Regression and Classification Approaches Using Linear and Neural Network Models: A Study with Franke's Function and Breast Cancer Data Project 2

Eleonora Gandini Mazzarelli, Anna Milanesi, Mia Margrethe Ramberg Storstad

November 4, 2024

Abstract

In this study, we evaluate the effectiveness of various regression and classification models applied to two datasets: Franke's function for regression and the Wisconsin Breast Cancer dataset for binary classification.

Breast cancer is a major global health concern, affecting over 2 million women annually worldwide [1]. Early detection and accurate classification are crucial to improving patient outcomes, underscoring the importance of effective predictive modeling. However, challenges like high dimensionality and overfitting often limit model performance.

To address these problems, we applied a variety of regression and classification techniques, evaluating their convergence, accuracy, and generalizability. For regression, our Stochastic Gradient Descent (SGD) neural networks with Leaky ReLU and Sigmoid activations achieved a minimum Mean Squared Error (MSE) of 0.0012 on Franke's function, outperforming traditional ridge regression, which reached an MSE of 0.11. In the classification task, Sigmoid-activated logistic regression models and neural networks achieved similarly high accuracy rates, with neural networks reaching a peak accuracy of 0.9825.

Our results highlight the ability of neural networks to address data variability and complexity more effectively than linear models. In fact, while logistic regression remains a robust choice for binary classification, neural networks excel in handling non-linear relationships, making them valuable for medical applications where predictive accuracy and early detection are paramount.

Click here to go the GitHub repository

Contents

1 Introduction

There are many different methods for evaluating data and developing predictive models in machine learning, yet this project focuses on both regression and classification strategies. From more complex neural network architectures to more conventional techniques like logistic regression and linear regression, the main goal is to create, apply, and evaluate machine learning models. Our objective is to assess the efficacy and constraints of each method in identifying patterns in intricate datasets by examining them.

We looked into linear regression techniques applied to the Franke Function for regression analysis. Starting with Ordinary Least Squares (OLS), this procedure was extended to incorporate regularization methods that improve the generalization of the model. Along with more sophisticated techniques like AdaGrad, RMSprop, and Adam, we also investigated a variety of optimization algorithms, such as gradient descent (GD) and stochastic gradient descent (SGD). The purpose of this study is to determine how convergence and predictive accuracy are affected by adjusting these optimization techniques.

We analyzed the Wisconsin Breast Cancer dataset using logistic regression and neural networks for the classification component. This dataset allows us to compare the effectiveness of customized neural network models against conventional logistic regression, which is especially pertinent for binary classification tasks. In particular, we looked at how the network behaved and how sensitive it was to hyperparameters like regularization terms and learning rates in order to comprehend how they affected model performance.

Overall, this project offers a thorough framework for comprehending and comparing machine learning algorithms in practical settings. We show common applications of regression and classification methods by examining these specific datasets, but we give attention to the special difficulties and insights they offer. The report's structure is as follows: the Method section describes each algorithm and strategy that was employed, such as gradient descent techniques, neural networks, and validation methods. The Results and Discussion section presents the findings for both regression and classification tasks, accompanied by detailed analysis and visualization to illustrate model performance under varying configurations. And finally, the Conclusion provides a summary of our findings, reflections, difficulties encountered, and ideas for further research.

By examining these models within the context of regression and classification, we aim to contribute to the ongoing conversation around machine learning's role in healthcare and the challenges associated with deploying reliable predictive models. This project not only highlights the strengths and limitations of traditional and neural network-based approaches but also underscores the importance of hyperparameter tuning, model regularization, and appropriate performance metrics in achieving optimal results.

2 Method

2.1 Linear Regression

A basic statistical method for modeling the relationship between one or more independent variables and a continuous dependent variable is linear regression. It is based on the assumption that the dependent variable y and the predictors have a linear relationship, as shown by the formula $\tilde{y} = X\beta$, where X is the design matrix containing our predictor variables, β is the vector of unknown parameters that we wish to estimate in order to get the model to fit us the best, and $\tilde{y}$ is the predicted outcome.
The goal of linear regression is to identify the values of β that minimize a cost function, $C(X, \beta)$, which determines the discrepancy between observed and predicted values [2]. To find the minimum of this function, we usually use numerical techniques.

Ordinary Least Squares (OLS)

The most common approach for estimating the coefficients in linear regression is the Ordinary Least Squares (OLS) method. The sum of squared residuals, or the squared difference between the observed and predicted values, is minimized by OLS.
The cost function for OLS is given by:

$$C_{\text{OLS}}(\beta) = \frac{1}{n}(y - X\beta)^T(y - X\beta)$$

The best linear fit is obtained by minimizing this function, which produces the optimal parameter $\hat{\beta}$. The model variance, the Mean Squared Error (MSE), and the confidence intervals for β can all be estimated using OLS. The matrix $X^T X$ can frequently be inverted to find the solution [3].

Ridge Regression

Ridge regression includes a penalty term to the linear model to avoid overfitting, particularly when there are several predictors. A regularization term is integrated into the Ridge regression cost function:

$$C_{\text{Ridge}}(X, \beta) = (y - X\beta)^T(y - X\beta) + \lambda\beta^T\beta$$

where λ is the regularization parameter that regulates the coefficients' degree of penalty. High values of λ move the coefficients toward zero, which lowers model variance and complexity but may also increase bias. In contrast, Ridge regression more closely appears like OLS when λ is small. Ridge regression generates a more stable and robust model than OLS, which is especially helpful in high-dimensional settings where interpretability may be secondary to model performance [3].

2.2 Classification

Logistic Regression

Logistic regression is widely used method for classification tasks with discrete (e.g., binary or categorical) outcomes. Logistic regression models the likelihood that an observation falls into a specific class rather than predicting a continuous outcome. The logistic (sigmoid) function is used to model the relationship between predictors and the binary outcome: The formula

$$p(y = 1|x, \beta) = \frac{e^{\beta_0+\beta_1 x}}{1 + e^{\beta_0+\beta_1 x}}$$

The output of this function, which represents the likelihood that an observation belongs to one of two classes, is compressed to fall between 0 and 1 [4].
Maximum Likelihood Estimation (MLE) is used in logistic regression to estimate the coefficients β, with the goal of maximizing the probability that the observed data fits the model [5]. This is achieved using a cost function called cross-entropy, which calculates the difference between the actual binary outcomes and the predicted probabilities. Since it provides probabilities for each class prediction, logistic regression is particularly common for binary classification tasks because it is simple to understand.

Cross Entropy

Cross-entropy is a frequently used performance statistic in classification models such as logistic regression. Cross-entropy, which may be expressed as follows:

$$C(\beta) = -\sum_{i=1}^{n} (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

where y_i is the real binary label and $\tilde{y}_i$ is the expected probability, measures the difference between the predicted probability distribution and the actual distribution of the data. A lower cross-entropy indicates greater model performance. Cross-entropy measures how well the anticipated probabilities match the observed outcomes. It works especially well for assessing how successfully probabilistic models perform [5].

Cross Ridge Entropy

To counteract overfitting when doing logistic regression, we can add an extra term to the cost function that is proportional to the size of the weights. To measure the size of the weights, L2-norm is used, which gives us the Cross Ridge Entropy function [6]:

$$C(\beta) = -\sum_{i=1}^{n} (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)) + \lambda \sum_{j=1}^{m} \beta_j^2$$

where λ is the regularization parameter and β is the weights at each feature j. The addition of this ridge penalty has the effect of shrinking the weights toward zero but does not set any weights exactly to zero. This shrinkage reduces the model's sensitivity to individual features, making it less likely to overfit.

2.3 Scaling data

Feature scaling is an important step in machine learning preprocessing because real-world data frequently contains features with variable units and sizes. This variance has a substantial influence on models such as Mean Squared Error (MSE), which are sensitive to data scale. To remedy this, scaling characteristics makes them more similar and prevents biased findings owing to outliers. The most commonly used scaling technique is Standardization, which adjusts the characteristics by subtracting the mean and dividing by the standard deviation. This produces features with a mean of zero and a standard deviation of one. For each feature:

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \bar{x}_j}{\sigma(x_j)}$$

where $\bar{x}_j$ and $\sigma(x_j)$ are the mean and standard deviation, respectively, of feature x_j [3] [7].
When it comes to various machine learning models, including Neural Networks (NN), Ridge regression, and Ordinary Least Squares (OLS) regression, scaling is a crucial factor to take into account. Every technique has distinct features that affect how scaling affects the way it works.
Since variables with larger magnitudes may dominate the model fitting process, different feature scales in ordinary least squares (OLS) regression can result in biased coefficient estimates. Modeled relationships are more accurate when features are standardized to ensure that each predictor contributes equally [8].
Ridge regression reduces overfitting by introducing a regularization term, but it only works if features are scaled correctly. Features with greater variances may distort the penalty applied to coefficients in the absence of standardization, which would impair model performance. By facilitating uniform regularization across all features, standardization improves stability and generalization [9].
Algorithms like gradient descent are used in the neural network optimization process. Inefficient training may result from a distorted cost function landscape caused by input features that are not on the same scale. A smoother optimization landscape is produced by standardizing inputs, which facilitates quicker convergence and better model performance overall [10].

In summary, feature scaling is strongly recommended for datasets with different units and ranges, even though it may not always be required. By putting this technique into practice, model training is improved, and predictions and performance on a range of machine learning tasks are strengthened. Achieving the best model effectiveness requires knowing when to use feature scaling.

2.4 Model Validation

To make sure a model performs properly when applied to new, unseen data, effective model validation is essential. The process of choosing a model can be significantly influenced by common evaluation criteria, which varies based on the problem and model type. In this section, we go over two important metrics that are used in classification and linear models.

Mean Squared Error (MSE)

A popular metric for assessing linear regression models is Mean Squared Error, or MSE. It can be computed as follows:

$$\text{MSE}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

is the average of the squared discrepancies between the observed and expected values [3].
Whereas a high MSE implies the model's predictions are less accurate, a low MSE shows the model accurately predicts the observed values. When comparing several linear models and adjusting regularized regression parameters, such the λ parameter in Ridge regression, MSE is especially helpful.

Accuracy

To assess the performance of our neural network, we measure its ability to make accurate predictions on previously unseen data, specifically the test dataset. We evaluate this performance using the accuracy metric.
Accuracy is defined as the ratio of correctly classified instances to the total number of instances in the test set. A perfect classifier achieves an accuracy score of 1, indicating that all predictions are correct. Mathematically, accuracy can be expressed as:

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

where I is an indicator function, which equals 1 if the predicted label $\tilde{y}_i$ matches the true label y_i, and 0 otherwise [6]. This formula effectively counts the number of correct predictions and divides it by the total number of predictions, giving a straightforward measure of model accuracy.

