLU	No	No

The first derivative of LReLU is defined as:

$$LReLU^{'}(x) = \begin{cases} x & : x \geq 0 \\ 0.01x & : x < 0 \end{cases}$$

The selection of AF is crucial to the performance of the NN, in some cases, a unsuitable AF can lead to non-convergence of the model. Table 1 shows some comparison of the different AFs explored in this project.

2.4.3. Feed-forward

In the forward pass, each layer is defined by number of nodes and its AF. The input, which can be either a vector or matrix depending on whether batch input is implemented, is passed to the AF. Then for subsequent layers, the output of the previous layer will be used as the input of that layer. Output of a layer can be defined as:

$$z^l = (W^l)^T a^{l-1} + b^l$$

where W is the weights of layer l, a is the output from AF of layer $l-1$ and b is the bias of layer l.

2.4.4. Back-propagation

The back-propagation is the main mechanism to how the NN "learn". It is basically a gradient method to optimize the cost function based on the derivative of the cost function given a certain learning rate.

The cost function which we want to optimise in this project is given by equation (1) - MSE - for regression tasks, and equation (2) - Cross Entropy Loss - for classification tasks.

For each layer, we want to optimise the cost function with respect to the weights, I will then utilise the chain rule have to find

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w}$$

where $\frac{\partial C}{\partial a}$ is given by $\frac{\partial C}{\partial z} \cdot W_{l-1}^T$ for non-output layers, and $\frac{\partial z}{\partial w}$ is just the layer input. For simplicity, we will define δ as,

$$\delta = \frac{\partial C}{\partial a}\frac{\partial a}{\partial z}$$

After obtaining the gradient of cost function for a layer, I will then update the weights and biases according to the gradient method that was chosen. The simplest example for how the weight and biases are updated is

$$W \leftarrow W - lr \cdot \frac{\partial C}{\partial w}$$

$$b \leftarrow b - lr \cdot \delta$$

where W and b is the weights and bias respectively.

2.5. Gradient Methods

Gradient methods are a class of optimisation algorithms that is used to minimize the loss function by iteratively adjusting model parameters in the direction of the negative gradient [9]. In this project, I explored multiple different gradient methods in the for both linear regression and NN.

2.5.1. Gradient Descent (GD)

GD is the simplest gradient method. It is a first-order iterative optimisation algorithm where each step updates parameter in the direction of the steepest descent. The update rule is simply:

$$\theta_{i+1} = \theta_i - \eta \cdot \nabla C(\theta_i)$$

where θ is the parameter you want to optimise, η is the learning rate.

2.5.2. Stochastic Gradient Descent (SGD)

An easy and effective improvement is to approximate the gradient using a subset of the training data, known as a minibatch, rather than computing the gradient over the entire dataset.

In SGD, instead of using a fixed learning rate for every iteration, we add a time decay rate to the learning rate,

$$\gamma(t;, t_0, t_1) = \frac{t_0}{t + t_1}$$

where t_0 and t_1 are new parameter to tune, and t is the current epoch. We can then define the update rule for SGD as:

$$\theta_{t+1} = \theta_t - \gamma(t) \cdot \nabla C(\theta_t)$$

where t is the current epoch.

This adds stochasticity which decreases the chance of the algorithm getting stuck in a local minima. In this project, SGD will refer to stochastic gradient descent with mini-batches.

2.5.3. Adaptive Gradient Descent (AdaGrad)

The goal of AdaGrad is to minimize the expected value of a stochastic objective function concerning a set of parameters, based on a sequence of function realizations. Similar to other methods that rely on sub-gradients, AdaGrad updates the parameters in the opposite direction of the sub-gradients. However, unlike standard sub-gradient methods that apply fixed step sizes without considering past observations, AdaGrad customizes the learning rate for each parameter individually by utilizing the sequence of gradient estimates [10]. In AdaGrad, we introduce G_t, the outer product of all previous subgradients in addition to the learning rate, given by:

$$G_{t+1} = G_t + \nabla C(\theta_t)^2$$

We can then define the update rule of AdaGrad as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t}} \cdot \nabla C(\theta_t)$$

2.5.4. Root Mean Squared Propagation (RMSProp)

RMSprop is an adaptive learning rate algorithms. Adagrad adjusts the learning rate by scaling each gradient element-wise based on a running sum of squared gradients for each parameter, helping it adapt to directions with different gradient magnitudes. When dealing with high condition numbers, this scaling helps accelerate movement in directions with small gradients and slows down in directions with large gradients by dividing by a larger value.

As training progresses, Adagrad's steps continuously shrink due to the growing accumulation of squared gradients, making it suitable for convex optimization where slowing down near a minimum is beneficial. However, in non-convex optimization, this can lead to issues like getting stuck at saddle points. RMSprop addresses this by maintaining an estimate of squared gradients but using a moving average instead of an accumulating sum, allowing for more consistent updates throughout training [11]. For RMSProp, we define G_t as:

$$G_{t+1} = \rho \cdot G_t + (1 - \rho)(\nabla C(\theta_t))^2$$

Here, we see that ρ is a new tunable parameter. We can then define the update rule of RMSProp as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t}} \cdot \nabla C(\theta_t)$$

2.5.5. Adaptive Moment Estimation (Adam)

Adam is a first-order optimization method specifically designed for stochastic objectives in high-dimensional parameter spaces, where higher-order methods are impractical. Adam calculates adaptive learning rates for each parameter using estimates of the gradients' first and second moments, drawing from AdaGrad's strength in handling sparse gradients and RMSProp's suitability for non-stationary and online settings. Adam's advantages include gradient-rescaling invariance, approximate control over step sizes, compatibility with non-stationary objectives,

support for sparse gradients, and built-in step size annealing [12]. For Adam, we will have to define the first moment vector, m_t, and the second moment vector, v_t,

$$m_{t+1} = \beta_1 \cdot m_t + (1 - \beta_1) \cdot \nabla C(\theta_t)$$

$$v_{t+1} = \beta_2 \cdot v_t + (1 - \beta_2) \cdot \nabla C(\theta_t)$$

where β_1 and β_2 are new tunable parameters, with a recommended default value to 0.9 and 0.999 respectively. We will also need to compute the bias-corrected first moment estimate, $\hat{m}_t$, and the bias-corrected second raw moment estimate, $\hat{v}_t$,

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

With this, we can then define the Adam update rule as:

$$\theta_{t+1} = \theta_t - \frac{\eta \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where ϵ is a very small value to prevent zero-division error.

2.5.6. Momentum

The problem with gradient descent is that the weight update at moment t is governed by the learning rate and gradient at that moment only. It does not take into account the past steps taken while traversing the cost space. By applying momentum to gradient descent methods, we can allow the algorithm to prevent itself getting stuck in a saddle point [13]. We can define the velocity, v_t, as:

$$v_t = \gamma \cdot v_{t-1} + updaterule$$

where γ is the momentum parameter. With this, we can define the update rule as:

$$\theta_{t+1} = \theta_t - v_t$$

We note that momentum can be added to any of the gradient methods above.

3. Method

3.1. Dataset

3.1.1. Second-order Polynomial Function

For regression tasks, I generated my own data from a second-order polynomial function,

$$f(x) = 4 + 3x + 2x^2 \tag{3}$$

where $x \in [0, 2]$. Figure 2 shows a sample of the dataset.

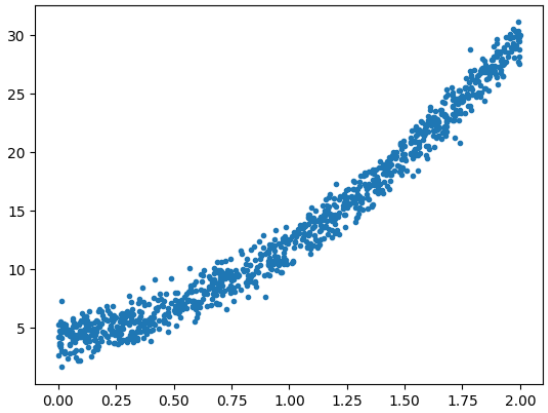

Figure 2: Dataset for Regression Task

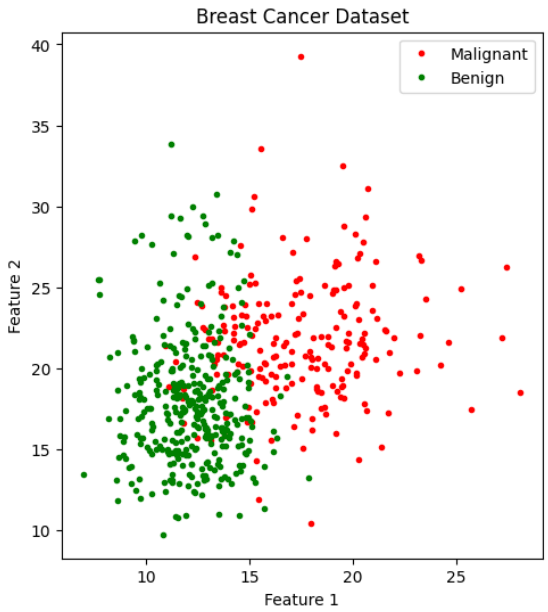

Figure 3: Scatterplot of Feature 1 & 2 grouped by benign or malignant

3.1.2. Wisconsin Breast Cancer Dataset

This dataset presents 569 samples, each with 30 features and target label of 2 classes, 'Benign' or 'Malignant'.

