UiO : Department of Physics
University of Oslo

FYS-STK3155/4155 -

APPLIED DATA ANALYSIS AND MACHINE LEARNING

Classification and Regression: From Linear and Logistic Regression to Neural Networks

Authors:
Anders Rudi Bråthen Bakir
Nadia Elise Helene Ørning
Trond Victor Qian

5th November 2024

Abstract

In this project, we applied both gradient descent (GD) and stochastic gradient descent (SGD) to a regression problem. Using plain GD, we achieved a minimum mean squared error (MSE) of 0.14, while plain SGD further reduced the MSE to 0.01. Adding momentum improved GD's performance, lowering its MSE to 0.06, though it had no impact on SGD. Among the adaptive learning rate algorithms of SGD, both Adam and RMSprop produced the best results. Additionally, we implemented a neural network for a regression problem with one hidden layer and 50 nodes, testing three activation functions—Sigmoid, ReLU, and Leaky ReLU. Hyperparameters were optimized for each configuration. Leaky ReLU performed best, with an optimal learning rate of $\eta = 0.0046$ and $\lambda = 0.0002$, achieving a test MSE of 0.012 and $R^2 = 0.997$.

For the classification of benign and malignant tumors using the Wisconsin cancer dataset, we compared two custom neural network architectures with the scikit-learn MLPClassifier. The custom models were trained using SGD, while the MLPClassifier used the Adam optimizer. Both models showed high accuracy across most hyperparameters, with the best training accuracy of 1.000 for both methods and architectures. The best test accuracy was 0.991 for both models using the simpler configuration, while the complex model achieved 0.982 for the custom network and 0.991 for the MLPClassifier.

Additionally, logistic regression was used for classification, yielding a training accuracy of 0.987 for the scikit-learn method and 0.991 for the custom method. For the test data, the scikit-learn model achieved 0.991 accuracy, while the custom model achieved 0.982. A hyperparameter grid search showed that higher learning rates (η) and lower regularization values (λ) generally led to better results.

Table of Contents

1 Introduction

Neural networks have emerged as powerful tools in machine learning and data prediction. This report focuses on feed-forward neural networks and explores methods for optimizing their performance using gradient descent and backpropagation techniques.

Gradient descent is widely used for optimizing neural networks, fine-tuning the parameters to improve the model's accuracy. Backpropagation works in tandem with gradient descent by systematically updating the weights to reduce the error between predicted and actual outcomes, allowing the network to learn iteratively and become more accurate.

The aim of this project is to understand how different factors, such as the learning rate, network structure, and activation function, impact the efficiency of feed-forward neural networks. By experimenting with these variables, we aim to identify the best methods and parameters for different types of machine-learning problems.

This report is divided into six sections. The first section introduces gradient descent regression and compares different GD methods. The second section focuses on implementing our own neural network to solve regression problems. In the third section, we analyze the impact of different activation functions and parameter tuning on network performance. The fourth section shifts to binary classification, applying the network to predict outcomes like the classification of breast cancer as benign (0) or malignant (1) using the Wisconsin Breast Cancer dataset. The fifth section compares the classification results with logistic regression. Finally, the sixth section reviews all the methods used throughout the report and evaluates their overall effectiveness.

2 Theory

2.1 Gradient descent methods

To build an effective model, we aim to minimize the cost/loss/error function, denoted as **f(x)**. Gradient descent (GD) is a widely used optimization technique for achieving this by iteratively updating the model's parameters to find the values that minimize the function [1].

Gradient descent begins with an initial guess for the parameters and iteratively updates them. At each step, the parameters are adjusted in the opposite direction of the gradient of the function, ∇f(x), which indicates the steepest descent towards the minimum. The update rule is given by:

$$x_{k+1} = x_k - \eta_k \nabla C(x_k), \tag{1}$$

where: η is the learning rate, a hyperparameter that controls the size of each step, ∇C(x) represents the gradient of the cost function C(x), indicating the slope and direction of the steepest ascent at the current parameter values. Here is a step-by-step procedure for performing GD [1]:

- **Compute the gradient:** In each iteration, the gradient of the cost function is calculated $\mathbf{g} = \frac{1}{n}\nabla \sum_{i=1}^{n} C(f(x_i;\beta), y_i)$. This gradient indicates the slope of the cost function at the current parameter values, thus providing the direction and rate of the steepest increase.
- **Update the parameters:** Adjust the parameters in the opposite direction of the gradient to minimize the cost function: $\Delta\beta = -\eta\mathbf{g}$ and $\beta = \beta + \Delta\beta$. The size of each step is determined by the learning rate η. A small η can lead to slow convergence, while a very large η can cause the algorithm to diverge.
- **Repeat until convergence:** Continue iterating until the changes in the cost function are minimal, indicating that a minimum has been reached.

One significant challenge with GD is choosing an appropriate learning rate. If the learning rate is too high, the algorithm may overshoot the minimum and fail to converge. If it is too low, the process can be extremely slow. Additionally, gradient descent may find a local minimum rather than the global minimum in non-convex functions [2]. For large datasets, computing the gradient for all data points can be computationally expensive. This problem can be addressed by using Stochastic Gradient Descent (SGD), which calculates the gradient based on a random subset of the data.

2.1.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) optimizes model parameters by computing the gradient based on a small subset of the data, rather than using the entire dataset as in GD. The process involves dividing a dataset with n data points into smaller groups called mini-batches, each of size M. This results in a total of $m = n/M$ mini-batches. During each iteration, a randomly chosen mini-batch is used to calculate the gradient. The algorithm performs several cycles, or epochs, where an epoch consists of m iterations to cover the entire dataset. Here is the step-by-step procedure for performing SGD [1]:

- **Initialize the parameters:** Start with an initial guess x_0.
- **Shuffle training data:** Randomly shuffle the training dataset to avoid bias in the mini-batch selection, producing mini-batches of training examples (x_i,y_i).
- **Compute the gradient:** For each mini-batch, the gradient of the cost function is calculated $\mathbf{g} = \frac{1}{m}\nabla\sum_{i=1}^{m} C(f(x_i;\beta), y_i)$.
- **Update the parameters:** Adjust the parameters in the opposite direction of the gradient, scaled by the learning rate η: $\Delta\beta$ = -$\eta\mathbf{g}$ and $\beta = \beta + \Delta\beta$.
- **Repeat for all epochs:** Continue this process for a predefined number of epochs, making several passes over the entire training data.