2.5 Gradient Descent

In machine learning, minimizing the cost function C is the main goal. This relationship is frequently expressed in the context of regression as follows:

$$\frac{\partial C}{\partial \theta_i} = 0$$

where the regression line $\hat{y}_i = \theta_0 + \theta_1 x_{i,0} + \ldots + \theta_n x_{i,n}$ has parameters θ_j and C is the cost function [11].
Gradient Descent (GD) is introduced because this problem might not have an easy analytical solution. GD is a technique for calculating weights θ in neural networks or regression analysis. The objective is to create a model that makes predictions on new data, given a dataset containing inputs $\mathbf{x}$ and outputs $\mathbf{y}$. Here, $\mathbf{x}$ is a $n \times m$ matrix.

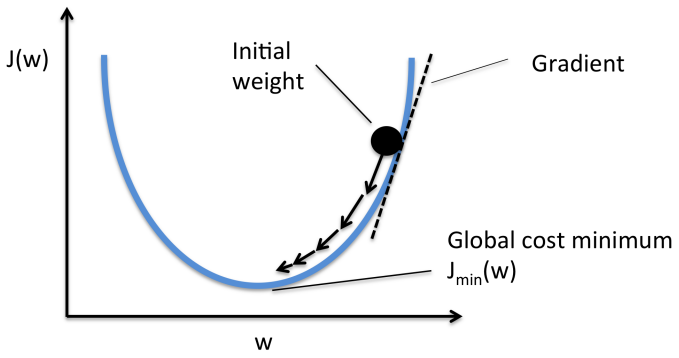


Figure 1: Gradient Descent visualization, in which the gradient of the cost function is calculated after selecting an initial set of weights. then descending to the global/local minimum by following the gradient. Image: [12]

Plain Gradient Descent

We start by choosing a random set of weights that match the quantity of inputs. The model that we want to optimize looks like this:

$$\hat{y} = \mathbf{X}\theta$$

This set of weights' associated cost is calculated, and then the cost's gradient is determined:

$$\nabla C(\theta) = \begin{pmatrix} \frac{\partial C}{\partial \theta_0} \\ \frac{\partial C}{\partial \theta_1} \\ \vdots \\ \frac{\partial C}{\partial \theta_n} \end{pmatrix}$$

The cost function's strongest slope is indicated by the gradient. The gradient is then used to update each weight:

$$\theta_{i+1} = \theta_i - \eta \frac{\partial C}{\partial \theta_i}$$

where the learning rate is represented by η, which establishes the minimum step size. A learning rate that is too small may delay convergence, while one that is too great could cause the minimum to be exceeded [11].

Momentum Gradient Descent

It can be computationally expensive to use gradient descent. We use Momentum Gradient Descent (MGD), which is defined by the following equations, to accelerate convergence:

$$v_{i+1} = \gamma v_i + \eta \frac{\partial C}{\partial \theta_i}$$

$$\theta_{i+1} = \theta_i - v_{i+1}$$

The size of the acceleration towards the minimum is determined by the γ factor [11]. Without altering the learning rate, we raise the weight change. Although the velocity v_{i+1} drops as $\eta \frac{\partial C}{\partial \theta_i}$ decreases, we may still overshoot the minimum. By using the example of a ball going down a hill, we can observe that while the velocity v_{i+1} is high at first, it will gradually drop as the acceleration $\eta \frac{\partial C}{\partial \theta_i}$ does, bringing us to the bottom of the hill.

Stochastic Gradient Descent

Randomness can be added using Stochastic Gradient Descent (SGD) to overcome some of the limitations of conventional gradient descent. One way to define the cost function is as a sum over n data points:

$$C(\beta) = \sum_{i=1}^{n} c_i(x_i, \beta)$$

The gradient can then be computed as:

$$\nabla_\beta C(\beta) = \sum_{i=1}^{n} \nabla_\beta c_i(x_i, \beta)$$

By applying gradients to a subset of the data, known as minibatches, represented by B_k, where $k = 1, \ldots, \frac{n}{M}$ and M is the minibatch size, SGD adds stochasticity. The gradient update turns into:

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k} \nabla_\beta c_i(x_i, \beta)$$

Because each gradient descent step employs a random subset of data, this method enables faster convergence [11].

2.6 Learning Rate

Generally speaking, it is impossible to predict from the start what value γ or η should be. One of the most important aspects of solving supervised learning problems is adjusting these hyper parameters. By employing an algorithm with adaptive learning rates, we can lessen some of the difficulties associated with hyper parameter adjustment.

AdaGrad

AdaGrad, or Adaptive Gradient, does at the name suggests, adapts the learning rate for each parameter. The algorithm scales the parameter based on the "history" of its gradients. It does so by scaling the parameters inversely proportional to the square root of all the previous squared values of the gradient [9]. We get the update rule for the parameters θ:

$$g_t = \nabla_\theta E(\theta)$$
$$G_t = G_{t-1} + g_t^2$$
$$\theta_{t+1} = \theta_t - \eta \frac{g_t}{\sqrt{G_t + \epsilon}}$$

where $E(\theta)$ is the loss function, and ϵ is a small constant to avoid dividing by zero [9].

RMSprop

An issue with AdaGrad is the possibility of the learning rate to become too small over time because of the accumulation of squared gradients, especially in a non-convex setting. Root Mean Square Propagation tries to improve this algorithm by making the more recent gradients have more influence over the learning rate than the older gradients. It does so by changing the learning rate based on a moving average of the squared gradients. The update rule is given by

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

with s_t as the second-order moment equation, and β controls the averaging time of the second moment and is often set to $\beta = 0.9$ [9].

Adam

The Adam algorithm derives its name from "adaptive moments", as it builds on RMSprop by incorporating momentum and a bias correction. Momentum is incorporated as an estimate of the first-order moment of the gradient. This is done by adding momentum to the rescaled gradients. The bias correction is added to make sure the first-order moment does not have too small of a gradient average, and the second-order moment to not have a too small learning rate in the beginning [9]. The update rule for Adam is given by

$$g_t = \nabla_\theta E(\theta)$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$s_t = \beta_2 s_{t-1} + (1-\beta_2) g_t^2$$
$$m_t = \frac{m_t}{1-\beta_1^t}$$
$$s_t = \frac{s_t}{1-\beta_2^t}$$
$$\theta_{t+1} = \theta_t - \eta_t \frac{m_t}{\sqrt{s_t} + \epsilon}$$

where β_1 and β_2 is the memory lifetime of the first and second moment, and is often set to 0.9 and 0.99 respectively [11].

2.7 Neural Network

Artificial neural networks (ANN) are computational models that learn to perform tasks by analyzing examples, typically without needing specific, pre-programmed instructions. They are designed to resemble biological systems, where neurons communicate by passing signals represented as mathematical functions between layers. Each is composed of layers of neurons that are connected to one another through weighted connections. By optimizing these weights during training, the network can generate classifications or predictions without the need for explicit task-specific rules. Two hidden layers in a neural network are shown in Figure 3 [13].

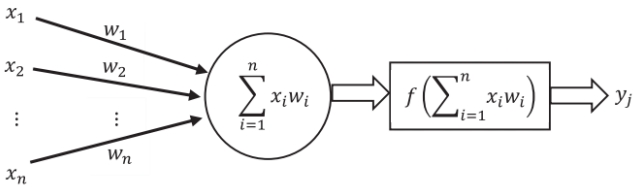


Figure 2: Illustration of a single perceptron model. (Image: [13])

2.8 Feed-Forward Neural Networks

The most basic type of neural networks are called feed-forward neural networks, or FFNNs. Data flows from the input layer to the output layer through any hidden layers in a single direction in an FFNN. Through the use of activation functions and weighted sums, each layer modifies the input data to produce complex data representations. Note that this would be the same as multinomial logistic regression in the absence of a hidden layer [10].

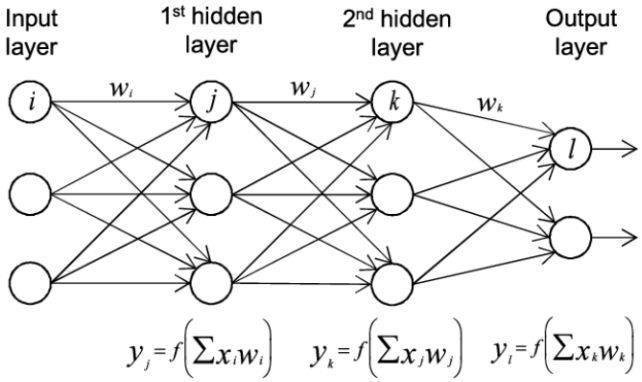


Figure 3: Illustration of a neural network with two hidden layers. (Image: [13])

Circles stand in for nodes, and arrows show the connections between them as well as the direction in which information flows. Furthermore, every arrow represents a weight variable. This is referred to as a fully-connected FFNN since we can see that every node in a layer is connected to every node in the layer that follows.

Input information travels through the network without going backwards from the input layer, through the hidden layers in between, and out to the output layer when each node, which is defined by a model function, passes information to the nodes ahead of it [10].

For a neuron j in layer $l+1$, the weighted input $z_j^{(l+1)}$ is calculated as:

$$z_j^{(l+1)} = \sum_k w_{j,k}^{(l+1)} a_k^{(l)} + b_j^{(l+1)}$$

where [14]:

- $w_{j,k}^{(l+1)}$ are the weights connecting neurons from layer l to $l+1$,
- $a_k^{(l)}$ represents the activation from the previous layer,
- $b_j^{(l+1)}$ is the bias term for neuron j in layer $l+1$.

2.9 Backpropagation

The neural network training algorithm known as backpropagation uses weight adjustments to reduce the discrepancy between the network's predicted and actual outputs. In order to minimize error, the network iteratively modifies weights using gradient descent. The backpropagation algorithm's fundamental mathematics involves determining a cost function with regard to weights and biases: $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ (by the chain rule). The error is calculated after the output layer and the targets are compared [9]. Next, iterating backward from the previous layer, the backpropagation algorithm calculates the gradient using the derivatives for each layer. The gradient enables us to evaluate the speed at which changing the weights and biases affects the cost. Because they are initially initialized at random, this enables the NN to update the weights and biases values.

To minimize this cost function, backpropagation computes the gradients of C with respect to each weight $w_{j,k}$ and bias b_j using the chain rule. The update rule for each weight and bias becomes:

$$w_{j,k}^{(l)} \leftarrow w_{j,k}^{(l)} - \eta \frac{\partial C}{\partial w_{j,k}^{(l)}},$$

$$b_j^{(l)} \leftarrow b_j^{(l)} - \eta \frac{\partial C}{\partial b_j^{(l)}}$$

where η is the learning rate [14].

2.10 Activation Functions

Activation functions allow the network to capture intricate relationships in the data by adding non-linearity to the model. An activation function is applied to the weighted input z to determine the output of each neuron [9].