Figure 3 is a simple visualisation of the dataset with just feature 1 and 2. We can clearly see that there are clusters for Malignant and Benign cases, which can be classified using machine learning techniques. In the dataset, there are 30 of these features which we can use to train our model for classification.

3.2. Train Test Split

I only did a train-test split for the Wisconsin Breast Cancer Dataset since the number of samples is limited, hence I have to hold out a portion of the data for testing.

As for the second-degree polynomial function, I trained all models on one set of data generated initially. Afterward, I simply generated new data on the fly using the same function when testing the trained model.

3.3. Normalisation

Following the train-test split, I normalized the feature values to ensure that all input variables are on a comparable scale. Normalization is particularly important for optimizing the performance of gradient-based algorithms, as it helps in achieving faster convergence and preventing any single feature from disproportionately influencing the model's weights. This preprocessing step also reduces the impact of scale differences, improving the robustness of the model to various input ranges.

The scaling is done with the help of Scikit learn's **StandardScalar** class.

3.4. Model Evaluation

3.4.1. Metric

For regression task, I evaluated all models with MSE,

$$C(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y_i})^2$$

And for classification task, I evaluated all models with a simple Accuracy defined as:

$$Accuracy = \frac{Number of Correct Predictions}{Total Number of Predictions}$$

3.4.2. Convergence

Throughout the project, I compared the models' convergence by examining the number of iterations required. This was achieved by tracking changes in the loss function or accuracy across epochs through a graph like Figure 4.

For further analysis, I define a model as *converged* if and only if any metric score θ_i at the i-th epoch, the following condition holds: $|\theta_i - \theta_{i+10}| \leq \delta$, where δ is a convergence threshold set to a default value of 0.01. This implies that the metric score remains within a small tolerance δ over a span of 10 epochs, indicating stabilization.

It should also be noted that convergence does not imply optimality. That is to say that a model can possibly converge to a local minima, but not the global minima.

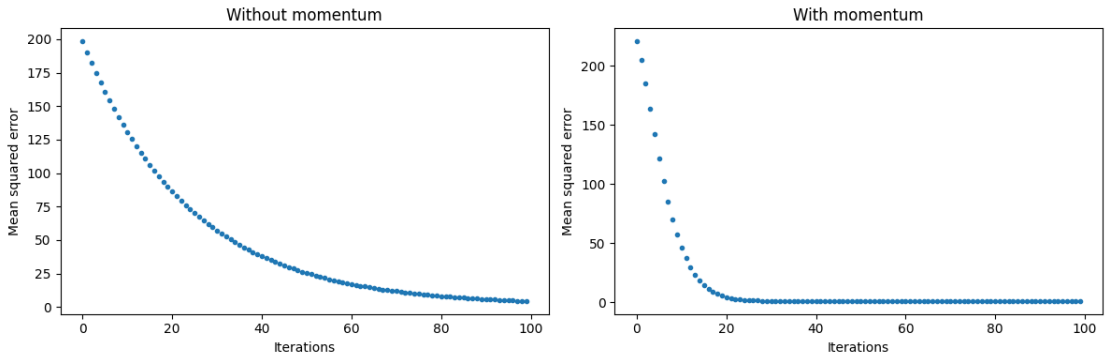

Figure 4: Example of MSE-epoch graph to observe convergence of a model

4. Implementation

All Regression and Neural Network models are implemented in Object-Oriented Programming (OOP) classes. This is to ease the reusability of the code. All code can be found on the Github repository, and specifically, Regression models can be found in ols_ridge.py, NN models can be found in nn.py, Logistic Regression model can be found in logistic.py, and finally, all other utility functions such as functions for sigmoid or mse can be found in utils.py.

4.1. Implementation of Regression Methods with Gradient Methods

Both the OLS and Ridge models can be initialized with the **LinearRegression** class and by specifying the *reg_method* as "ols" or "ridge".

For gradient methods, I implemented them separate classes which needs to be initialized and be fed to the regression model. I believe that this architecture allows more gradient methods to be easily added in the future. The following are the optimisers implemented for regression models in this project

- GradientDescentOptimiser
- SGDOptimiser
- AdagradOptimiser
- RMSPropOptimiser
- AdamOptimiser

Note that in this project, I will refer to Stochastic Gradient Descent (SGD) as Minibatch Gradient Descent with a default *batch_size* of 32. All the above optimisers except for GradientDescentOptimiser have a *batch_size* parameter, and can be generalised to a plain GD optimiser by setting the *batch_size* to be equal to the number of sample data. Below shows an example of how to set up the model,

```
from ols_ridge import LinearRegression, SGDOptimiser

# Initialise optimiser
```

```
optimizer = SGDOptimiser( # see source for more arguments
    learning_rate=0.01,
    batch_size=32         # setting batch_size to sample size generalise this to plain GD
)
model = LinearRegression(
    reg_method="ols",     # or can be "ridge"
    optimizer=optimizer   # if no optimizer is provided, the model will perform fitting by
    matrix inversion
)
```

Listing 1: Linear regression example

4.2. Implementation of Neural Networks

The Neural Network class can be initialised with **NeuralNetwork** class. It will require a few parameters, namely:

- number of input features
- a list of layer sizes
- a list of activations function (AF) corresponding to each layer
- a list of derivatives corresponding to each AF in the list
- cost function
- derivative of the cost function
- optimiser

You can see that I adopted a similar architecture as how I designed the classes for Regression models, separating the model and optimiser class to allow for more flexibility in choosing which gradient method to use. The following optimisers are implemented for neural network in this project:

- NNBasicOptimiser
- NNSGDOptimiser
- NNAdagradOptimiser
- NNRMSpropOptimiser
- NNAdamOptimiser

All the above optimisers except NNBasicOptimiser can be generalised to a plain GD optimiser by setting the *batch_size* parameter to be equal to the number of sample data used for fitting. Below shows an example of how to set up a neural network,

```
from nn import NeuralNetwork, NNBasicOptimiser

network_input_size = 1
layer_output_sizes = [32, 32, 1]
activation_functions = [sigmoid, sigmoid, leaky_ReLU]
activation_derivatives = [sigmoid_derivative, sigmoid_derivative, leaky_ReLU_der]
cost_function = mse
cost_derivative = mse_derivative
optimizer = NNBasicOptimizer(learning_rate=0.01)
```

```

# create the neural network
nn = NeuralNetwork(
    network_input_size=network_input_size,
    layer_output_sizes=layer_output_sizes,
    activation_functions=activation_functions,
    activation_derivatives=activation_derivatives,
    cost_function=cost_function,
    cost_derivative=cost_derivative,
    optimizer=optimizer,
    debug=False
)
```

Listing 2: Neural network example

4.3. Implementation of Logistic Regression

The Logistic Regression model can be initialized with the **LogisticRegression** class. It's implementation follows exactly as described in section 2.3 where an L2 regularisation term is included.

4.4. Benchmarking

In this section, I will benchmark my own implementation against external libraries such as PyTorch's implementation.

4.4.1. OLS & Ridge

Since PyTorch's implementation of OLS and ridge regression is the same, I will test the two model at the same time. I compared my own ols, ridge, and pytorch linear model using different gradient methods. To minimize errors, I ran all tests for 7 runs, and 10 loops each.