SGD offers advantages over plain GD due to its efficiency. By computing gradients based on smaller subsets, the computational cost per iteration is significantly reduced, especially for large datasets. The smaller mini-batches introduce more variability in the gradient estimates, which results in noisier updates. While this causes more oscillation in the path toward the minimum, it also helps prevent the algorithm from getting trapped in local minima, improving its ability to find a global minimum.

2.1.2 Momentum

Momentum can be used to address the issue of slow convergence in gradient descent by incorporating information from previous iterations. It helps accelerate convergence by ”remembering” the direction of past parameter updates, thus maintaining momentum in the same direction. This algorithm can be written as:

$$v_i = \gamma v_{i-1} + \eta\nabla C(\beta_i) \tag{2}$$

$$\beta_{i+1} = \beta_i - v_i \tag{3}$$

where v_i is the velocity term, which accumulates the gradient updates over time, and γ ($0 \leq \gamma \leq 1$) is the momentum factor that determines how much of the previous velocity is retained [3].

2.1.3 Learning Rate Optimization

Selecting the right learning rate is crucial for the success of gradient-based optimization, but so far, we've only discussed using a constant learning rate. Typically, the learning rate is chosen through experimentation, but a fixed rate is not always ideal. In non-convex functions, for instance, there may be flat regions where large steps are needed to make progress and narrow regions where smaller steps are necessary to avoid overshooting the minimum. A constant learning rate does not adapt to these variations.

This problem can be mitigated by using adaptive learning rate algorithms that adjust the step size based on the gradient and, in some cases, its second derivative. Such methods include Adagrad, RMSprop, and Adam, which we will explore next [3].

Adagrad

Adagrad is an algorithm that adjusts the learning rate based on the accumulation of past gradients. It uses a gradient accumulation term **r**, which is updated at each iteration. The algorithm starts with $r_0 = 0$, and proceeds as follows:

$$r_{i+1} = r_i + g \odot g \tag{4}$$

$$\Delta\beta = -\frac{\eta}{\delta + \sqrt{r}} \odot g \tag{5}$$

Here, $\odot$ is the elementwise multiplication (Hadamard product), and δ is a small constant (usually set to 10^{-7}) to avoid division by zero. This process is repeated at every iteration. Adagrad automatically reduces the learning rate for parameters with large gradients and increases it for parameters with smaller gradients. This allows for faster movement along flatter regions of the function while slowing down in steeper areas[1][2]. However, since it accumulates the squared gradients over time, the learning rate can decrease too much, which may prevent it from reaching the optimal solution in non-convex functions.

RMSprop

RMSprop is a variation of Adagrad that addresses the issue of diminishing learning rates in non-convex functions. It introduces a decay factor ρ, which scales $\mathbf{r}$, helping maintain a more stable learning rate. The typical value for ρ is 0.9. The RMSprop algorithm starts with $\mathbf{r}_0 = 0$, and updates are given by:

$$r_{i+1} = \rho r_i + (1-\rho) g \odot g \tag{6}$$

$$\Delta\beta = -\frac{\eta}{\sqrt{\delta + r}} \odot g \tag{7}$$

This modification ensures that the learning rate adapts over time based on the recent gradient information, which helps navigate non-convex functions more effectively.

Adam

The Adam method combines features from both RMSprop and momentum, making it one of the most robust and widely used optimization algorithms. It utilizes two terms, $\mathbf{r}$, $\mathbf{s}$, along with the decay rates ρ_1, ρ_2, which are usually set to 0.9 and 0.99, respectively. The algorithm starts with $\mathbf{r}_0 = 0$ and $\mathbf{s}_0 = 0$, as the updates are performed as follows:

$$s_{i+1} = \rho_1 s_i + (1-\rho_1) g \tag{8}$$

$$r_{i+1} = \rho_2 r_i + (1-\rho_2) g \odot g \tag{9}$$

Adam also implements a bias correction:

$$\hat{s}_{i+1} = \frac{s_i}{1-\rho_1^i} \tag{10}$$

$$\hat{r}_{i+1} = \frac{r_i}{1-\rho_2^i} \tag{11}$$

The update is then:

$$\Delta\beta = -\eta \frac{\hat{s}}{\delta + \sqrt{\hat{r}}} \tag{12}$$

The learning rate for Adam is typically set to 10^{-3}. By incorporating both gradient information and momentum, Adam adjusts the learning rate for each parameter more effectively, making it suitable for a variety of optimization problems [1].

2.2 Feedforward Neural Network

The FeedForward Neural Network (FFNN) consists of an input layer, one or more hidden layers, and an output layer. It uses backpropagation to train the model by adjusting its parameters to

make the most accurate predictions. An example of a typical FFNN architecture is shown in Figure 2.1.

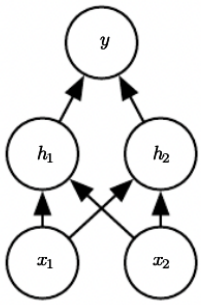


Figure 2.1: Feedforward Neural Network from Goodfellow et al.[1]. x_1 and x_2 are the nodes of the input layer, h_1 and h_2 are the nodes of the hidden layer, and y is the output layer.