Function	$f(z_i)$	$f'(z_i)$
Sigmoid	$(1+e^{-z_i})^{-1}$	$f(z_i)(1-f(z_i))$
ReLU	$\begin{cases} z_i, & \text{for } z_i > 0 \\ 0, & \text{else} \end{cases}$	$\begin{cases} 1, & \text{for } z_i > 0 \\ 0, & \text{else} \end{cases}$
Leaky ReLU	$\begin{cases} z_i, & \text{for } z_i > 0 \\ \alpha z_i, & \text{for } z_i \leq 0 \end{cases}$	$\begin{cases} 1, & \text{for } z_i > 0 \\ \alpha, & \text{for } z_i \leq 0 \end{cases}$
Softmax	$\frac{e^{z_i}}{\sum_k e^{z_k}}$	$\begin{cases} s_i(1-s_i), & \text{for } i = j \\ -s_i s_j, & \text{for } i \neq j \end{cases}$

Table 1: Table listing a number of common activation functions and their derivatives.

Sigmoid function

A popular activation function in neural networks, particularly for binary classification tasks, is the sigmoid function. It works well with models that produce probabilities because it converts input values to a range of 0 to 1. Even though it is widely used, the sigmoid function has a problem with the vanishing gradient when the input values are far from zero. Because gradients get closer to zero as a result, learning is restricted during backpropagation, particularly in deeper networks. Appropriate weight initialization and input scaling are advised to lessen these problems [9].

ReLU Function

One of the most popular activation functions in deep learning is the Rectified Linear Unit (ReLU). Since ReLU is non-saturating, it has the advantage of enabling models to converge more quickly than sigmoid and tanh. It may, however, have the "dying ReLU" issue, in which neurons that continuously produce zero can become inactive and stop learning[9].

Leaky ReLU

For negative input values, the Leaky ReLU function adds a slight slope to solve the dying ReLU issue. This increases learning efficiency by enabling gradients to flow even in the case of negative input, which keeps neurons active [9].

Softmax Function

Neural networks frequently use the Softmax function in their output layer to solve multi-class classification problems. It transforms raw logit scores into summable probabilities. When a model is required to predict one class from a variety of categories, this function is especially helpful. Softmax output is appropriate for cross-entropy loss training since it can be interpreted as probabilities [9].

2.11 Data

The library numpy [15] was utilized to store and arrange the data, matplotlib [16] was used to create line plots, and seaborn [17] was utilized to create heatmaps.
Our algorithm was evaluated using the scikit-learn package [18] to benchmark it in classification tasks. We used the TensorFlow [19] package for linear regression comparisons .

Franke's Function

Franke's bivariate test function is a common two-dimensional test function for interpolation and fitting. It represents the weighted sum of four exponential. It is initialized as follows:

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

where x, y $\in$ [0, 1]. For this reason, we won't scale our Franke's function data, as the input values are already scaled to a particular domain.
We create a dataset with uniformly distributed values of x, y $\in$ [0, 1] and we added a normally distributed noise $\epsilon = \mathrm{N}(0, \sigma^2)$.

Breast Cancer data

The second dataset we look at is the Wisconsin Breast Cancer data set, which can be useful for testing machine learning algorithms because it is a typical binary classification problem with a single output.

This dataset, which was created in 1995, contains detailed images that capture different tumor characteristics, making it easier to distinguish between benign and malignant cases. With 569 cases in total—212 classified as malignant and 357 as benign—the dataset offers a balanced basis for assessing classification models. It is still a useful tool for researchers looking to test and improve classification algorithms, having been gathered by the University of California, Irvine (UCI) Machine Learning Repository [20]. In order to ensure a method that is appropriate for the binary nature of the data, we modify the neural network code's cost function for our investigation to concentrate on classification analysis unique to this dataset.

3 Results and Discussion - Franke's Function

3.1 GD compared to SGD

We started by applying different optimization algorithms to a regression problem using gradient descent and stochastic gradient descent in order to compare the rates of convergence and choose the best one. We could have calculated the gradient of the cost function analytically, but we chose to use the grad function from the autograd library [21] because, in general, it is not always possible to compute the gradient expression analytically.
We decided to concentrate on a polynomial of degree 11 for our analysis. This choice was influenced by our results from Project 1, which showed that degree 11 was one of the best options for using bootstrap techniques to minimize the MSE. We hope to achieve a balance between generalizability and model complexity by choosing this degree. The risk of overfitting should be decreased by a polynomial of this degree being sufficiently flexible to capture the underlying patterns in the data without being excessively complex. We hope that this decision will allow us to maintain a robust model and produce accurate predictions.
In the Figure 4, the MSE for various optimization techniques - such as gradient descent (GD), stochastic gradient descent (SGD), ADAM, ADAM SGD, RMSprop, RMSprop SGD, AdaGrad, and AdaGrad SGD - evolves over increasing epochs for an OLS regression model. This configuration enables a controlled comparison of the convergence behavior of different approaches.

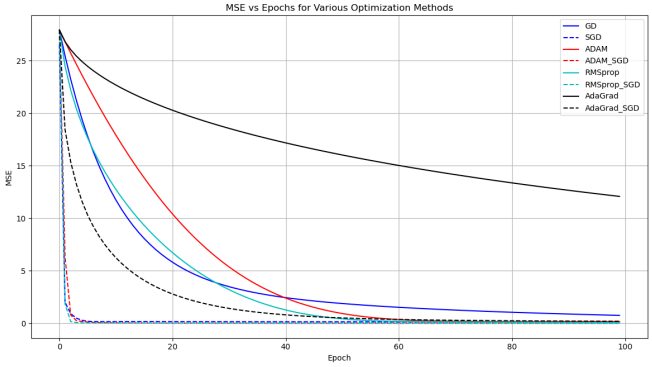


Figure 4: The graphs show how the MSE evolves as the epochs increase for different update rules in the Gradient Descent algorithm without momentum, applied to an Ordinary Least Squares (OLS) regression model. The learning rate η is fixed to 0.01, and the batch size is fixed to 10.

It appears that RMSprop SGD, SGD, and ADAM SGD reach faster the lowest MSE. Since they achieve lower MSE levels than the other algorithms, these techniques show the best long-term performance in addition to a rapid initial reduction in error.
On the other hand, plain GD, ADAM, RMSprop, and AdaGrad, the approaches that do not use SGD, display higher MSE values. These approaches might converge more slowly or set in local minimums with higher errors because they do not have the stochastic variability that SGD introduces.
Overall, the best-performing techniques with the lowest MSE are RMSprop SGD, SGD, and ADAM

SGD; the techniques without SGD show higher MSE values. This demonstrates how adding stochastic updates can improve generalization and speed up convergence.
Since it might be intriguing to investigate how the MSE behaves with different learning rates, we will specifically continue our analysis using the SGD method. By concentrating on SGD, we would like to learn more about how sensitive it is to learning rate and how this parameter affects error reduction and convergence.

As can be seen in the Figure 5, the model (using an OLS regression setup) is learning quickly at the start of training because the MSE rapidly drops within the first few epochs for all batch sizes. Following the initial sharp decline, the MSE steadily declines and stabilizes. The first epochs of the curves for the methods with smaller batch sizes (1, 5, 10) show a rapid improvement in error reduction. This implies that faster initial learning is made possible by smaller batch sizes.
The full-batch gradient descent (GD) is represented by the curve with a batch size of 400. In contrast to smaller batch sizes, the MSE declines more gradually and smoothly in this case, improving more slowly in the initial epochs. Weights are in fact only updated once every epoch, which slows convergence.

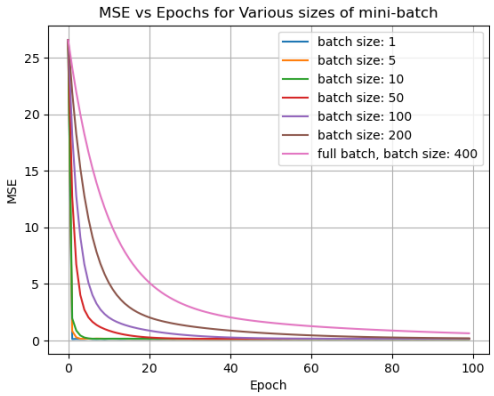


Figure 5: The line plot show how, as the size of the minibatch approaches the total number of data points (full batch, batch size = 400), the convergence rate of the stochastic gradient descent (SGD) method increasingly resembles that of the standard gradient descent (GD) method. The learning rate η is fixed 0.01.

It makes sense to proceed with the analysis using stochastic gradient descent (SGD) rather than full-batch gradient descent (GD) because smaller batches result in faster convergence. With mini-batch updates, SGD enables faster optimization and can investigate a wider range of solutions, both of which are beneficial for the model's generalization.

Stochastic Gradient Descent

Subsequently, we thoroughly investigated the possible parameters for stochastic gradient descent. Our aim was to optimize the performance of our model by identifying the optimal batch size and epoch count combination that produces the lowest MSE value.

The heatmap, shown in Figure 6, shows that the MSE reaches noticeably high values for a low number of epochs (10) and a high batch size (128 and 256); for instance, the MSE is approximately 0.5 for a batch size of 256. When we increase the number of epochs to 50 or more, however, the MSE falls to significantly lower values, all of which stay below 0.18. Based on these results, we have decided to continue our analysis with a batch size of 32 and 100 epochs because this configuration offers the best trade-off between model accuracy and training efficiency, ultimately minimizing the MSE while ensuring stable performance throughout the training phase.

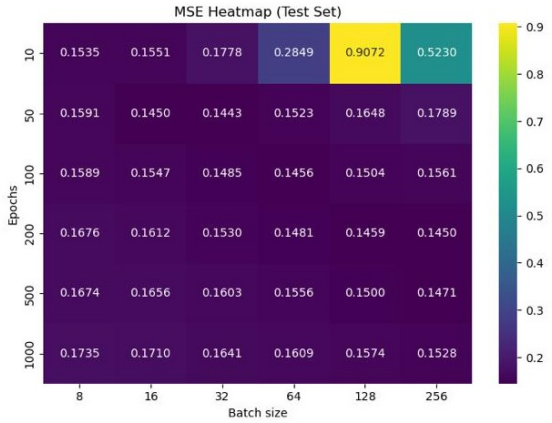


Figure 6: Heatmap of MSE for test data on the OLS model showing variation across the number of epochs and batch sizes. The learning rate η is fixed at 0.01.

Following this analysis, we investigated into the possible effects of adding a regularization parameter to our model.

In particular, we wanted to determine the potential impact of this addition on the MSE quantity. In order to constrain or 'shrink' the model parameters, we introduced this regularization parameter, which we hypothesized could result in a better MSE. In the end, this strategy might produce a more generalized model with better predictive performance by reducing problems like overfitting.

According to the heatmap shown in Figure 7, the gradient descent is unable to converge when the learning rate is 1. This lack of convergence could be explained by an overly high learning rate value, which results in an excessively large gradient step and instability throughout the optimization process.
On the other hand, the MSE stays low, ranging between 0.1 and 0.2, when lambda is set between 1 and 0.001. Nevertheless, the MSE rises to roughly 0.3 as lambda increase to 10. Consequently, we can state with confidence that in this situation, Ridge regression, with lambda ranging between 0.01 and 0.1, is a better option for minimizing MSE. By successfully reducing overfitting, we can conclude that in this situation, adding a regularization parameter can improve model performance.