Table 2
This table shows the run time to fit the model at epoch for different implementations. All test is run with learning rate=0.01, batch_size=32, degree=2, epochs=1000

gradient method ↓	implementation→	Own OLS	Own Ridge	PyTorch
GD		25.4 ms ± 1.5 ms	24.7 ms ± 1.83 ms	183 ms ± 16 ms
SGD		418 ms ± 16 ms	439 ms ± 5.97 ms	237 ms ± 5.44 ms
Adagrad		457 ms ± 17.9 ms	446 ms ± 13.1 ms	248 ms ± 9.61 ms
RMSprop		590 ms ± 8.31 ms	547 ms ± 25 ms	228 ms ± 6.32 ms
Adam		738 ms ± 23.4 ms	789 ms ± 25.7 ms	368 ms ± 11.2 ms

From Tables 2 and 3, it's evident that my implementations of both OLS and Ridge are generally less computationally efficient than PyTorch's, often exhibiting nearly twice the runtime. However, an interesting exception is my Gradient Descent (GD) implementation, which actually outperforms PyTorch's LinearModel. Overall, while my implementations are less optimized, they still perform adequately for the testing purposes of this project.

Table 3
This table shows the run time to fit the model at epoch for different implementations. This tests are ran with the same settings as Table 2 but with epochs=5000

gradient method ↓ implementation→	Own OLS	Own Ridge	PyTorch
GD	123 ms ± 6.32 ms	122 ms ± 9.03 ms	898 ms ± 11 ms
SGD	2.02 s ± 31.7 ms	1.95 s ± 102 ms	1.78 s ± 1.04 s
Adagrad	2.07 s ± 36.1 ms	2.14 s ± 61.2 ms	1.14 s ± 54.5 ms
RMSprop	2.75 s ± 74.6 ms	2.84 s ± 64.9 ms	1.18 s ± 56.7 ms
Adam	3.7 s ± 78.4 ms	3.84 s ± 85.2 ms	1.97 s ± 56.4 ms

Table 4
This table shows the run time to fit the model at different degree for different implementations. This tests are run with the same settings as Table 2 but with varying degrees.

gradient method ↓ degree→	3	4	5
torch GD	188 ms ± 10.3 ms	174 ms ± 7.08 ms	175 ms ± 8.59 ms
own ols GD	25.4 ms ± 1.1 ms	30.2 ms ± 1.55 ms	30.2 ms ± 2.7 ms
own ridge GD	28.5 ms ± 2.79 ms	28.6 ms ± 2.64 ms	30.2 ms ± 1.67 ms
torch SGD	279 ms ± 29.4 ms	231 ms ± 5.98 ms	249 ms ± 11.2 ms
own ols SGD	413 ms ± 10.8 ms	410 ms ± 8.9 ms	422 ms ± 15.5 ms
own ridge SGD	416 ms ± 6.38 ms	427 ms ± 13 ms	419 ms ± 10.2 ms
torch Adagrad	241 ms ± 4.19 ms	233 ms ± 7.07 ms	235 ms ± 5.61 ms
own ols Adagrad	460 ms ± 8.45 ms	447 ms ± 16.7 ms	451 ms ± 11.4 ms
own ridge Adagrad	448 ms ± 13.1 ms	451 ms ± 13.6 ms	461 ms ± 10.5 ms
torch RMSprop	238 ms ± 4.93 ms	241 ms ± 5.35 ms	245 ms ± 7.59 ms
own ols RMSprop	547 ms ± 12.1 ms	569 ms ± 11.2 ms	578 ms ± 15.2 ms
own ridge RMSprop	566 ms ± 12.9 ms	586 ms ± 15.8 ms	606 ms ± 11 ms
torch Adam	376 ms ± 6.39 ms	380 ms ± 7.29 ms	400 ms ± 15.1 ms
own ols Adam	740 ms ± 21.7 ms	731 ms ± 13.6 ms	742 ms ± 20.4 ms
own ridge Adam	743 ms ± 14.3 ms	752 ms ± 20.7 ms	746 ms ± 31.7 ms

Table 4 demonstrates that increasing the dimensions of the design matrix has minimal impact on runtime. However, it's noteworthy that the variance in runtime does increase with higher polynomial degrees. It's important to mention that this behavior has only been tested up to a polynomial degree of 5.

4.5. Neural Network

In this section, I will compare the performance of my own Neural Network and PyTorch's implementation.

Tables 5 and 6 highlight that my custom implementation of a neural network is significantly less efficient than PyTorch's, with performance discrepancies by large margins. Table 6, in particular, illustrates PyTorch's strong scalability, with its runtime increasing by only around 20 ms per additional hidden layer. In contrast, my implementation shows a much steeper runtime increase—over 100 ms for each added hidden layer.

Table 5
This table shows the run time to fit the model at epoch for different implementations. This tests are ran with learning rate=0.01, 2 hidden layers each with 16 neurons with a ReLU activation after each hidden layer, the output layer have a single neuron followed by a Leaky ReLU activation, MSE as the loss function, and epochs=100.

gradient method ↓ implementation→	Own NN Code	PyTorch NN
GD	128 ms ± 4.94 ms	95.5 ms ± 3.95 ms
SGD	520 ms ± 6.84 ms	105 ms ± 6.14 ms
Adagrad	599 ms ± 21 ms	126 ms ± 1.9 ms
RMSprop	619 ms ± 5.87 ms	123 ms ± 10.3 ms
Adam	711 ms ± 5.42 ms	131 ms ± 6.02 ms

Table 6
This table shows the run time to fit the model with different number of hidden layers for different implementations. This tests are run with the same settings as Table 5 but with different number of hidden layer. Each hidden layer has 16 neurons followed by a sigmoid activation.

gradient method ↓ no. of hidden layers→	3	4	5
torch NN GD	124 ms ± 6.67 ms	182 ms ± 27.8 ms	165 ms ± 10.7 ms
own NN GD	187 ms ± 5.97 ms	281 ms ± 67.1 ms	316 ms ± 20.9 ms
torch NN SGD	133 ms ± 6.85 ms	160 ms ± 5.15 ms	162 ms ± 9.58 ms
own NN SGD	684 ms ± 4.08 ms	879 ms ± 9.68 ms	1.04 s ± 6.54 ms
torch NN Adagrad	153 ms ± 8.55 ms	182 ms ± 7.25 ms	197 ms ± 18 ms
own NN Adagrad	785 ms ± 5.54 ms	947 ms ± 16.7 ms	1.16 s ± 6.15 ms
torch NN RMSprop	157 ms ± 5.51 ms	192 ms ± 8.33 ms	194 ms ± 13.4 ms
own NN RMSprop	828 ms ± 9.48 ms	969 ms ± 11.2 ms	1.23 s ± 4.95 ms
torch NN Adam	179 ms ± 6.73 ms	207 ms ± 10.5 ms	221 ms ± 14.9 ms
own NN Adam	933 ms ± 3.96 ms	1.19 s ± 8.16 ms	1.43 s ± 13.7 ms

4.6. Logistic Regression

In this section, I will compare the performance of my own Neural Network and PyTorch's implementation.

Table 7
This table shows the run time to fit the model at different epoch for different implementations. This tests are ran with learning rate=0.01.

epochs ↓ implementation→	Own logistic Code	PyTorch logistic
100	41.5 ms ± 2.06 ms	28.1 ms ± 1.95 ms
200	80.5 ms ± 3.34 ms	50.2 ms ± 2.1 ms
300	115 ms ± 2.46 ms	74 ms ± 2.37 ms
400	154 ms ± 2.38 ms	99.6 ms ± 1.71 ms
500	193 ms ± 2.58 ms	125 ms ± 1.78 ms

From Table 7, we can also see that pytorch's implementation of logistic regression is much

more performant than mine.

It's worth noting that all tests in this section were conducted on my personal laptop, which has limited computational resources. This hardware constraint may impact the consistency and reliability of the runtime results. All code for PyTorch's implementation can be found in *torch_models.py*.

5. Results & Discussion

5.1. Overview of Algorithms and Their Applications

In this project, several machine learning algorithms were implemented to address both regression and classification tasks, enabling a broad exploration of model performance and efficiency. The primary algorithms tested include Ordinary Least Squares (OLS) regression, Ridge regression, logistic regression, and a Feed-Forward Neural Network (FFNN). Each algorithm was applied to tasks best suited to its inherent strengths, and the performance was examined in various contexts, including comparisons between custom implementations and library-based solutions.

OLS and Ridge Regression: These algorithms, focused on regression, provided a balance between interpretability and simplicity. Ridge regression, with regularization, improved generalization, especially in high-dimensional data settings.

Logistic Regression: Used for binary classification, logistic regression served as a reliable and interpretable baseline, particularly effective on structured datasets.