The network takes a design matrix $\mathbf{X}$ as input, which is then processed by the hidden layers. In each hidden layer l, the input is multiplied by a weight matrix $\mathbf{W}_l$ and added to a bias term $\mathbf{b}_l$, resulting in:

$$z_l = XW_l + b_l \tag{13}$$

Before passing to the next layer, $\mathbf{z}_l$ is transformed using an activation function to introduce non-linearity. The activated value is given by a = f($\mathbf{z}_l$). For the output layer (when l = L), $\mathbf{z}_L$ is also processed through an activation function to generate the final output.

2.2.1 Activation functions

As discussed earlier, selecting an activation function for the hidden layers is crucial, and the choice is often determined through experimentation. The activation function in the output layer, however, depends on the type of output desired. Here, we consider three common activation functions [1].
The sigmoid function:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{14}$$

The sigmoid function maps input $\mathbf{z}_l$ to a value between 0 and 1, making it suitable for classification tasks. However, if the output saturates near 0 or 1, the derivative approaches zero, leading to ”vanishing gradients” where the weights and biases stop updating effectively, slowing down the learning process.
The ReLU function:

$$\text{ReLU(x)} = \max(0, x) \tag{15}$$

ReLU addresses the vanishing gradient issue by allowing gradients to pass through for positive values, thus improving training speed. However, it can suffer from the "dying ReLU" problem, where neurons become inactive if too many input values are negative, causing gradients to be zero and halting learning for those neurons.
The Leaky ReLU function:

$$\text{The Leaky ReLU function: Leaky ReLU(x)} = \begin{cases} x & \text{if x} > 0 \\ \delta x & \text{otherwise} \end{cases} \tag{16}$$

Leaky ReLU modifies the ReLU function by introducing a small slope δ (commonly set to 0.01) for negative input values, preventing the gradients from being zero. This helps mitigate the "dying ReLU" problem by allowing small updates for negative inputs, ensuring that learning continues even if many values are negative.

2.2.2 Backpropagation

Backpropagation is used to optimize the weights **W** and biases **b** in the hidden layers and the output layer of a neural network. It begins with random initialization of these parameters, and then iteratively adjusts them to minimize the cost function. The process starts at the output layer L and propagates backward through each hidden layer l, updating the parameters based on the error from the subsequent layer l + 1. To optimize the parameters, the weights and biases are updated using the gradient of the cost function with respect to these parameters, scaled by the learning rate. The error for the output layer L is computed by differentiating the cost function C with respect to the output z_L [4]:

$$\delta^L = \frac{\partial C}{\partial z} = \frac{\partial a}{\partial z}\frac{\partial C}{\partial a} = f^{'}(z^L)\frac{\partial C}{\partial a}, \tag{17}$$

Here, f'(z^L) is the derivative of the activation function for the output layer, and $\frac{\partial C}{\partial a}$ represents the derivative of the cost function with respect to the output activations. For the hidden layer l, the error is calculated using the error of the subsequent layer l + 1 and the weights W^{i+1}:

$$\delta^l = \delta^{l+1}(W^{l+1})^T\frac{\partial a^l}{\partial z^l} = \delta^{l+1}(W^{l+1})^T f'(z^l) \tag{18}$$

This expression propagates the error backwards through the network, taking into account the derivative of the activation function for each layer. The optimal weights and biases are found using gradient descent, which updates the parameters iteratively. The gradients for the weights in layer l are given by:

$$\nabla W^l = (a^{l-1})^T\delta^l \tag{19}$$

Where a^{l-1} is the activation from the previous layer. For the first layer, we will have the design matrix as **a**. The gradient for updating the bias in layer l is:

$$\nabla b^l = \sum_{i=1}^{n_{\text{inputs}}} \delta_i^l \tag{20}$$

This process is repeated iteratively, updating the weights and biases until the changes in the parameters are small enough, indicating that an acceptable convergence threshold has been reached.

3 Methods

The code we developed can be found in the GitHub repository https://github.com/TrondVQ/FYS_STK_P2. See README for information on how to reproduce the results in the report. Github copilot, version: 1.168.0, and ChatGPT were used to aid in the implementation of the code. The methods were implemented in Python using the libraries numpy, matplotlib, scikit-learn, autograd, itertools and seaborn. Much of the code was inspired by the lecture notes and the weekly exercises by Hjorth-Jensen, M, see: Hjort-Jensen, M. Applied Data Analysis and Machine Learning — Applied Data analysis and Machine learning. https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html.

3.1 Datasets

3.1.1 Second Order Polynomial

This second-order polynomial was used to generate the first dataset:

$$f(x) = 4x^2 + 3x + 2 + \varepsilon, \tag{21}$$

where ε represents simulated noise. The dataset consisted of 400 data points (part a) and 200 data points (part b and c), and the x values were generated from a uniform distribution between 0 and 1. The noise was generated from a normal distribution. The data was split into training (80%) and test (20%) sets using Scikit-learn's train_test_split function before training the models in parts b and c. Scaling was not applied as the dataset was already within a reasonable range for both OLS and the neural network. OLS is not very sensitive to scaling, and the neural network trained effectively without it. See code below:

```
np.random.seed(2014)
n = 200  # Number of data points
x = np.random.rand(n, 1)  # Input
y = 2 + 3 * x + 4 * x**2 + 0.1 * np.random.randn(n, 1)
p = 2
X = Design(x, p)
X_train, X_test, y_train, y_test
= train_test_split(X, y, test_size=0.2,  random_state=42)
```

3.1.2 Wisconsin Breast Cancer Data

The Breast Cancer Wisconsin (Diagnostic) dataset contains 569 instances with 30 features used for the binary classification of breast tumor malignancy. These features includes radius, area, texture, perimeter, smoothness, compactness, concavity concave points, symmetry, and fractal dimension. The features are derived from images of biopsies with values representing the mean, standard error, and worst-case scenario for each measurement. The dataset was standardized using scikit-learn's StandardScaler to ensure that each feature had a mean of 0 and a standard deviation of 1. This helped normalize the input data, ensuring that all features contributed equally during model training and facilitating faster convergence during optimization, particularly when using gradient-based algorithms like SGD [5].