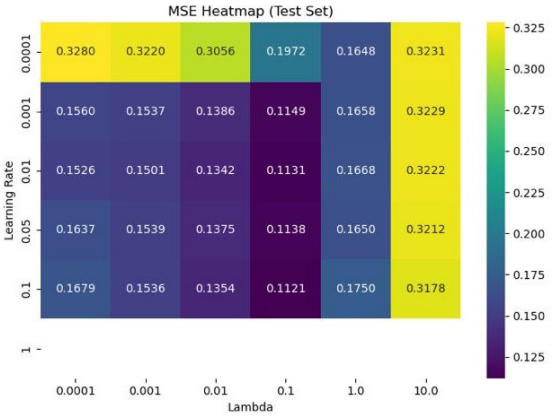


Figure 7: Heatmap of MSE for test data of the ridge model showing variation across lambda values and learning rate values, with 100 epochs and a batch size of 32

Regularization reduces MSE values by limiting the model parameters, which improves generalization to unknown data.

3.2 Neural Network

In order to assess different network parameters and configurations, including the number of nodes in a single hidden layer and the total number of hidden layers, we expanded our analysis by applying a feed-forward neural network to both the OLS and Ridge regression models. Our objectives were to determine the best configuration for obtaining the best fit to the data and to evaluate whether the Ridge regression model and the OLS model performed similarly in this neural network setting.

To find out how different activation function combinations affected model performance, we conducted experiments. In particular, three configurations were tested: (1) applying the ReLU function in the hidden layers with Sigmoid in the output layer, (2) applying the Sigmoid function in the hidden layers with Sigmoid in the output layer, and (3) applying Leaky ReLU with $\alpha = 0.01$ in the hidden layers paired with Sigmoid in the output layer.
Also, we used stochastic gradient descent with momentum of 0.9, which is often associated with improving neural network training stability and convergence speed. This could result in enhanced overall performance and more stable minima in high-dimensional data spaces [9]. Which configurations were best for minimizing MSE while maintaining model generalizability and preventing overfitting were indicated by the outcomes of this parameter tuning and activation function selection.

Sigmoid

For the purpose of showing the feed-forward neural network's performance, we first made a line plot that showed how the MSE changed with the number of nodes in network made of a single hidden layer.
We believed that the sigmoid activation function would be suitable for managing nonlinear relationships in the data, so we chose it for this model.
Our objective was to see how various node and layer configurations affected the model's capacity to reduce error, offering information about the right amount of network complexity. This method would allow us to determine the optimal parameter configurations for maintaining a balance between computational efficiency and accuracy.

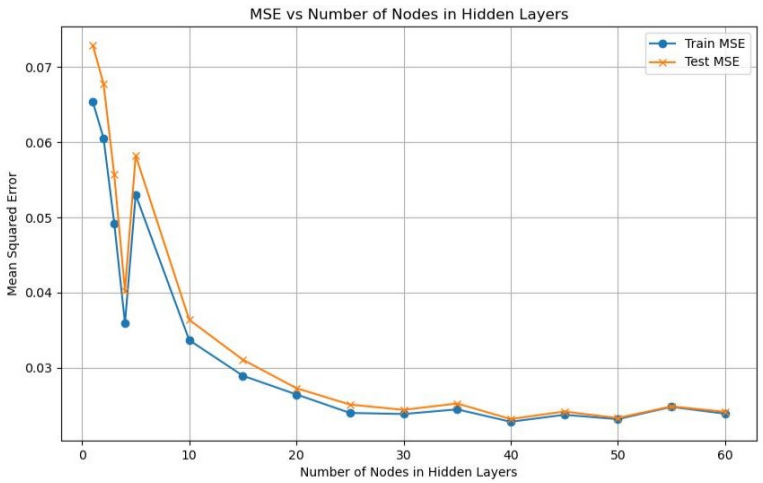
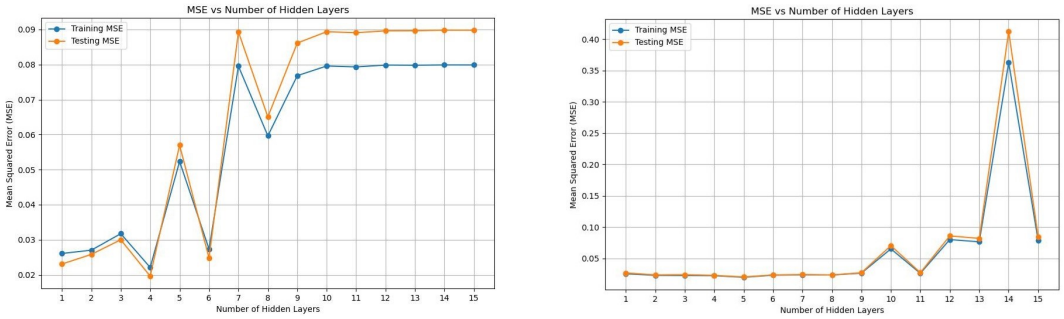


Figure 8: The graphs show how the MSE evolves as the number of nodes in one hidden layer increase, applied to an Ordinary Least Squares (OLS) regression model, with 100 epochs and a batch size of 20. The learning rate η is fixed to 0.01.

The Figure 8 makes it evident that when there are few nodes, the model is unable to sufficiently fit the data, leading to a higher MSE than in configurations with more nodes. The MSE drops significantly as the number of nodes rises, especially between 0 and 20 nodes. For models with 25–60 nodes, the MSE stabilizes at 0.01 after this. It's important to remember that choosing too many nodes may result in overfitting, a situation in which the model performs poorly on new data because it is too specific to the training set.
We expanded our analysis to look at the ideal number of layers for the neural network in order to further improve the model. We contrasted two possible configurations: a network with 20 nodes and one with 30 nodes each layer.



(a) (b)

Figure 9: The graphs show how the MSE evolves as the number hidden layer increase, applied to an Ordinary Least Squares (OLS) regression model. In the figure (a) we have 20 nodes in each layer and in the figure (b) there are 30 nodes. The learning rate η is fixed to 0.01, the number of epoch is 100 and the batch size is 30.

It is clear from both graphs that, particularly in specific configurations, the MSE increases as the number of hidden layers increases. In particular, the MSE in the model with 20 nodes per layer starts to rise rapidly as the number of hidden layers surpasses six, ultimately reaching values between 0.08 and 0.09. In this configuration, the four hidden layers show the lowest MSE, suggesting the best possible balance between error minimization and model complexity.
Across hidden layer counts ranging from 1 to 9, the MSE for the model with 30 nodes per layer stays relatively low and stable. The MSE, however, begins to rise gradually after nine hidden layers and reaches its maximum value at fourteen. According to this pattern, while the addition of hidden layers initially increases the model's capacity, too many of them cause overfitting, in which the model captures noise in the data instead of significant patterns.
These results suggest that choosing the right number of hidden layers is essential because too many layers can reduce the generalizability of the model. Therefore, in order to achieve robust performance, we will choose the configuration that minimizes MSE while avoiding overfitting.

Four hidden layers, two with 30 nodes and two with 20 nodes, were set up for the model shown in the Figure 10. The resulting heatmap shows that the MSE is at its highest when the batch size is large (128 and 256) and the number of epochs is low (10). However, even if these conditions are present, the maximum MSE is still quite low at about 0.14.
The MSE continuously decreases to lower values as the number of epochs is increased to 50 or more, with all values falling below 0.09. At 1000 epochs, the MSE is at its lowest, suggesting that additional training may marginally lower error. But for practical reasons, we have decided to continue with a batch size of 16 and a configuration of 100 epochs.
This decision strikes the ideal balance between computational efficiency and model accuracy because it successfully reduces MSE while preserving consistent performance during the training phase.

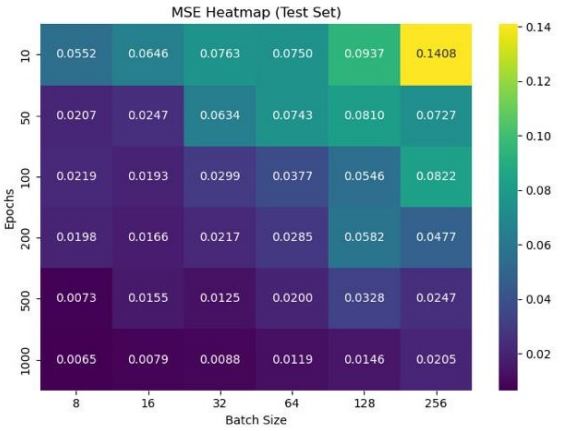


Figure 10: Heatmap of MSE for test data on the OLS model showing variation across the number of epoch and batch size. The model has 4 hidden layers of which 2 of them have 30 nodes and two of them have 20 nodes. The learning rate η is fixed 0.01.

With this setup, we should be able to concentrate on other areas of analysis since the model should

generalize well without requiring an excessive amount of training time.
Following this analysis, we investigated into the possible effects of adding a regularization parameter to our model.
The heatmap in Figure 11 suggests that the lambda parameter has a greater impact on the MSE than the learning rate does. The MSE tends to increase and steadily reaches values around 0.06 when lambda is set to values higher than 0.001. The MSE, on the other hand, decreases to approximately 0.01–0.03 for smaller lambda values, suggesting that a lower lambda value enables the model to fit the data more precisely by more successfully minimizing error. This implies that lower lambda values reduce bias by promoting a more accurate fit through less penalization.
These results generally confirm our previous decision to set the learning rate at 0.01 because it seems to achieve a good balance between enabling stable convergence and preventing overshooting. We can maintain effective model performance during training with this configuration and still obtain a low MSE.
In the neural network model, which employs the sigmoid activation function for both the hidden layer and the output layer, we compare the OLS results to those obtained with Ridge regularization. We find that the MSE values for the model without regularization tend to be lower than those for the Ridge-regularized version.

However, the difference is small and only presents at the decimal place level. This suggests that there are no important improvements in performance from adding the regularization parameter in this neural network configuration.
We might say that, given the selection of sigmoid as the activation function across layers, the neural network may already be well-suited to capture the data's structure, making additional regularization less effective. Therefore, when trained without regularization, the neural network offers adequate accuracy and robustness, efficiently capturing data patterns without the need for additional constraints.