Feed-Forward Neural Network (FFNN): The FFNN was tested with optimizers like Gradient Descent, RMSprop, and Adam, learning non-linear patterns in both regression and classification tasks. While flexible and powerful, it had higher computational demands than simpler models.

5.2. Comparison of Performance Metrics

In this section, we compare the models' performance based on key metrics relevant to regression and classification tasks. These metrics provide a quantitative view of each algorithm's strengths and limitations, particularly in accuracy, computational efficiency, convergence rates, and robustness across different optimizers.

5.2.1. Regression Performance (OLS and Ridge Regression)

For regression tasks, performance was evaluated using Mean Squared Error (MSE) as the primary metric. OLS and Ridge regression showed strong performance on simpler datasets.

Effect of Learning Rate on OLS and Ridge

The learning rate significantly influences a model's training efficiency. As shown in Figure 5, smaller learning rates result in slower convergence, meaning the models require more epochs to reach optimal performance. This effect highlights the balance required in selecting an appropriate learning rate: while too high a rate risks overshooting minima, too low a rate can prolong training unnecessarily.

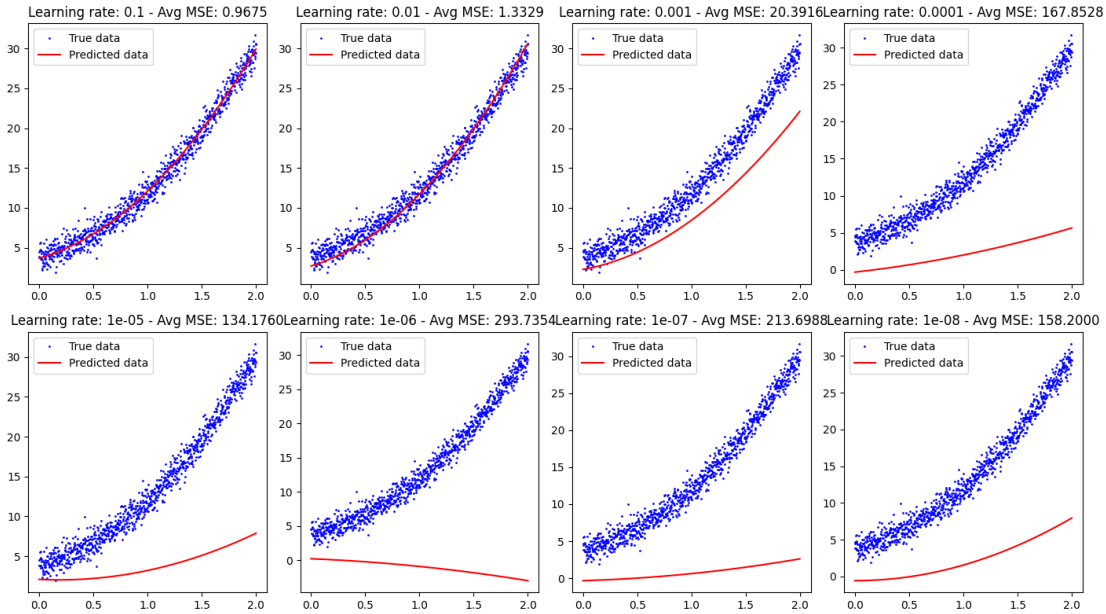

Figure 5: Visualisation of OLS /w Plain GD fitting at epoch=1000 at different learning rates

Table 8
This table shows the MSE values at different learning rates for OLS and Ridge /w Plain GD after 100 epochs of training.

learning rate ↓	model→	OLS	Ridge
1e-1		0.9675	1.05
1e-2		1.3329	1.3138
1e-3		20.392	22.291
1e-4		167.85	168.786
1e-5		134.18	134.264
1e-6		293.74	293.750
1e-7		213.7	213.7
1e-8		158.2	158.2

From Figure 5 and Table 8, we can see that a learning rate of 0.1 or 0.01 is an acceptable learning rate to use. See Appendix A for ridge regression's result.

Effect of Lambda, λ on Ridge

From Table 9, we can see that lambda has minimal effect on the MSE score of the model. This is perhaps because the data is easy to fit too, hence the regularisation effect is not so pronounced. See Appendix A for visualisation of the result.

Table 9
This table shows the MSE values at different learning rates for Ridge /w Plain GD after 100 epochs of training.

lambda, λ	MSE
1e-1	1.0501
1e-2	0.9813
1e-3	0.9739
1e-4	1.0059
1e-5	0.9805
1e-6	0.9721
1e-7	0.9807
1e-8	0.9707

Effect of adding Momentum to Gradient Methods

Momentum is a powerful mechanism that helps optimize the learning rate and prevents the algorithm from getting trapped in local minima. By carrying forward a fraction of the previous update, momentum smooths the path toward convergence, accelerating training and aiding in overcoming shallow local minima. Figure 6 is a good example showing how momentum can speed up the convergence of the model. See Appendix A for Ridge's result.

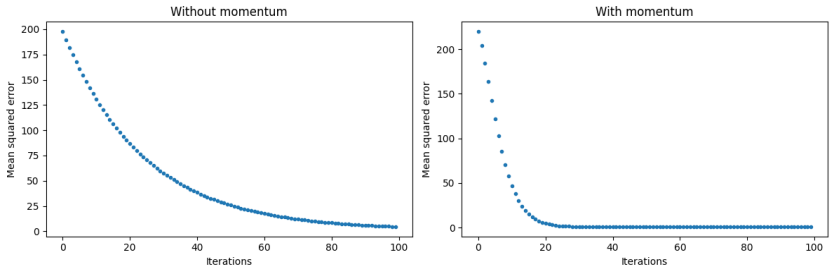

Figure 6: Convergence of OLS /w Plain GD /w and w/o momentum

Effect of Different Gradient Methods on Convergence

Gradient methods are techniques used to adjust the learning rate dynamically during training. When evaluating these methods, focusing on the learning rate itself is less informative, as the algorithm adapts it throughout the process. Instead, a more meaningful analysis revolves around the model's convergence, as it provides a clearer picture of the algorithm's effectiveness in reaching optimal performance.

See Section 3.4.2 on how *convergence* is defined in this project.

Table 10
Comparison of convergence of all gradient methods with and without momentum. *DNC means Did Not Converge.

method ↓	epoch to convergence	final MSE after 1000 epochs
Plain GD w/o m	500	1.0578
Plain GD w/ m	227	0.9334
SGD w/o m	9	0.9417
SGD w/ m	10	0.9347
AdaGrad w/o minibatch w/o m	DNC*	123.9374
AdaGrad w/o minibatch /w m	DNC	160.3874
AdaGrad w/ minibatch w/o m	490	0.9335
AdaGrad w/ minibatch /w m	261	0.9335
RMSProp w/o minibatch w/o m	DNC	3.0266
RMSProp w/o minibatch /w m	660	0.9341
RMSProp w/ minibatch w/o m	152	0.9342
RMSProp w/ minibatch /w m	85	0.9335
Adam w/o minibatch w/o m	DNC	95.3424
Adam w/o minibatch /w m	DNC	28.6623
Adam w/ minibatch w/o m	59	0.9338
Adam w/ minibatch /w m	29	0.9371

As shown in Table 10, SGD and Adam are the most efficient gradient methods, achieving the fastest convergence. In contrast, methods without mini-batches failed to converge within 1000 epochs, highlighting the significant advantage of using mini-batch training. For a detailed comparison of the convergence behavior of all gradient methods, refer to the visualization in Appendix A.

5.2.2. Feed-Forward Neural Network (Regression Task):

Neural networks are highly effective for modeling complex relationships, but they come with increased computational demands. For this analysis, I focused on a comparison with OLS methods, as previous results indicated minimal performance differences between Ridge and OLS for this dataset.

Comparison with OLS

Figure 7 clearly illustrates both the strengths and limitations of neural networks. While it took around 100 epochs for OLS to reach an acceptable MSE score, the neural network required at least 2000 epochs to achieve similar accuracy. This highlights that, given sufficient training epochs, neural networks are capable of fitting complex data patterns exceptionally well. However, the downside is that they require significantly more epochs to converge compared to simpler methods like OLS.