3.2 Gradient Descent Regression

In this analysis, we used both analytical gradients and automatic gradients (computed using the Python library Autograd) to evaluate four different gradient descent methods: gradient descent (GD), stochastic gradient descent (SGD), and their momentum versions; gradient descent with momentum (GDM) and stochastic gradient descent with momentum (SGDM). Our goal was to compare these methods and identify the most effective approach.

We applied these optimization techniques to both Ordinary Least Squares (OLS) and Ridge regression for the second-order polynomial, Equation 21.

Additionally, we experimented with three methods for scaling the learning rate: Adagrad, RMSprop, and Adam. To systematically explore the impact of various hyperparameters, we performed a grid search and visualized the results using the Python library Seaborn, showing the performance metrics as a function of different parameter settings.

3.3 Classes and Functions

3.3.1 Evaluation and Implementation

Scikit-learn was utilized to evaluate the models developed in this project. Implementation details for the neural network methods can be found in the Scikit-learn documentation at https://scikit-learn.org/stable/api/sklearn.neural_network.html. For logistic regression, refer to the documentation at https://scikit-learn.org/1.5/modules/generated/sklearn.linear_model.SGDClassifier.html. A jupyter notebook can be found in the "Supplemental material" folder in the GitHub repository, which assesses the various classes and functions. The verification code was generated with ChatGPT.

3.3.2 Neural Network Class

The FFNN was designed as a Python class with the following parameters: cost function, cost derivative, network input size, layer output sizes, activation functions, and their derivatives. This architecture provides flexibility to tailor the model to specific datasets. Key methods include the predict method, which outputs activations for the final layer, and the cost method, which computes the cost based on inputs and targets using the chosen cost function. The _feed_forward_save method performs a feed-forward pass through the network, saving intermediate values such as z, activations, and layer inputs for use in backpropagation. The compute_gradient method calculates gradients of weights and biases, while the update_weights method adjusts these parameters based on the computed gradients, learning rate, and L2 regularization parameter. The train method uses the cost, compute_gradient, and update_weights methods to set up a training scenario that specifies the number of epochs and batches for performing stochastic gradient descent (SGD).

3.3.3 Logistic Regression Class

The Logistic Regression class was implemented with parameters including input data X, target data Y, learning rate η, regularization parameter λ, number of iterations, number of batches, and momentum term γ. It includes methods such as sigmoid, which computes the sigmoid activation, and predict, which makes predictions from input data. The predict_class method translates predicted probabilities into class labels, while the accuracy method assesses model performance. The stochastic gradient descent (SGD) method was adapted from a previous implementation for use in logistic regression. The class was compared with scikit-learns SGDClassifier

class.

3.4 Analysis

3.4.1 Part b and c

In part b of the assignment, we analyzed a second-order polynomial to compare the performance of a feedforward neural network with ordinary least squares (OLS) regression. Next, we created the design matrix with the intercept and performed a standard OLS regression using matrix inversion. The intercept was included in the model to ensure accurate fitting, as it allows the polynomial regression to account for any offset or baseline in the data[6]. The results, including the mean squared error (MSE) and R^2 values, were then printed for comparison.

Subsequently, we generated a grid of 10 learning rate values ranging from 0.001 to 0.1 and 10 lambda values between 0.00001 and 0.1. We performed a grid search by iterating over these values, conducting neural network analysis for each combination. The neural network architecture was designed for regression, featuring one hidden layer with 50 nodes and one output node. The activation functions for the hidden layer was sigmoid, while the output layer used a linear activation. Training was configured with a batch size of 10 over 100 epochs. Finally, we plotted the MSE and R^2 of the test predictions as a heatmap using Seaborn and matplotlib libraries. The best MSE and R^2 values were printed for both the training and test datasets to assess potential overfitting.

In part c, we used the implementation in part b, changing only the activation functions to ReLU and leaky ReLU.

3.4.2 Part d and e

In part d, we analyzed the breast cancer dataset in a binary classification problem. The architecture of the neural network was configured for the classification task by using the sigmoid activation for both layers and the cross-entropy cost function. The learning rates, lambda values, batch size, and the number of epochs remained consistent with parts b and c. The scikit-learn method was configured with the default adam as optimization algorithm, while the custom neural network used stochastic gradient descent (SGD). By comparing these two optimizers, the study provides a broader understanding of how the choice of optimization algorithm impacts the training dynamics and accuracy of neural network models.

For part e, we reused the same grid and applied logistic regression to analyze the breast cancer data, comparing it with the neural network implementation. The configuration was set to 100 iterations and 10 batches.

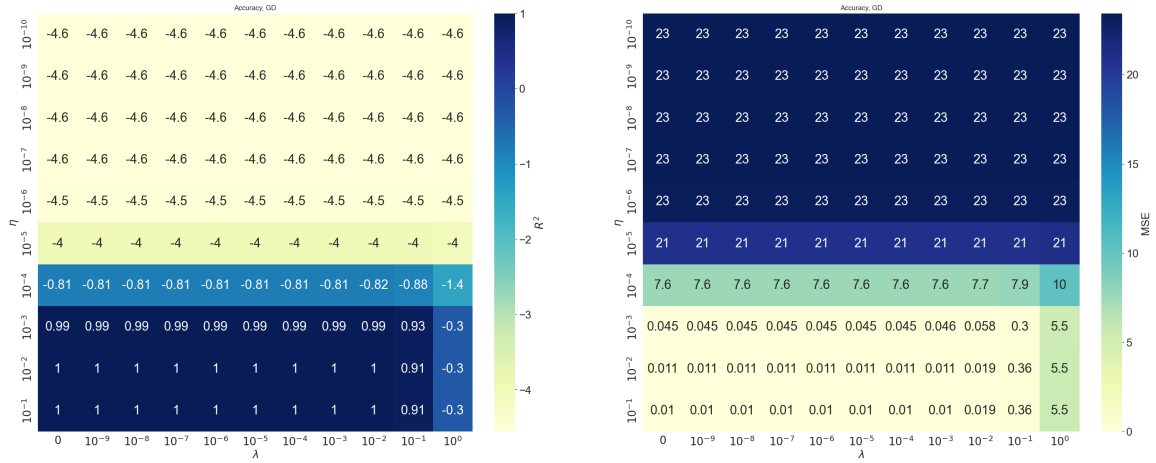
4 Results

4.1 Gradient Descent Regression

In this section, we looked at the performance of GD and SGD, both with and without momentum. Additionally, we examined the effects of three adaptive learning rate algorithms: Adagrad, RMSprop, and Adam.