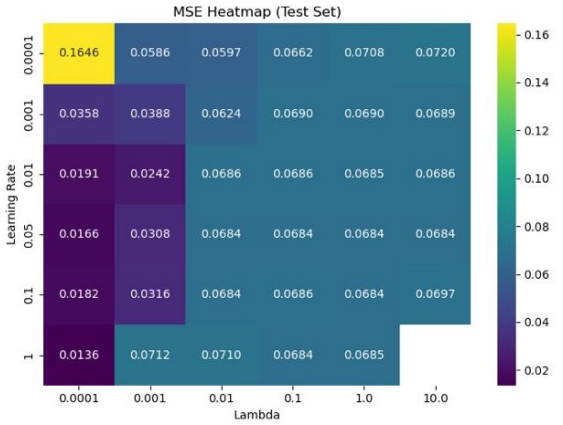


Figure 11: Heatmap of MSE for test data on the Ridge model showing variation across lambda values and learning rate values, with 100 epochs and a batch size of 16. The model has 4 hidden layers of which 2 of them have 30 nodes and two of them have 20 nodes.

ReLu

We then chose to apply the Sigmoid function to the output layer and the ReLU activation function to the hidden layers in order to examine a different neural network configuration. We wanted to see how changing the number of nodes and hidden layers would affect the model's MSE in this setup. We also wanted to see if using Sigmoid for the output layer and ReLU for the hidden layers could improve performance by better identifying non-linear patterns in the data.

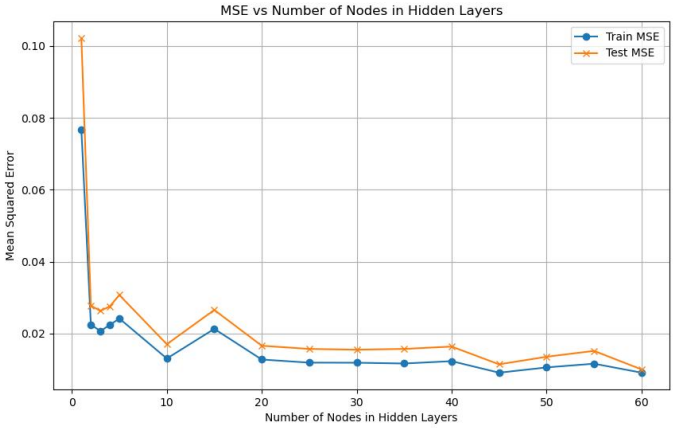


Figure 12: The graphs show how the MSE evolves as the number of nodes in one hidden layer increase, applied to an Ordinary Least Squares (OLS) regression model with the ReLU activation function applied on the hidden layer, and the Sigmoid functin applied to the output layers, with 100 epochs and a batch size of 20. The learning rate η is fixed to 0.01.

As can be seen in Figure 12 , the model finds it difficult to fit the data with fewer nodes, which leads to a higher MSE than in configurations with more nodes. The figure illustrates this in particular. As the number of nodes rises from 0 to 20, the MSE dramatically drops before stabilizing at 0.01. Although 45 or 60 nodes seem to have a slightly lower MSE, it's important to remember that too many nodes can cause overfitting. As a result, the model might perform worse on unseen data since it would capture noise unique to the training set. In order to further investigate the optimal number of hidden layers while balancing model complexity and generalization ability, we chose a configuration of 20 nodes.

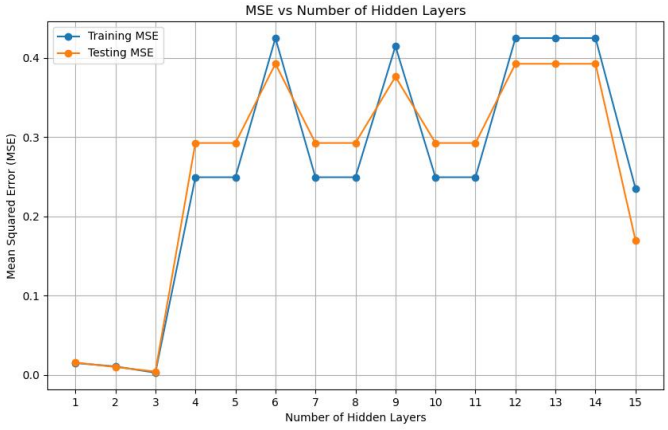


Figure 13: The graphs show how the MSE evolves as the number of hidden layers increase, applied to an Ordinary Least Squares (OLS) regression model with the ReLU activation function applied on the hidden layer, and the Sigmoid function applied to the output layers, with 100 epochs and a batch size of 20, and 20 nodes. The learning rate η is fixed to 0.01.

As the number of hidden layers increases from three to four, we see a noticeable increase in MSE in Figure 13. For configurations with four to fifteen hidden layers, values range from roughly 0.25 to 0.4 MSE. Between one and three hidden layers yield the lowest MSE values; at three layers, the MSE is almost zero. We chose three hidden layers for the remaining analysis in light

of these findings. Effective generalization to unseen data is made possible by this configuration, which strikes a good balance between reducing error and keeping a reasonable degree of model complexity.

The heatmap in Figure 14 reveals that the MSE is influenced by both the batch size and the epochs. Smaller batch sizes (8 and 16) generally achieve lower MSE values, particularly at higher epochs (500 and 1000), suggesting that smaller batch sizes may allow the model to generalize better in this configuration. In contrast, larger batch sizes (such as 128 and 256) tend to have higher MSE values across epochs. Looking at the epochs, the lower epochs, especially 10 and 50, gives higher MSE across batch sizes. Higher epochs generally gives a lower MSE, however for the lower batch sizes (8 and 16), the MSE changes marginally across epochs from 100 and above. To get the best balance between computational efficiency and model accuracy, as well as for practical reasons, we decided to investigate the effects of adding a regularization parameter with a batch size of 16, and 100 epochs.

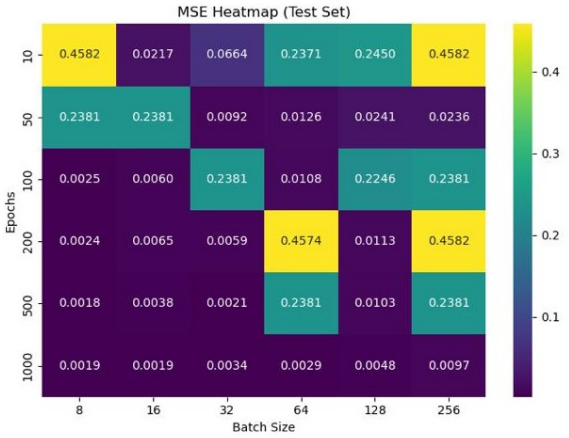


Figure 14: Heatmap of MSE for test data on the OLS model showing variation across batch sizes and epochs, with 3 hidden layers and 20 nodes. The learning rate η is fixed 0.01.

When adding the penalty term λ, we can see how the MSE error varies over different lambdas and learning rates in Figure 15. If we look at the learning rate, both the lowest (0.0001) and the highest (1) learning rates gives higher MSE across lambdas.
For the lambdas, the MSE generally decreases as the lambda decreases, except when both the lambda and the learning rate are at their lowest. In this instance, we get the maximum MSE at approximately 0.25. We can also see how the MSE is unable to converge when the learning rate is 1 and lambda in 10. The lowest MSE is 0.0043, and is given at the smallest lambda (0.0001) and when the learning rate is 0.05.

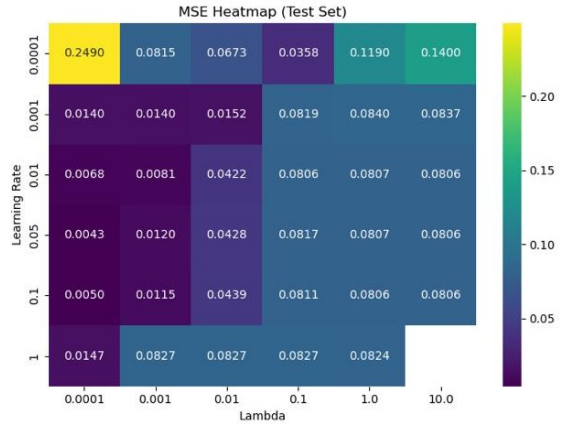


Figure 15: Heatmap of MSE for test data on the Ridge model showing variation across lambdas and learning rates, with 3 hidden layers, 20 nodes, batch size of 16 and 100 epochs.

The MSE values for the OLS model (using a batch size of 16 and 100 epochs) are slightly less than those for the Ridge-regularized version of this neural network model. The difference is only apparent at the fourth decimal place (0.006). This small variation suggests that adding a regularization parameter, like Ridge, might not significantly enhance model performance in this situation. Rather, the neural network without regularization appears to generalize adequately on this dataset, indicating that regularization might not be as important for obtaining the best MSE in these particular model architectures and training conditions.

Leaky ReLU

We then tested the feed-forward neural network using LeakyReLU as the activation function in the hidden layers.

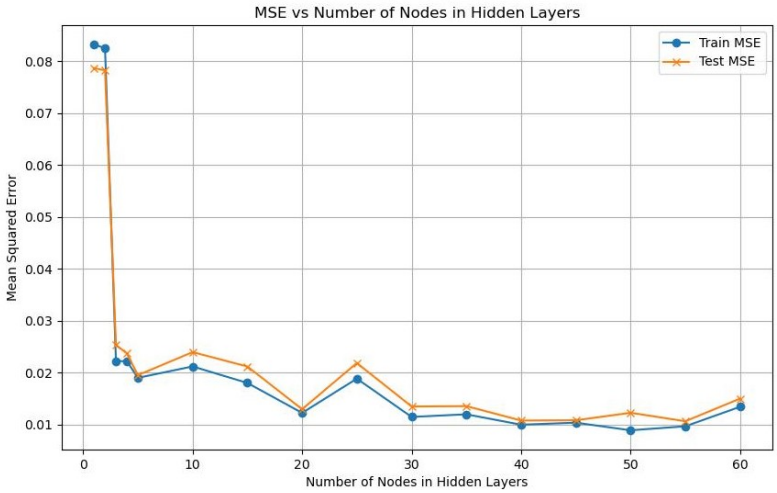
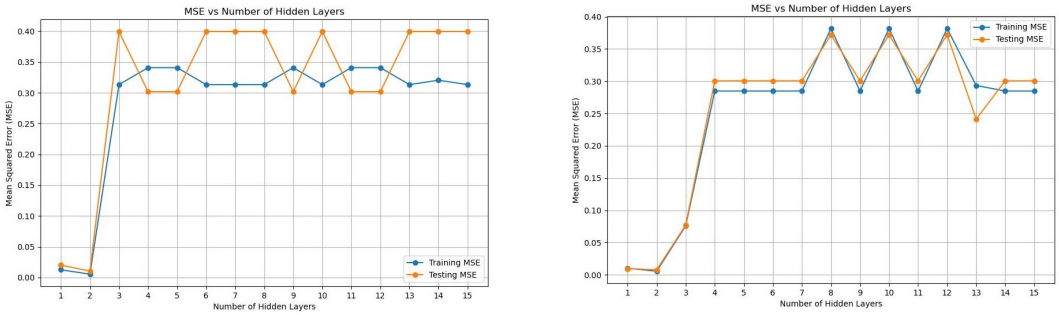


Figure 16: The graphs show how the MSE evolves as the number of nodes in one hidden layer increase, applied to an Ordinary Least Squares (OLS) regression model with the LeakyReLU activation function applied on the hidden layer, and the Sigmoid function applied to the output layers, with 100 epochs and a batch size of 20. The learning rate η is fixed to 0.01.