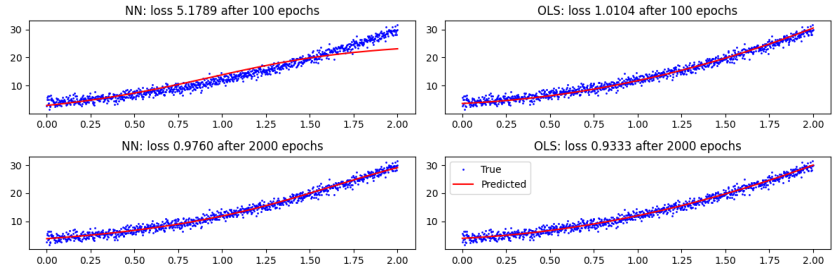

Figure 7: Comparison of OLS and NN with Plain Gradient Descent at 100 and 2000 epochs

Effect of Different Gradient Methods on Convergence

Table 11 shows that SGD, RMSProp, and Adam with momentum are the most effective optimizers, achieving the fastest convergence among all methods tested. This shows the importance of choosing the right optimiser. See the Appendix for visualisation of the convergence.

Table 11
Comparison of convergence of all gradient methods with and without momentum for Neural Network.

method ↓	epoch to convergence	final MSE after 200 epochs
Plain GD w/o m	DNC	2.3295
Plain GD w/ m	DNC	1.3204
SGD w/o m	24	1.2195
SGD w/ m	10	1.0992
AdaGrad w/o minibatch w/o m	DNC	44.0760
AdaGrad w/o minibatch /w m	DNC	24.0301
AdaGrad w/ minibatch w/o m	DNC	5.3485
AdaGrad w/ minibatch /w m	105	1.1047
RMSProp w/o minibatch w/o m	DNC	1.5499
RMSProp w/o minibatch /w m	101	1.0152
RMSProp w/ minibatch w/o m	25	0.9815
RMSProp w/ minibatch /w m	8	0.9940
Adam w/o minibatch w/o m	DNC	1.2551
Adam w/o minibatch /w m	125	1.0560
Adam w/ minibatch w/o m	19	0.9535
Adam w/ minibatch /w m	12	1.0905

Effect of Different AFs for Output Layer

The choice of activation function for the output layer is crucial, as certain functions are suited specifically for either regression or classification tasks. In Figure 8, we observe that the Sigmoid activation function got stuck in a local minimum, with an MSE around 222, while models with ReLU and Leaky ReLU activations successfully converged to an acceptable MSE close to 0. This experiment used Plain GD with momentum as the gradient method.

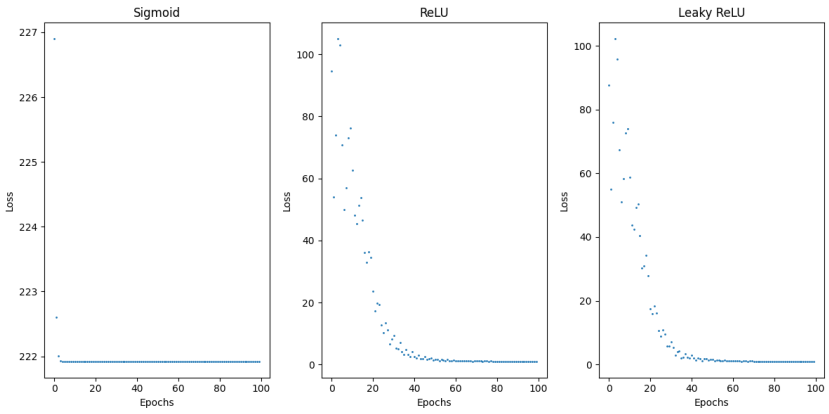

Figure 8: Comparison of sigmoid, ReLU, and Leaky ReLU as activation function for output layer.

5.2.3. Classification Performance (Logistic Regression)

For classification tasks, performance was evaluated using the Binary Cross Entropy Loss as the primary metric, while accuracy is also used to evaluate the model.

Effect Learning Rate & L2 Regularisation

This section analyzes the impact of learning rate and L2 regularization on the performance of the Logistic Regression model. Table 12 shows that both parameters significantly influence the convergence rate. The results suggest that a learning rate of 0.1 and an L2 regularization value of 0.1 provide the optimal balance for this dataset. Refer to the Appendix for a visualization of these comparisons.

Table 12
This table shows the BCEL values at different learning rates for logistic regression after 500 epochs.

learning rate ↓	L2 regularisation→	0.0001	0.001	0.01	0.1
0.0001		131	129	129	124
0.001		87	87	84	70
0.01		37	37	33	18
0.1		14	14	8	2

5.2.4. Feed-Forward Neural Network (Classification Task):

Comparison with Logistic Regression

For classification tasks, logistic regression is often used as a baseline. In this project, the logistic regression model was implemented using SGD with L2 regularization. As shown in Table 13, both logistic regression and the neural network performed well on this dataset. Interestingly, the neural network tended to overfit, achieving a perfect fit on the training data but with slightly

reduced accuracy on the test set. Figure 9 illustrates that both models required a similar number of epochs to converge.

Table 13
This table shows the comparison of Binary Cross Entropy Loss of Neural Network and Logistic Regression after 500 epochs of training

model ↓	Train Accuracy	Test Accuracy
NN w/ SGD	1.000	0.9649
Logistic Regression w/ SGD	0.9846	0.9825

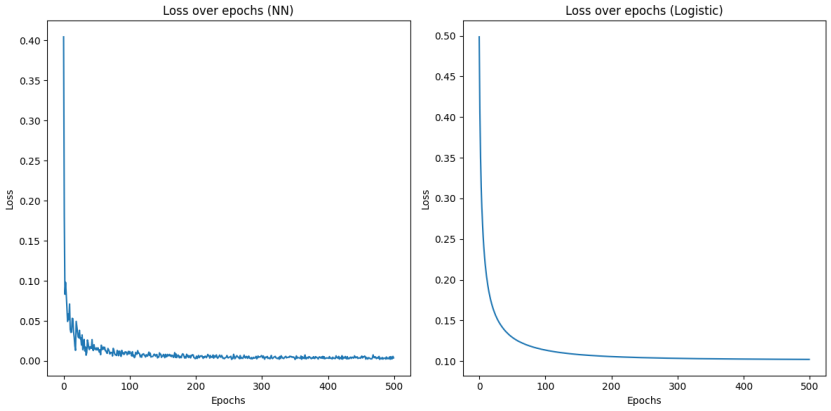

Figure 9: Comparison of convergence between neural network and logistic regression

Effect of AFs of Output Layer

The choice of AF is crucial in model performance, particularly in classification tasks where it can greatly influence accuracy and convergence. Table 14 shows that among the activation functions tested, Softmax consistently achieved the highest performance, making it the optimal choice for this classification problem. Softmax is particularly suited for multi-class classification as it outputs probabilities across classes. This contrasts with other activation functions like Sigmoid or ReLU, which are better suited for regression tasks. See Appendix for the visualisation of the comparison.

Table 14
This table shows the comparison for different AFs used for the final output layer of the NN after 200 epochs.

AF ↓	num of epochs to convergence
sigmoid	16
ReLU	15
Leaky ReLU	16
softmax	6

Exploring Different Architectures

I also experimented with various neural network (NN) architectures by adjusting the number of hidden layers and neurons in each layer. Table 15 highlights that the configurations 16-8-2, 16-16-8-2, and 16-16-16-8-2 yielded the fastest convergence, achieving satisfactory performance in approximately 7 epochs. Based on these results, I selected the 16-8-2 architecture for further testing due to its lower computational cost and efficiency.

Interestingly, these findings illustrate that adding more layers and neurons does not necessarily improve performance and can even lead to diminishing returns. While larger architectures allow for increased model capacity, they also introduce greater risk of overfitting, slower convergence, and higher computational demand without necessarily enhancing predictive power. The Appendix provides visualizations of the convergence behavior across these architectures, showing how simpler architectures can still achieve competitive results with optimal layer configuration.

Table 15
This table shows the comparison for different architectures used for the NN after 100 epochs.

layers ↓	num of epochs to convergence
32-16-2	16
16-8-2	7
32-16-16-2	11
16-8-8-2	13
32-32-16-2	8
16-16-8-2	16
32-32-32-16-2	14
16-16-16-8-2	6

5.2.5. More Insights on Gradient Methods

Effect of t0/t1 Ratio on SGD

For Stochastic Gradient Descent (SGD), we introduced two additional hyperparameters, t0 and t1 , which control the learning rate schedule during training. The ratio of these parameters (t0/t1) directly influences how the learning rate decreases over time, impacting the convergence speed and stability of the model.