4.1.1 Gradient Descent

We first implemented gradient descent, where we did a grid search to evaluate both R^2 and MSE values. This is shown in Figure 4.1a and Figure 4.1b, respectively. In this grid search, the learning rate η was plotted as a function of the hyperparameter λ. Here, the learning rate η ranged from 10^{-10} to 10^{-1}, while the hyperparameter λ ranged from 0 to 10^0. Note that when $\lambda = 0$, the model corresponds to ordinary least squares (OLS) regression, whereas non-zero λ values correspond to Ridge regression.



(a) Heatmap for R^2

(b) Heatmap for MSE

Figure 4.1: Heatmap for R^2 and MSE using plain gradient descent, with the learning rate (η) as a function of the hyperparameter (λ), for n = 400 and iterations = 2000.

The optimal learning rate η for both OLS (where $\lambda = 0$) and Ridge regression appeared to be in

the range 10^{-1} to 10^{-3}. Therefore, for further analysis of OLS regression, we chose to look at $\eta = 10^{-2}$. In addition, as the λ results for Ridge regression seemed to be stable after 10^{-3} and vastly different for 10^{0} and 10^{-1}, we decided to focus at the range $10^{-2} - 10^{-5}$ onward.

4.1.2 Gradient Descent with momentum

Next, to observe the effects of momentum, we compared the convergence rate of plain gradient descent and gradient descent with momentum for both OLS and Ridge regression, using a fixed learning rate of 10^{-2}, shown in Figure 4.2.

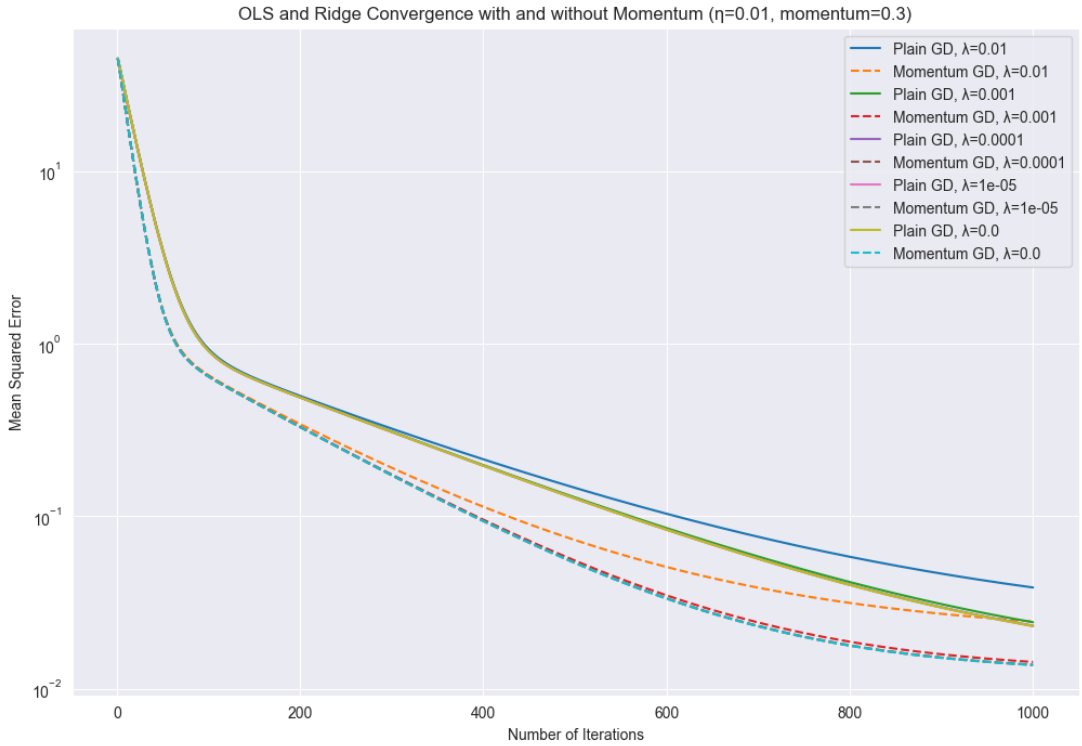


Figure 4.2: Comparing the convergence of OLS ($\lambda = 0$) and Ridge regression for various λ values using plain gradient descent (GD) and gradient descent with momentum. The learning rate was set to 0.01, with momentum = 0.3 and initial change set to 0.0.

4.1.3 Stochastic Gradient Descent

Further, we implemented Stochastic Gradient Descent (SGD). To see the effects of SGD, we compared GD and SGD where we plotted GD with momentum vs SGD with momentum, shown in Figure 4.3. In this comparison, we only looked at OLS convergence for simplicity's sake.