We started with looking at how the MSE changes with the number of nodes, as we did for the Sigmoid and the ReLU functions. In Figure 16 we can see how the number of nodes in the hidden layer increases, the MSE initially exhibits a sharp decline. This rapid reduction in MSE suggests that a small number of nodes (up to around 5) can significantly improve the model's ability to fit the training data. Beyond this point, the training and testing MSE values stabilize, with both metrics achieving their lowest and most consistent values between approximately 30 and 40 nodes. This indicates that the model reaches an optimal capacity to generalize without overfitting within this range. Increasing the number of nodes beyond 40 does not significantly reduce the error further; instead, the MSE for both training and testing shows minor fluctuations. We decided to use both 30 and 40 nodes to see how the model performs on different numbers of hidden layers.



(a) (b)

Figure 17: The graphs show how the MSE evolves as the number hidden layer increase, applied to an Ordinary Least Squares (OLS) regression model. In the figure (a) we have 30 nodes in each layer and in the figure (b) there are 40 nodes. The learning rate η is fixed to 0.01, the number of epoch is 100 and the batch size is 20.

Looking at the two graphs in Figure 17, we can see that they display a very similar pattern. Both have a low MSE (> 0.025) for 1 to 2 hidden layers, but then exhibits a steep incline from 2 to 3

layers. From that point the MSE oscillates between approximately 0.30 to 0.40, with the biggest difference between the two figures is that that the training and test lines is more aligned in figure b). As the MSE is lowest with two hidden layers, we chose to proceed with that for the next part of the analysis.

From Figure 18 we can see how, when using the OLS model, the MSE increases with lower epochs and higher batch sizes. We get the highest MSE with 10 epochs and a batch size of 256, however this MSE is relatively small, with a value of 0.1242. The lowest MSE is 0.0012, and is given when the batch size is 8 and with 1000 epochs. The differences in MSE when the batch size is 32 or below and the epochs are 100 or higher, is marginal (approximately ± 0.002). We therefore decided to again continue our analysis using a batch size of 16 and 100 epochs to get the best balance of computational efficiency and model accuracy.

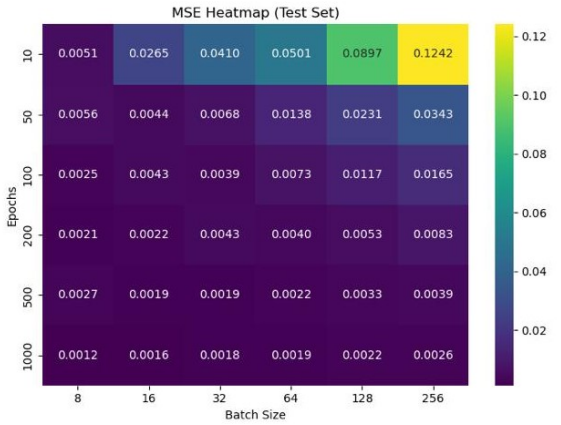


Figure 18: Heatmap of MSE for test data on the OLS model showing variation across batch sizes and epochs, with 2 hidden layers and one with 30 nodes and one with 40 nodes. The learning rate η is fixed 0.01.

We then added a regularization term to our model, and produced the heatmap in Figure 19 to see how the MSE would change with different learning rates and lambdas.

The heatmap reveals that the lowest MSE is 0.0029, and is achieved when lambda is 0.0001 and the learning rate is 0.01. This combination provides the best performance on the test set, suggesting that minimal regularization along with a moderate learning rate allows the model to capture essential patterns in the data without overfitting. The low lambda value reduces the regularization effect, permitting the model more flexibility in fitting the data, while the learning rate of 0.01 provides a stable and effective convergence rate within the 100-epoch limit.
In contrast, as the regularization strength increases to higher values, the MSE generally rises across most learning rates. This trend suggests that stronger regularization, particularly at values of lambda = 0.01, restricts the model's flexibility, potentially leading to underfitting.

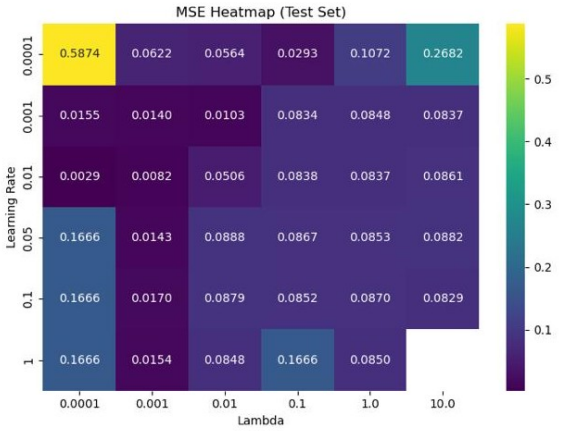


Figure 19: Heatmap of MSE for test data on the Ridge model showing variation across lambdas and learning rates, with the number of epoch is 100 and the batch size is 16.

With higher regularization, the model becomes more biased and less able to fit the underlying data patterns effectively, which is reflected in the increased MSE on the test set. We also see that when both lambda and the learning rate is high (10 and 1 respectively), the MSE is unable to converge. The impact of the learning rate is also evident. At the smallest learning rate (0.0001), MSE values tend to be higher across all lambda values, indicating that an overly small learning rate hinders convergence within the limited number of epochs. On the other hand, higher learning rates, such as 1, also lead to slightly increased MSE values in certain regularization settings, possibly due to instability in convergence. Moderate learning rates, particularly around 0.01, consistently yield

lower MSE values across various lambda values, striking a balance between convergence speed and stability.

TensorFlow

We then compared the outcomes of TensorFlow's optimized algorithms with the performance of our feed-forward neural network model, which was specifically set up with Sigmoid in the output layer and Leaky ReLU activation in the hidden layers. We decided to analyze the Leaky ReLU because, when compared to the other two activation configurations we tested, this combination of activation functions generally produced the most promising results in terms of minimizing the MSE.
TensorFlow's algorithms frequently use innovative techniques to improve stability and convergence speed, which may point out any possible difficulties with our model configuration.

In the figure 20 showing the Leaky ReLU model implemented with TensorFlow, the MSE rises, peaking at 0.06 as the number of epochs decreases and the batch size increases. On the other hand, 1000 epochs and a batch size of 8 produce the lowest MSE of 0.002.
It appears that the TensorFlow results are marginally better than those from our custom implementation of the Leaky ReLU model, which also uses stochastic gradient descent (SGD) with the same parameters. This discrepancy may result from small variations in how TensorFlow manages weight updates and data batching during training, even when the same hyperparameters are used.

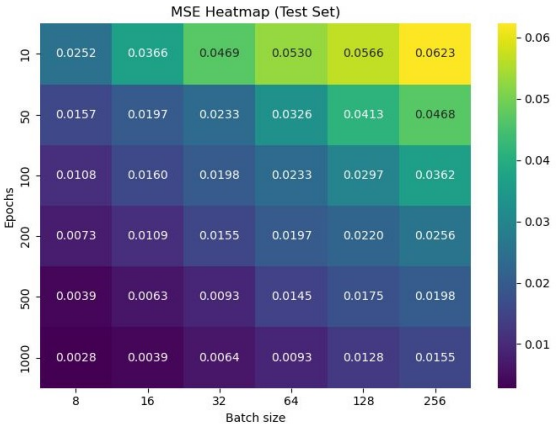


Figure 20: Heatmap of MSE for test data on the OLS model showing variation across batch sizes and epochs, with 2 hidden layers and one with 30 nodes and one with 40 nodes. The learning rate η is fixed 0.01.

The MSE in the figure of the TensorFlow-implemented Leaky ReLU model falls with increasing learning rate, and it reaches its minimum at a learning rate of 1. It's interesting to note that in this case, the lambda value seems to have little effect on the MSE. The results from our own implementation (fig. 19), where lambda had a more noticeable impact on the MSE, are in contrast to these findings. According to our analysis, learning rates ranging from 0.01 to 0.001 produced the lowest MSE. This discrepancy could be explained by the fundamental ways that TensorFlow and our method optimize parameters differently, perhaps as a result of differences in the underlying algorithms or how regularization is handled in the two frameworks.

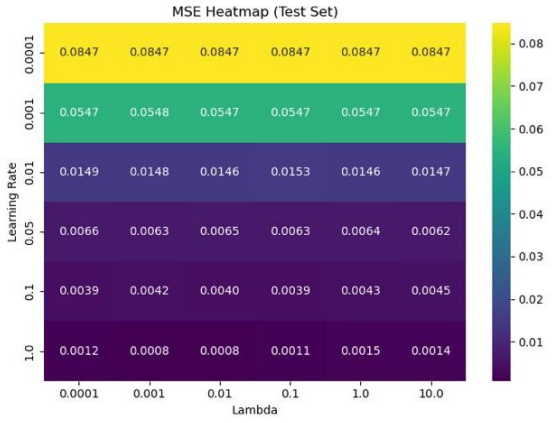


Figure 21: Heatmap of MSE for test data on the Ridge model showing variation across lambdas and learning rates, with the number of epoch is 100 and the batch size is 16.

4 Results and Discussion - Breast Cancer Data

In this section we are going to validate different methods using Gradient Descent for problems regarding classification. We are going to study the breast cancer data, which concern a binary classification problem. In order to provide consistent convergence and performance in gradient-based optimizations, we scaled the data before applying different models to this dataset.
When working with large datasets, data scaling is essential, especially for models that use gradient-based optimization techniques. In order to prevent features with wider numeric ranges from excessively affecting the result, scaling helps guarantee that each feature contributes equally to the model. Furthermore, scaling maintains a more consistent gradient, which speeds up convergence, particularly in deep learning or neural network architectures that are extremely sensitive to changes in feature magnitudes.

4.1 Logistic Regression

Since stochastic gradient descent (SGD) provides a more beneficial learning experience than more intricate optimizers like RMSprop, ADAM, or even standard gradient descent, we used it for gradient optimization in logistic regression, with momentum of 0.9. By concentrating on SGD, we can better understand how sensitive the model is to changes in the learning rate and how this parameter affects convergence speed and error reduction.
SGD is particularly helpful for big datasets because it avoids processing the complete dataset in each optimization step by gradually updating the model's parameters [9]. With each iteration, this incremental method lowers the computational cost of gradient calculations across the entire dataset, speeding up convergence. Moreover, SGD adds small variations in each update step by randomly choosing smaller data batches, which aids the model in avoiding local minima and improving generalization on unseen data. Because it strikes a balance between model performance and computational efficiency, SGD is a good fit for our dataset and modeling requirements.