As seen in Figure 10, a ratio of 0.001 (with t0=1 and t1=1000) appears optimal, leading to the fastest and most stable convergence. This finding aligns well with the default settings for t0 and t1 , as this specific ratio allows the model to maintain a learning rate that is neither too aggressive (which could cause instability) nor too conservative (which might slow down convergence).

Tuning the t0/t1 ratio can be especially useful for more complex datasets or tasks with varying gradients, as it enables SGD to adapt the learning rate schedule to the model's needs more effectively. This balance helps avoid scenarios where the model might oscillate around a minimum or take excessive epochs to reach a satisfactory level of convergence.

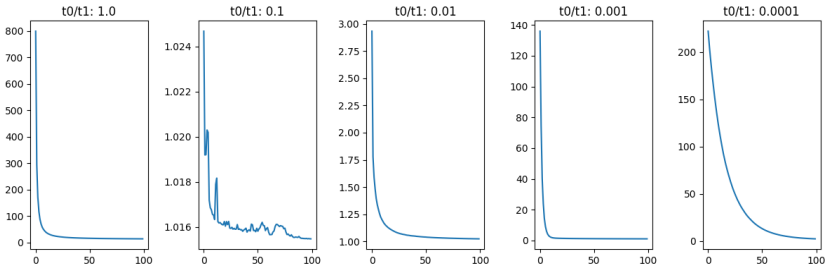

Figure 10: Comparison of different t0/t1 ratio for OLS with SGD

5.3. Pros and Cons of Various Models

In this section, I will discuss the pros and cons of the algorithms utilized in this study: Ordinary Least Squares (OLS), Ridge Regression, Stochastic Gradient Descent (SGD), Adagrad, RMSProp, Adam, and Neural Networks (NN). Each algorithm has its unique strengths and limitations, making them suitable for different types of regression and classification problems. Understanding the trade-offs involved in choosing each method is crucial for selecting the right approach based on the problem at hand.

5.3.1. OLS

Pros

Simplicity: OLS is one of the simplest regression algorithms, relying on closed-form solutions to estimate coefficients. This makes it computationally efficient, especially for small datasets. The implementation is straightforward, requiring minimal tuning, and it can often provide good results if the data follows a linear relationship.

Interpretability: OLS provides easily interpretable coefficients, making it ideal for applications where understanding the relationship between variables is crucial.

Cons

Overfitting Risk in High-Dimensional Data: As the number of features increases, the likelihood of overfitting grows. The model may fit the noise in the data rather than the underlying

relationship, especially in the presence of irrelevant features.

5.3.2. Ridge

Pros
Regularization to Prevent Overfitting: Ridge regression addresses the issue of overfitting by adding an L2 penalty to the coefficients. This regularization term shrinks the coefficients, helping to control model complexity and improve generalization.

Stable with Collinearity: Ridge regression can handle multicollinearity more effectively than OLS. The regularization prevents the model from giving disproportionately large weights to highly correlated features, improving stability.

Cons
Interpretability is Reduced: The introduction of regularization makes the model's coefficients harder to interpret directly, as they are no longer purely dependent on the underlying data, but also on the regularization term.

5.3.3. Logistic Regression

Pros
Works Well with Linearly Separable Data: Logistic regression can perform very well when the classes are linearly separable, and it provides a solid baseline for classification problems.

Cons
Limited to Linear Decision Boundaries: Logistic regression assumes a linear decision boundary, making it less effective for datasets where the decision boundary is non-linear. It may underperform compared to more complex models like decision trees or neural networks in such cases.

Sensitive to Feature Scaling: Logistic regression requires that input features are scaled appropriately, as large discrepancies in feature magnitudes can lead to slow convergence or poor performance.

5.3.4. NN

Pros
Modeling Complex Relationships: Neural networks are highly powerful for modeling complex, non-linear relationships between inputs and outputs. They can learn intricate patterns that are difficult to capture with traditional models like OLS or Ridge.

Customizable Architectures: Neural networks offer a high degree of flexibility, allowing you to experiment with different numbers of layers, neurons, and activation functions to optimize performance for specific tasks.

Cons

High Computational Cost: Training neural networks, especially deep ones, can be computationally expensive. They often require substantial resources (e.g., GPUs) and can be time-consuming to train.

Difficult to Interpret: Neural networks are often considered "black-box" models because understanding the relationship between input features and output predictions can be challenging. This lack of interpretability is a major limitation in some fields where transparency is critical, such as healthcare or finance.

6. Conclusion

In this project, we explored various models and optimizers, ultimately determining that the 16-8-2 neural network with the Adam optimizer (with momentum) provided the most robust and performant solution. This configuration not only converged the fastest but also demonstrated excellent accuracy, making it a well-balanced choice for both regression and classification tasks.

The algorithms examined offer a range of strengths and weaknesses. For regression tasks, OLS and Ridge Regression are effective for simpler datasets where linear relationships are present, while Neural Networks excel at capturing more complex, non-linear patterns. In classification tasks, Logistic Regression serves as a solid baseline, particularly for binary problems, but Neural Networks combined with the Adam optimizer outperform it for more intricate, non-linear, or multi-class problems.

Ultimately, selecting the right model requires careful consideration of factors like optimization methods, activation functions, and regularization techniques. By fine-tuning these elements, one can achieve optimal performance tailored to the specific needs of a given task, ensuring better generalization and accuracy in real-world applications.

Acknowledgments

Many references are taken from the course lecture notes https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html

References

[1] U. C. Science, Lecture notes (linear regression) - ucla computer science, 2019. URL: https://web.cs.ucla.edu/~chohsieh/teaching/CS260_Winter2019/notes_linearregression.pdf.

[2] W. N. van Wieringen, Lecture notes on ridge regression, 2023. URL: https://arxiv.org/abs/1509.09169. arXiv:1509.09169.
[3] N. Bhatt, Logistic regression with l2 regularization from scratch, 2023. URL: https://medium.com/@bneeraj026/logistic-regression-with-l2-regularization-from-scratch-1bbb078f1e88.
[4] S. Raschka, V. Mirjalili, Python machine learning: Machine learning and deep learning with Python, scikit-learn, and TensorFlow 2, Packt publishing ltd, 2019.
[5] S. R. Dubey, S. K. Singh, B. B. Chaudhuri, Activation functions in deep learning: A comprehensive survey and benchmark, Neurocomputing 503 (2022) 92–108. URL: https://www.sciencedirect.com/science/article/pii/S0925231222008426. doi:https://doi.org/10.1016/j.neucom.2022.06.111.
[6] P. Belagatti, Understanding the softmax activation function: A comprehensive guide, 2024. URL: https://www.singlestore.com/blog/a-guide-to-softmax-activation-function/.
[7] Niru, Understanding backpropagation with softmax, 2023. URL: https://stackoverflow.com/questions/58461808/understanding-backpropagation-with-softmax.
[8] A. L. Maas, Rectifier nonlinearities improve neural network acoustic models, in: Rectifier Nonlinearities Improve Neural Network Acoustic Models, 2013. URL: https://api.semanticscholar.org/CorpusID:16489696.
[9] L. Jin, S. Li, B. Hu, M. Liu, A survey on projection neural networks and their applications, Applied Soft Computing 76 (2019) 533–544. URL: https://www.sciencedirect.com/science/article/pii/S1568494619300067. doi:https://doi.org/10.1016/j.asoc.2019.01.002.
[10] D. Villarraga, Adagrad, 2021. URL: https://optimization.cbe.cornell.edu/index.php?title=AdaGrad.
[11] V. Bushaev, Understanding rmsprop — faster neural network learning, 2018. URL: https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a.
[12] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, 2017. URL: https://arxiv.org/abs/1412.6980. arXiv:1412.6980.
[13] R. Bhat, Gradient descent with momentum, 2020. URL: https://towardsdatascience.com/gradient-descent-with-momentum-59420f626c8f.