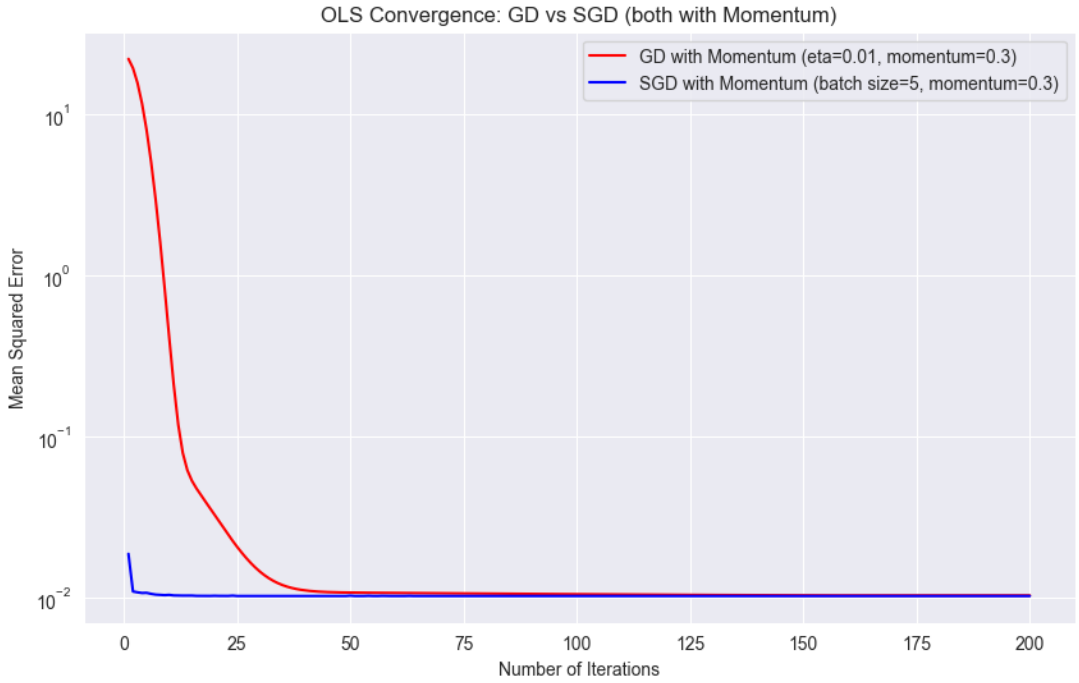
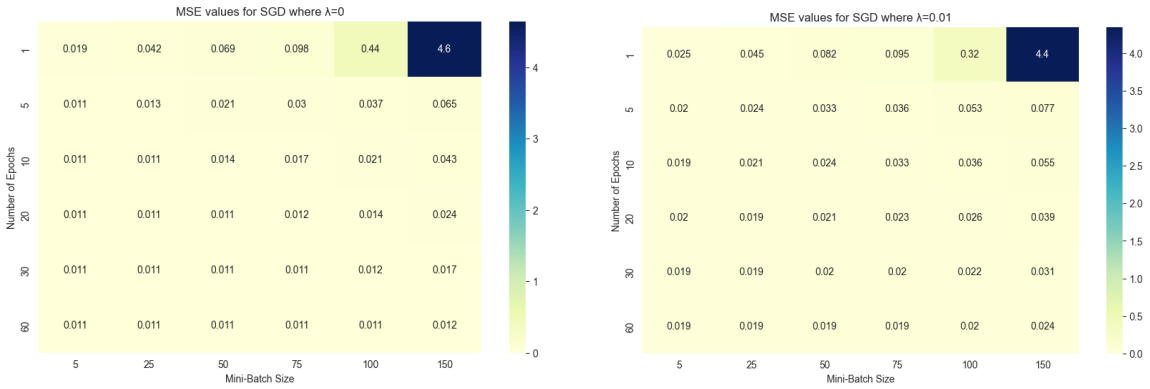


Figure 4.3: Comparing the convergence of OLS for GD and SGD, both with momentum set to 0.3 and initial change = 0. For SGD, the mini-batch size is set to 5, and for GD the learning rate to 0.01

To explore the effect of mini-batch sizes, epochs and the hyperparameter λ on SGD, we created several heatmaps, one for OLS ($\lambda = 0$) and one for each λ in the range 10^{-1} to 10^{-5}. In this result section, we show two heatmaps. One for OLS (Figure 4.4a), and one where the hyperparameter $\lambda = 0.01$ (Figure 4.4b). The rest of the heatmaps are in Appendix I.



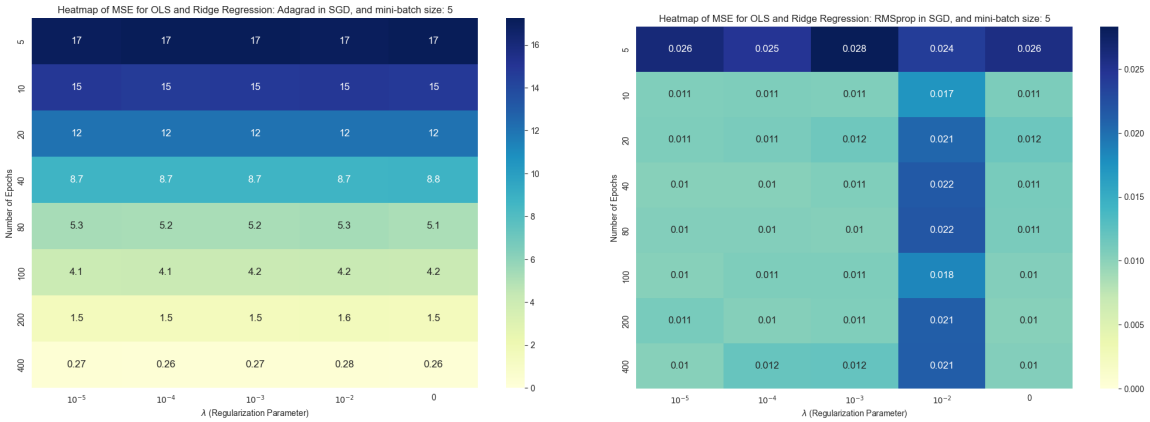
(a) OLS

(b) Ridge regression where lambda = 0.01

Figure 4.4: Two heatmaps to see the effect of the regularization term λ, mini-batch sizes and number of epochs on both OLS and ridge regression. Here the $\lambda = 0$ represents OLS and $\lambda = 0.01$ ridge regression.