The heatmap in Figure 22 shows the accuracy for our OLS SGD model on the test set. We can see how the accuracy decreases with larger batch sizes and smaller epochs, with the lowest accuracy being 0.1228, and the highest accuracy being 0.9727. However, for batch sizes of 64 and below, and epochs of 50 and above, the accuracy is relatively high (> 0.92), and only increases slightly for each decrease in batch size and increase in epochs. To still get a high model accuracy, but with less computational power, we chose to continue our analysis with a batch size of 32 and 100 epochs. There are probably different reasons for the variation in accuracy. For example, larger batch sizes might result in fewer model parameter updates, which would make the learning process less efficient.

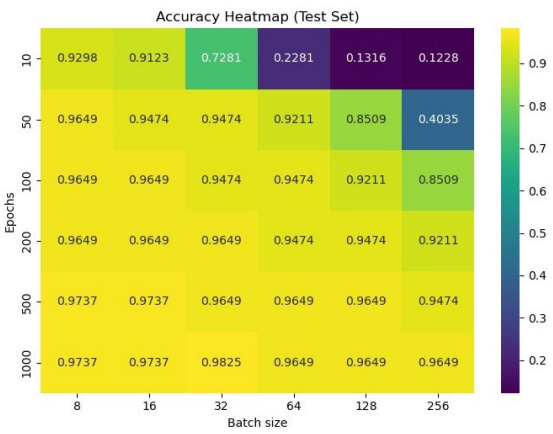


Figure 22: Heatmap of Accuracy for test data on the Logistic model showing variation across the number of epochs and batch sizes. The learning rate η is fixed at 0.01.

Furthermore, fewer epochs may not produce enough iterations for the model to converge sufficiently, which would affect how well the model performs on the test set.

When looking at the accuracy over different learning rates and lambdas in Figure 23, we can see that lower values of lambda (closer to 0.0001 or 0.001) combined with moderate learning rates (0.01 and 0.1) generally yield high accuracy scores, with values above 0.95.

As lambda increases, particularly beyond a value of 1, the model's accuracy tends to decrease, especially when paired with higher learning rates. This suggests that increased regularization penalizes the weights excessively, limiting the model's ability to fit the data well. The learning rate also plays a crucial role in determining the model's performance. The model performs optimally with learning rates of 0.01 and 0.1, where accuracy remains high across different lambda values, especially for lower regularization strengths.

A very small learning rate, such as 0.0001, results in slightly reduced accuracy, likely because the model converges too slowly to find an optimal solution within the 100 epochs. Conversely, high learning rates (e.g., 1.0) yield inconsistent performance, with accuracy dropping significantly at higher lambda values, as seen in the accuracy of 0.4035 for lambda = 10.

We then confronted our own Logistic Regression model with the one provided by Scikit-learn. We choose to analyze the confusion matrices of these two algorithms, since we believe this is the best way to visualize the performance of a classification algorithm for a binary problem.

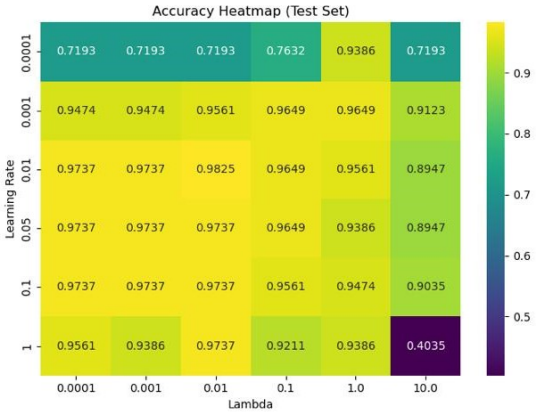
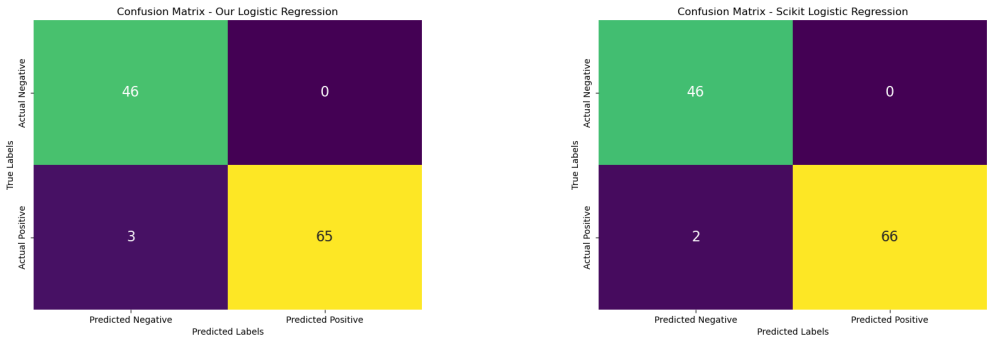


Figure 23: Heatmap of Accuracy for test data of the Logistic ridge model showing variation across lambda values and learning rate values, with 100 epochs and a batch size of 32



(a) (b)

Figure 24: The graphs show the confusion matrices of our algorithm (a) and scikit-learn algorithm (b). The learning rate η is fixed 0.05, with a total of 100 epochs and a batch size of 32.

In Figure 24 we can see how our own OLS SGD algorithm (a) and the Scikit-learn algorithm (b) preformed. Both models demonstrate strong performance in classifying the data, achieving high accuracy on both negative and positive classes. In matrix (a), our custom logistic regression model correctly classifies 46 negative samples as negative and 65 positive samples as positive. However, it misclassifies 3 positive samples as negative, resulting in a slight reduction in performance on the positive class. This yields a total of 111 correct predictions out of 114 samples, indicating high accuracy overall.
The scikit-learn model, as shown in matrix (b), performs slightly better, with only 2 positive samples misclassified as negative.
However, in the context of the Breast Cancer dataset, even such a small difference can have a big impact. False negatives, in which a cancerous tumor is incorrectly classified as benign, can have

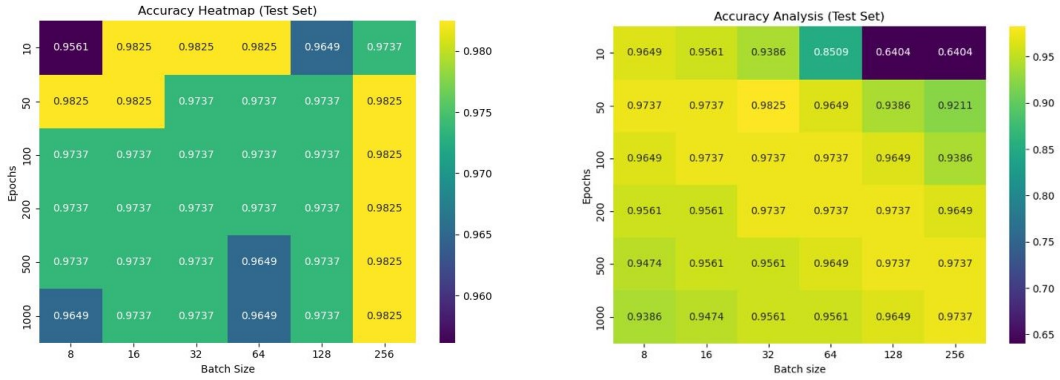
serious repercussions, such as treatment delays and a decline in patient health. This highlights how crucial it is to reduce false negatives in medical diagnostic models in order to provide immediate and effective patient care.
Like our model, the scikit-learn model correctly classifies all 46 negative samples. The slight improvement over our model can be observed in the additional correctly classified positive sample, bringing the total correct predictions to 112 out of 114. This result suggests that the scikit-learn model achieves a marginally higher accuracy than our implementation under the same learning rate, epoch, and batch size conditions.
The similarity in performance between the two models indicates that our logistic regression algorithm is comparable in effectiveness to the scikit-learn implementation, although with a small difference in classification accuracy. This small discrepancy may be due to differences in optimization methods or internal regularization handling between the two implementations.

4.2 Neural Network for classification

In order to avoid repetition with our earlier discussion of the linear regression model, where we examined the selection process for the best neural network architecture, including the number of nodes and hidden layers, we only presented the results that were most relevant to this analysis for the feed-forward neural network classification.
Three configurations of activation functions were tested in this classification context: leaky ReLU (α=0.01) for hidden layers paired with output layer sigmoid, sigmoid in both hidden and output layers, and ReLU for hidden layers with output layer sigmoid. We ran models with 100 epochs, a batch size of 32, and a learning rate of 0.05 in order to identify the best architecture for each configuration.
The best architectures that emerged were:

- Sigmoid-Sigmoid: 2 hidden layers, each with 40 nodes.
- ReLU-Sigmoid: 2 hidden layers, each with 15 nodes.
- Leaky ReLU-Sigmoid: 1 hidden layer with 15 nodes.

In the end, we discovered that the Sigmoid-Sigmoid configuration performed better than the others, maximizing accuracy and producing more consistent outcomes across a range of hyperparameter settings.
Although Softmax is frequently suggested for multi-class classification, we chose Sigmoid for the binary response because it is more appropriate for binary classification, requires less computing power, and typically performs better in this situation.
We then display the graphs which compare the output of our implementation using the Sigmoid-Sigmoid configuration with the outcomes produced by the TensorFlow implementation of the same configuration. The purpose of this comparison is to draw consideration to any variations in performance as well as any potential advantages or disadvantages specific to each implementation. By examining these graphs, we want to gain knowledge about the consistency of the outcomes between the two implementations, as well as how closely our custom model looks similar to a standard deep learning library, especially with regard to accuracy and convergence.



(a) (b)

Figure 25: Heatmap of Accuracy of our algorithm (a) and TensorFlow algorithm (b) for test data on the Logistic model showing variation across the number of epochs and batch sizes. The learning rate η is fixed at 0.01.

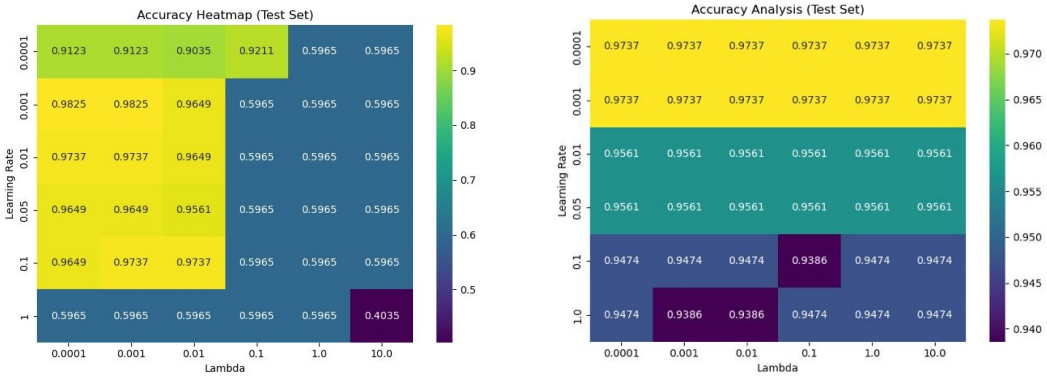
According to the figure showing our algorithm's accuracy, either a large batch size (256) or a small number of epochs (10–50) seem to yield higher accuracy values. On the other hand, minimizing the batch size and the number of epochs—that is, batch sizes of 8 and 10 epochs—provides the lowest accuracy.