A. Appendix: Various Results

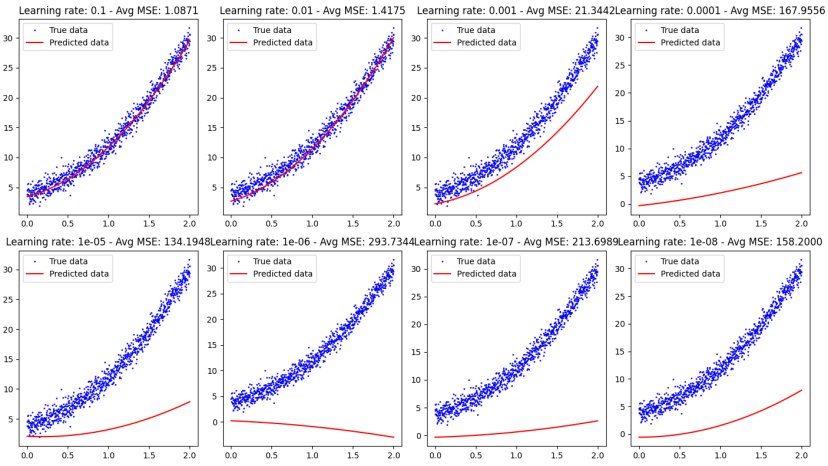

Figure 11: Visualisation of Ridge /w Plain GD fitting at epoch=100 at different learning rates

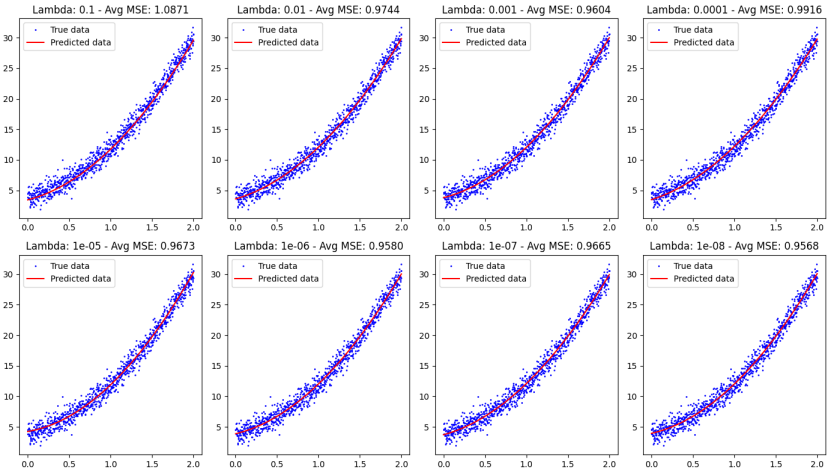

Figure 12: Visualisation of Ridge /w Plain GD fitting at epoch=100 at different lambda

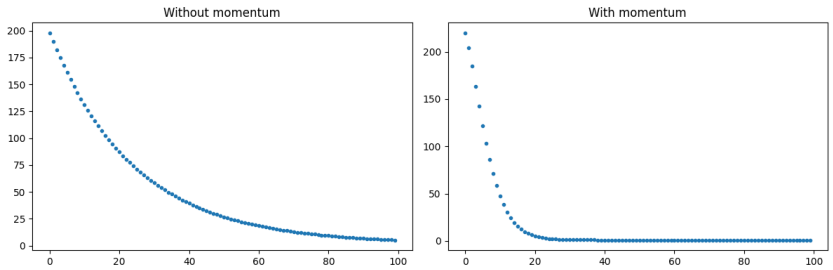

Figure 13: Convergence of Ridge /w Plain GD /w and w/o momentum

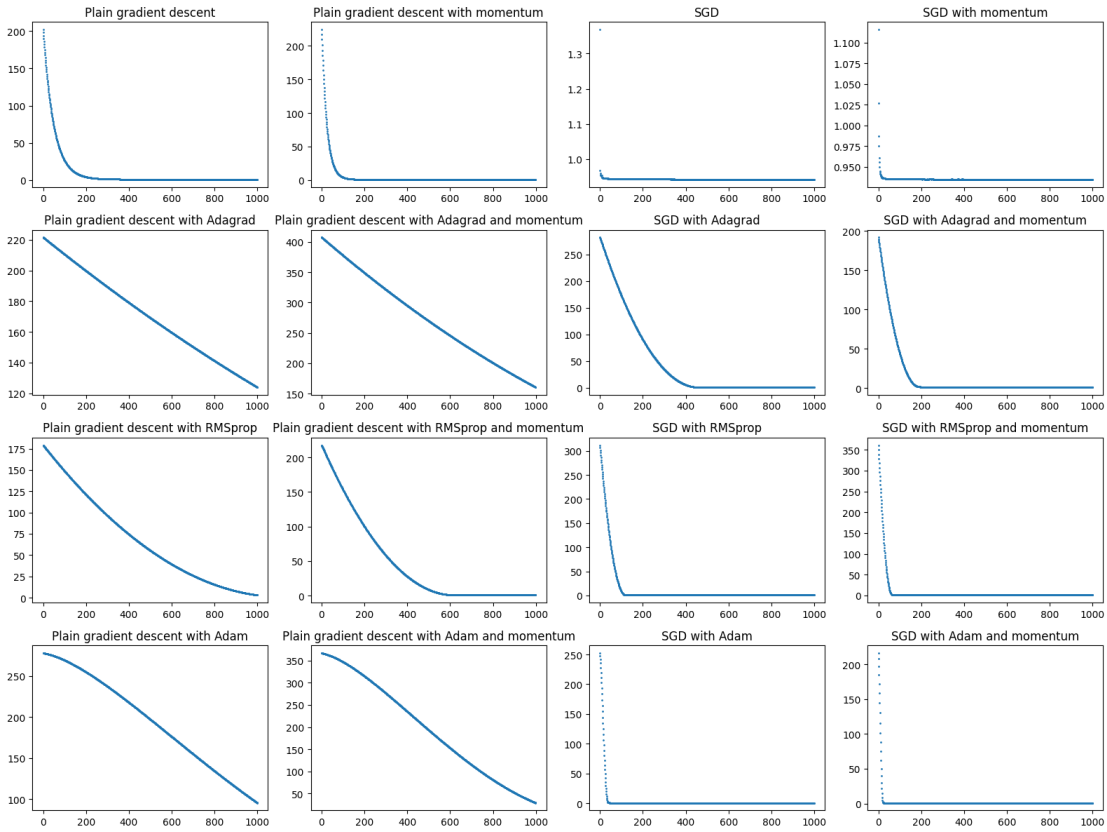

Figure 14: Visualisation of Convergence of all gradient methods with and without momentum.

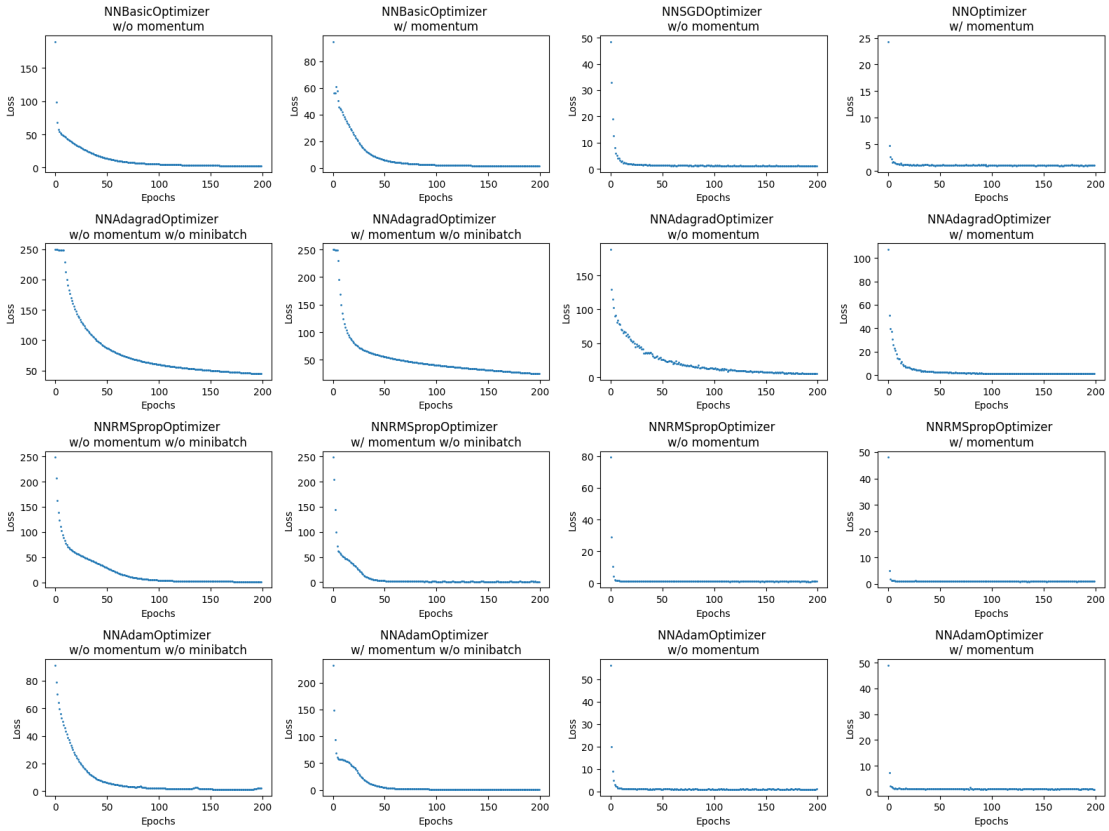

Figure 15: Visualisation of Convergence of all gradient methods with and without momentum for Neural Network