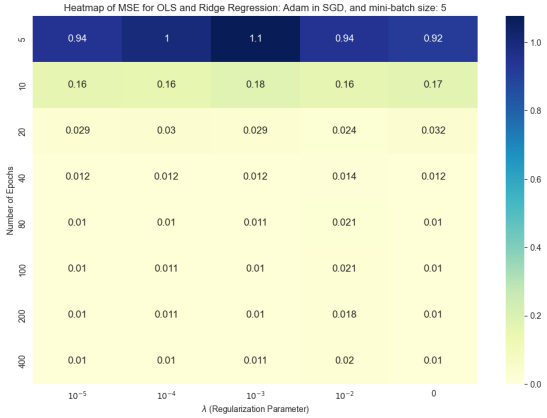
4.1.4 Adaptive learning rate algorithms: Adagrad, RMSprop and Adam

To see the effect of the adaptive learning algorithms Adagrad, RMSprop and Adam on SGD, we created three heatmaps shown in Figure 4.5a, Figure 4.5b, and Figure 4.5c. The three heatmaps show the MSE as a function of the regularization parameter λ and the number of epochs.



(a) Heatmap for Adagrad

(b) Heatmap for RMSprop



(c) Heatmap for Adam

Figure 4.5: Heatmaps showing the MSE for OLS and Ridge Regression using Adagrad, RMSprop and ADAM in SGD, as a function of the regularization parameter λ and number of epochs, with a mini-batch size of 5. When λ is 0, the results represent OLS.

4.1.5 Automatic gradient calculation: Autograd

We lastly implemented Autograd for automatic calculations of the gradients, instead of analytical, which has been used until now. In order to compare the effects of the different Gradient Descent methods, we created a table to show the results side by side (Table 4.1).

Table 4.1: MSE values for different optimization methods on OLS regression with $n = 400$ data points. Number of iterations = 400, η = 0.01, mini-batch size = 5, momentum = 0.3

Method	Analytical MSE	Autograd MSE
Plain GD	0.1425	0.045
GD with Momentum	0.0611	0.033
Adagrad GD	16.9702	16.970
Adagrad GD with Momentum	14.6350	14.636
Plain SGD	0.0106	0.321
SGD with Momentum	0.0106	0.100
Adagrad SGD	0.2599	0.259
Adagrad SGD with Momentum	0.0371	0.037
RMSprop SGD	0.0103	1.955
Adam SGD	0.0108	2.044

The table compares the MSE values for various optimization methods, showing results obtained from analytical gradient calculations and automatic differentiation via Autograd. Overall, the analytical methods yielded smaller MSE values across most optimization techniques, with noticeable differences in methods that involve stochastic gradient descent, except Adagrad.

The rest of the plots remain mostly the same and are shown in Appendix II.

4.2 Neural Network and OLS Regression

The ordinary least square (OLS) method produced the following results: training MSE = 0.0103 and R^2 = 0.9975. For the test data, the MSE was 0.0130 and the R^2 was 0.9968.

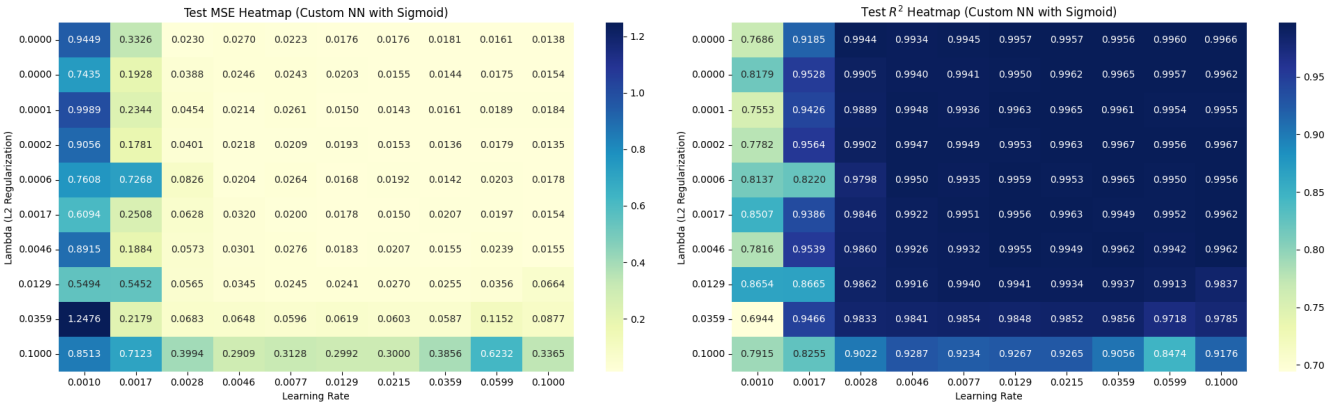


Figure 4.6: Neural Network with one hidden layer (50 nodes) and sigmoid activation: Heatmap for MSE and R^2 for different learning rates (η) and L2 regularization (λ).

Figure 4.6 shows the results on the test data of the grid search made on a simple neural network architecture with one hidden layer (50 nodes) and sigmoid activation. The model was trained for 100 epochs with a batch size of 10, applied to regression of a second-order polynomial. The minimum MSE value was 0.0107 and a maximum R^2 of 0.9974 for the train data, while the test data gave a minimum MSE of 0.0138 and maximum R^2 of 0.9966. The hyperparameters show better result with a higher learning rate (η) and a low lambda (λ), specifically $\eta = 0.1$ and $\lambda = 0.00001$, yielding best results for the training data, and $\eta = 0.1$ and $\lambda = 0.00022$ yielding best results for the test data.