TensorFlow yields results that exhibit smoother transitions between various configurations, with the highest accuracy along a "diagonal" trend. Accuracy is maximized when the number of epochs and batch size are proportionate. This pattern is clear since accuracy stays at high levels for small, moderate, and large batch sizes and epochs.

Comparing the two algorithms, TensorFlow shows two of the lowest accuracy values (with a large batch size and 10 epochs), which are even lower than the minimum accuracy found in our algorithm.

The two algorithms' different optimization strategies and how they handle batch size and epoch interactions are probably the causes of the accuracy discrepancies between them. The optimization of TensorFlow tends to stabilize outcomes across batch and epoch fluctuations. TensorFlow may be better at handling gradient updates and convergence, as evidenced by this smoother performance, particularly in proportional batch-epoch configurations where both parameters scale together. On the other hand, in certain configurations, our algorithm might not manage this interaction as well, resulting in greater accuracy fluctuations.

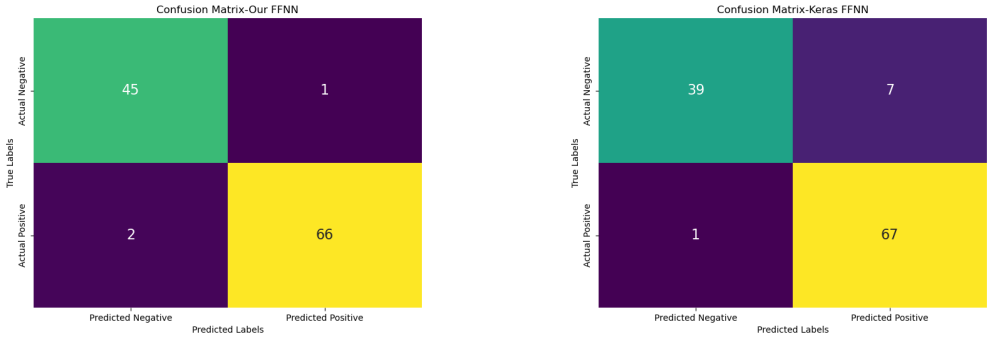
The stability of gradient updates, variations in optimization strategy, and the way each algorithm strikes a balance between regularization and convergence across various batch and epoch settings are generally the causes of these accuracy discrepancies.



(a) (b)

Figure 26: Heatmap of Accuracy of our algorithm (a) and TensorFlow algorithm (b) for test data on the Logistic model showing variation across lambda values and learning rate values, with 100 epochs and a batch size of 32.

Significant variations between the TensorFlow model's and our algorithm's outputs are shown in this figure 26. Our algorithm's maximum accuracy (0.98) occurs when the learning rate is between 0.001 and 0.1 and the lambda value is less than 0.1. However, accuracy drastically drops to about 0.59 when the learning rate rises to 1 or lambda rises above 0.1. The TensorFlow output, on the other hand, demonstrates that learning rate has a significant impact on accuracy, with the highest accuracy values occurring with lower learning rates, while lambda has little effect.
The two algorithms' different optimization mechanisms are probably the cause of this disparity in how lambda and learning rate affect them. TensorFlow frequently employs sophisticated optimization strategies, such as adaptive learning rates, which dynamically modify the learning rate during training to reduce the impact of regularization parameters like lambda. This keeps accuracy high even when lambda values are higher. Our algorithm, on the other hand, might be more sensitive to the regularization parameter and the learning rate since it uses a simpler optimization technique. Because our model lacks the adaptive flexibility to mitigate these variations, accuracy can be greatly impacted by lambda and learning rate.



(a) (b)

Figure 27: The graphs show the confusion matrices of our Neural Network (a) and TensorFlow algorithm (b). The learning rate η is fixed 0.05, with a total of 100 epochs and a batch size of 32.

The TensorFlow algorithm and the custom Neural Network model both perform well in data classification, attaining high accuracy for both positive and negative classes. Our custom logistic regression model correctly classifies 66 positive samples as positive and 45 negative samples as negative in the confusion matrix (a). It incorrectly labels two positive samples as negative and one

negative sample as positive, which marginally lowers its performance. This yields a respectable accuracy rate overall, with 111 accurate predictions out of 114 samples.
The TensorFlow algorithm, on the other hand, performs marginally worse, misclassifying 7 negative samples as positive and 1 positive as negative, as shown in confusion matrix (b).
There are a few reasons for this performance disparity. First, whereas the more intricate TensorFlow architecture runs the risk of overfitting or underfitting, the logistic regression architecture of the custom model is specifically designed for the dataset, potentially producing better decision boundaries. Second, convergence and misclassification rates may be impacted by changes in training procedures and hyperparameters, such as learning rate and regularization efficiency. Lastly, the dataset's properties, such as feature relevance and class imbalance, may also affect classification performance.

5 Conclusion

In conclusion, our research looked at both regression and classification tasks, evaluating predictive performance and efficiency using various models and optimization strategies. We discovered that using ridge regression in conjunction with stochastic gradient descent (SGD) enhanced model fit over ordinary least squares (OLS) in the regression analysis, obtaining a minimum MSE of 0.11 as opposed to 0.14 for OLS. This implies that in this situation, adding a regularization parameter can improve model accuracy. Next, we used SGD to test a feed-forward neural network with three different activation function configurations: LeakyReLU-sigmoid, ReLU-sigmoid, and sigmoid-sigmoid. OLS produced a lower MSE than ridge models, suggesting that regularization might not be required for neural networks in this situation because OLS adequately captured data patterns. With a minimal MSE of 0.0012, the LeakyReLU configuration performed better than the others; however, the variations between configurations were minimal (sigmoid-sigmoid achieved 0.006, for example). In terms of MSE, all feed-forward neural network models with SGD performed better than SGD in polynomial regression.

We optimized batch sizes, epochs, learning rates, and regularization parameters for the classification task using logistic and ridge logistic regression. The highest accuracy of 0.9825 was achieved by both models. Scikit-learn's model demonstrated a slight advantage in lowering false negatives when compared to our custom logistic regression implementation, but the difference was low. Additionally, we used feed-forward neural network models for both logistic and ridge logistic regression. Here, we found that the Sigmoid-Sigmoid activation function configuration outperformed other combinations, maximizing accuracy and yielding more consistent results across a variety of hyperparameter settings. The flexibility and efficacy of our method were demonstrated by the fact that our custom code enabled a wider range of batch size, epoch, and lambda-learning rate combinations that produced optimal results when compared to TensorFlow.

Overall, our results demonstrate how regularization may improve model performance in regression tasks. Specifically, we found that ridge regression with stochastic gradient descent (SGD) performed better than ordinary least squares (OLS) by successfully lowering the MSE. However, the advantages of regularization seemed to be limited in neural network applications, as OLS frequently produced lower MSE than ridge, particularly in configurations that used optimized activation functions like LeakyReLU-sigmoid. While neural network models combined with SGD, regardless of regularization, consistently performed well, the small variations in accuracy and MSE between different approaches suggest that multiple approaches may be practical for particular tasks.
This implies that the choice and adjustment of models may be more influenced by the intrinsic complexity of the data and model structure than by regularization alone. Future studies could investigate optimization parameters like activation functions, alternative regularization strategies, and learning rate adjustments in more detail to improve model stability, predictive accuracy, and generalizability across a range of datasets.

Compared to Project 1 [7], this project emphasizes the limited advantage of regularization for neural networks, as OLS produced a lower MSE than Ridge for these models. For Project 1, we discovered that OLS was the better fit for more complex data, such as the Stavanger terrain, but Ridge regression performed better than OLS for simpler data, such as the Franke function.
A potential limitation of our study is that we chose to use stochastic gradient descent (SGD) for the majority of the analysis instead of investigating other optimization techniques like gradient descent (GD), RMSprop, or ADAM. We chose SGD because of its excellent MSE reduction capabilities and manual learning rate control, which allowed us to see how changes in learning rate affected the MSE. In contrast to RMSprop and ADAM, which use adaptive learning rate tuning, SGD made it possible for us to test the impact of this parameter on model performance directly. We understand that this decision might be constrictive, though, as more analyses using different optimization techniques might produce better outcomes.
A more thorough investigation of these techniques may be beneficial for future research since it could provide information on how to improve model performance and achieve even lower error rates than SGD alone.

References

[1] World Health Organization, "Breast cancer," 2024. Accessed: 2024-11-04.

[2] Morten Hjorth-Jensen, "Machine learning: Lecture notes, week 34," 2024. Accessed: 2024-10-25.

[3] Morten Hjorth-Jensen, "Machine learning lecture notes, week 35," 2024. Accessed: 2024-11-04.

[4] Morten Hjorth-Jensen, "Machine learning: Lecture notes, week 43," 2024. Accessed: 2024-10-25.

[5] Morten Hjorth-Jensen, "Machine learning: Lecture notes, week 38," 2024. Accessed: 2024-10-28.

[6] Morten Hjorth-Jensen, "Machine learning: Lecture notes, week 42," 2024. Accessed: 2024-11-03.

[7] E. G. Mazzarelli, A. Milanesi, and M. M. R. Storstad, "Regression analysis and resampling methods: Project 1," tech. rep., University of Oslo, October 2024. Available at `https://github.com/annamilanesi/FYS-STK4155-2024-25`.

[8] "Plot all scaling - scikit-learn." Accessed: 2024-11-04.

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

[10] J. Friedman, T. Hastie, and R. Tibshirani, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer, 2001.

[11] Morten Hjorth-Jensen, "Machine learning: Lecture notes, week 39," 2024. Accessed: 2024-10-27.

[12] Ruman, "Understanding optimization in ml with gradient descent: Implement sgd regressor from scratch," *Medium*, 2023. Accessed: 2024-11-04.

[13] Morten Hjorth-Jensen, "Machine learning: Lecture notes, week 40," 2024. Accessed: 2024-10-27.

[14] Morten Hjorth-Jensen, "Machine learning: Lecture notes, week 41," 2024. Accessed: 2024-10-27.

[15] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[16] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in science & engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[17] M. L. Waskom, "Seaborn: Statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng,

"TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[20] W. Wolberg, O. Mangasarian, N. Street, and W. Street, "Breast Cancer Wisconsin (Diagnostic)." UCI Machine Learning Repository, 1993. DOI: https://doi.org/10.24432/C5DW2B.

[21] D. Maclaurin, D. Duvenaud, and M. Johnson, "Autograd: Reverse-mode differentiation of native python," 2015. Version 1.3, Accessed: 2024-10-31.