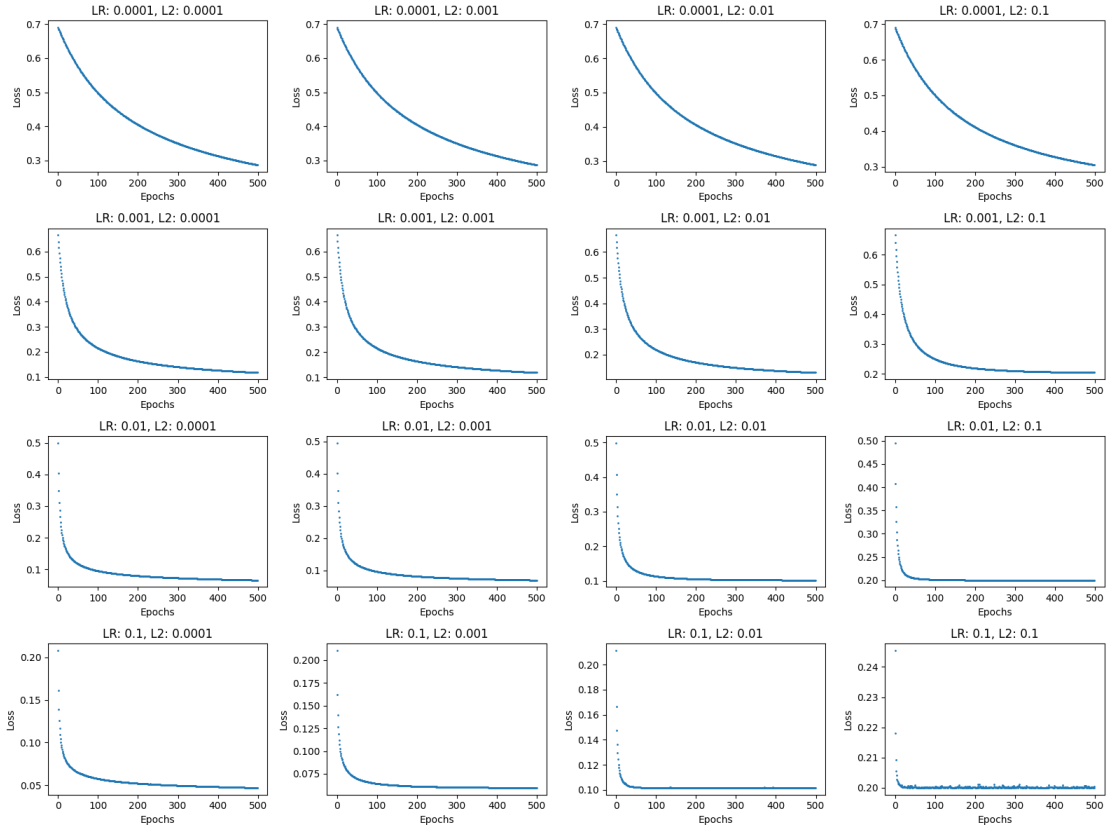

Figure 16: Visualisation of the effects of learning rate and l2 regularisation has on convergence of logistic regression.

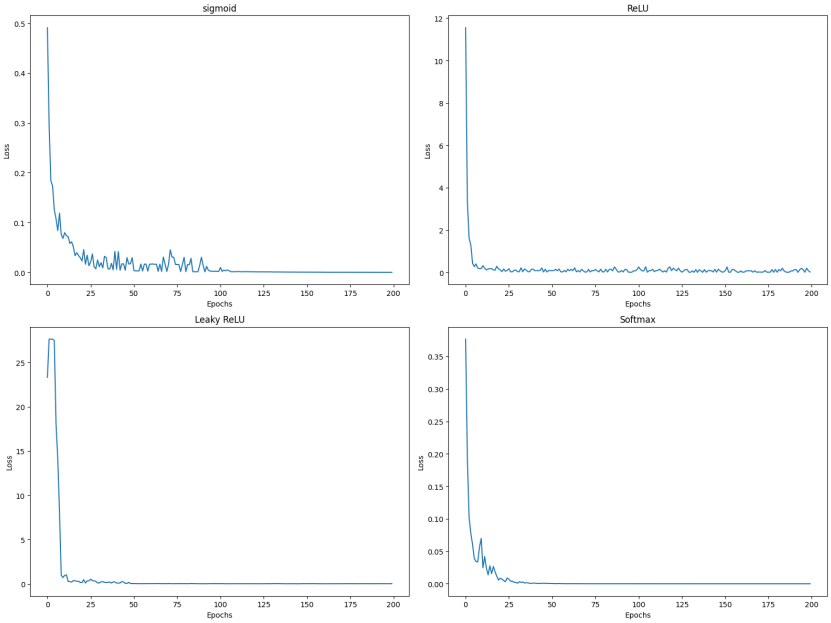

Figure 17: Visualisation of convergence of NN model for classification using different AFs for output layer

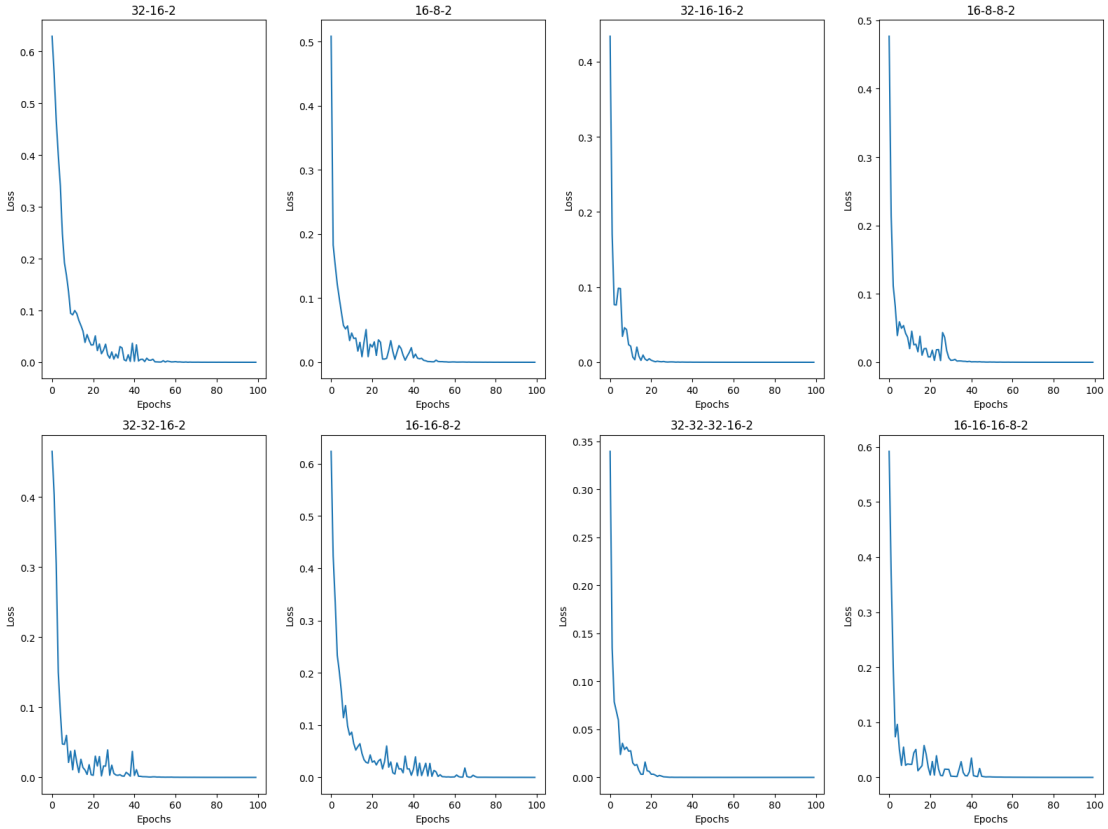

Figure 18: Visualisation of convergence of NN model for classification using different layers.