4.3 Testing Different Activation Functions

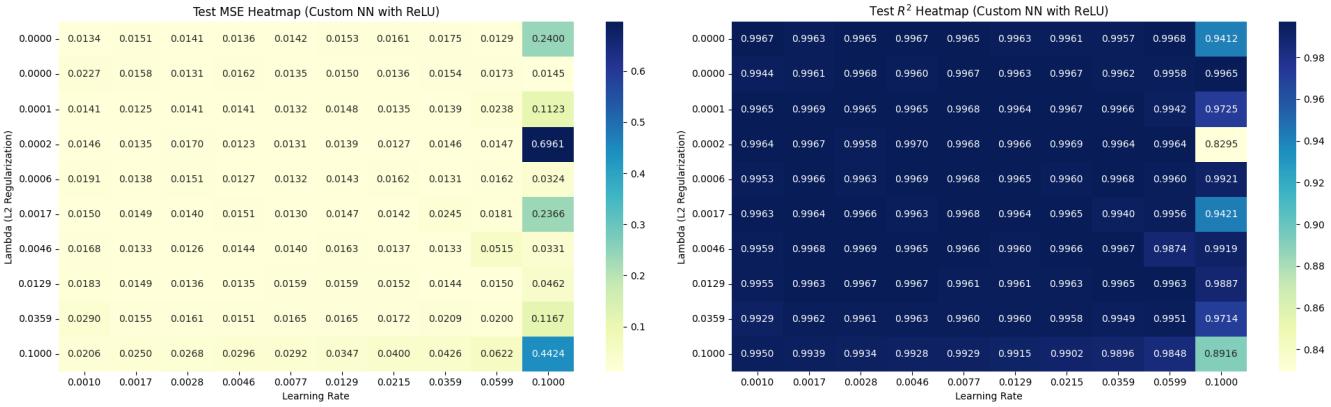


Figure 4.7: Neural Network with one hidden layer (50 nodes) and ReLU activation: Heatmap for MSE and R^2 for different learning rates (η) and L2 regularization (λ).

Figure 4.7 shows the results of the grid search conducted on a simple neural network architecture with one hidden layer (50 nodes) and ReLU activation. The model was trained for 100 epochs with a batch size of 10, applied to the regression of a second-order polynomial. The heatmaps show a minimum MSE of 0.0101 and a maximum R^2 of 0.9976 for the training data, while the test data yielded a minimum MSE of 0.0123 and a maximum R^2 of 0.9970.

For the training data, the best hyperparameters with respect to the lowest MSE and highest R^2 were found to be $\lambda = 0.000599$ and $\eta = 0.0359$. For the test data, the best hyperparameters for the lowest MSE and highest R^2 were $\lambda = 0.000215$ and $\eta = 0.00464$.

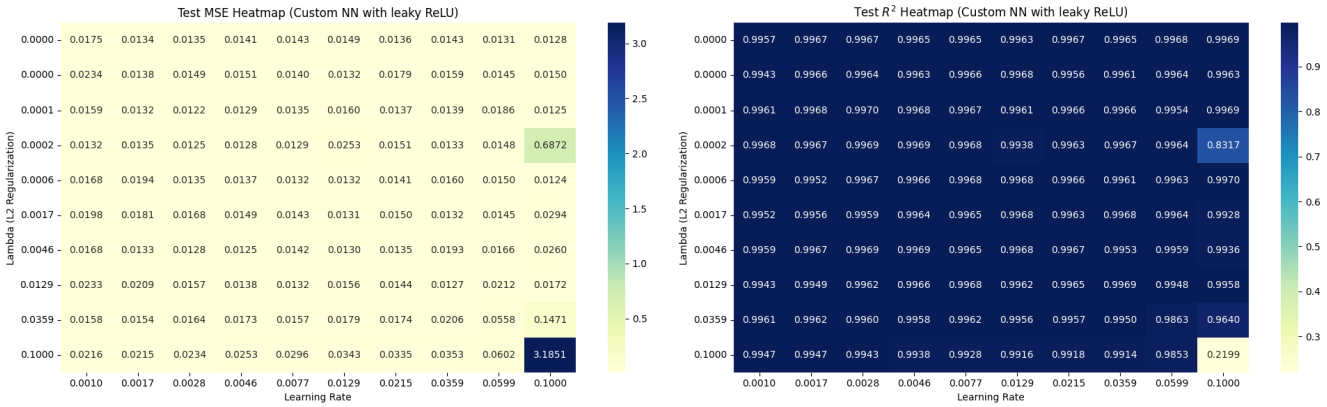


Figure 4.8: Neural Network with one hidden layer (50 nodes) and Leaky ReLU activation: Heatmap for MSE and R^2 for different learning rates (η) and L2 regularization (λ).

Figure 4.8 shows the results of the grid search conducted on a simple neural network architecture with one hidden layer (50 nodes) and Leaky ReLU activation. The model was trained for 100 epochs with a batch size of 10, applied to the regression of a second-order polynomial. The heatmaps reveal a minimum MSE of 0.0122 and a maximum R^2 of 0.9970 for the training data, while the test data yielded a minimum MSE of 0.0123 and a maximum R^2 of 0.9970.

For the training data, the best hyperparameters with respect to the lowest MSE and highest R^2 were found to be $\lambda = 0.0000774$ and $\eta = 0.00278$. For the test data, the best hyperparameters for the lowest MSE and highest R^2 were $\lambda = 0.000215$ $\eta = 0.00464$.

The optimized parameters for the test data are presented in Table 4.2.

Table 4.2: Performance Evaluation of OLS and Neural Networks for Regression: Optimized Parameters for test data, Learning Rate (η), Lambda (λ), MSE, and R^2

Method	Learning rate (η)	Lambda (λ)	MSE	R^2
OLS	NA	NA	0.013	0.997
NN sigmoid	0.1	0.0002	0.014	0.997
NN ReLU	0.0046	0.0002	0.012	0.997
NN Leaky ReLU	0.0046	0.0002	0.012	0.997

4.4 Neural Network Classification

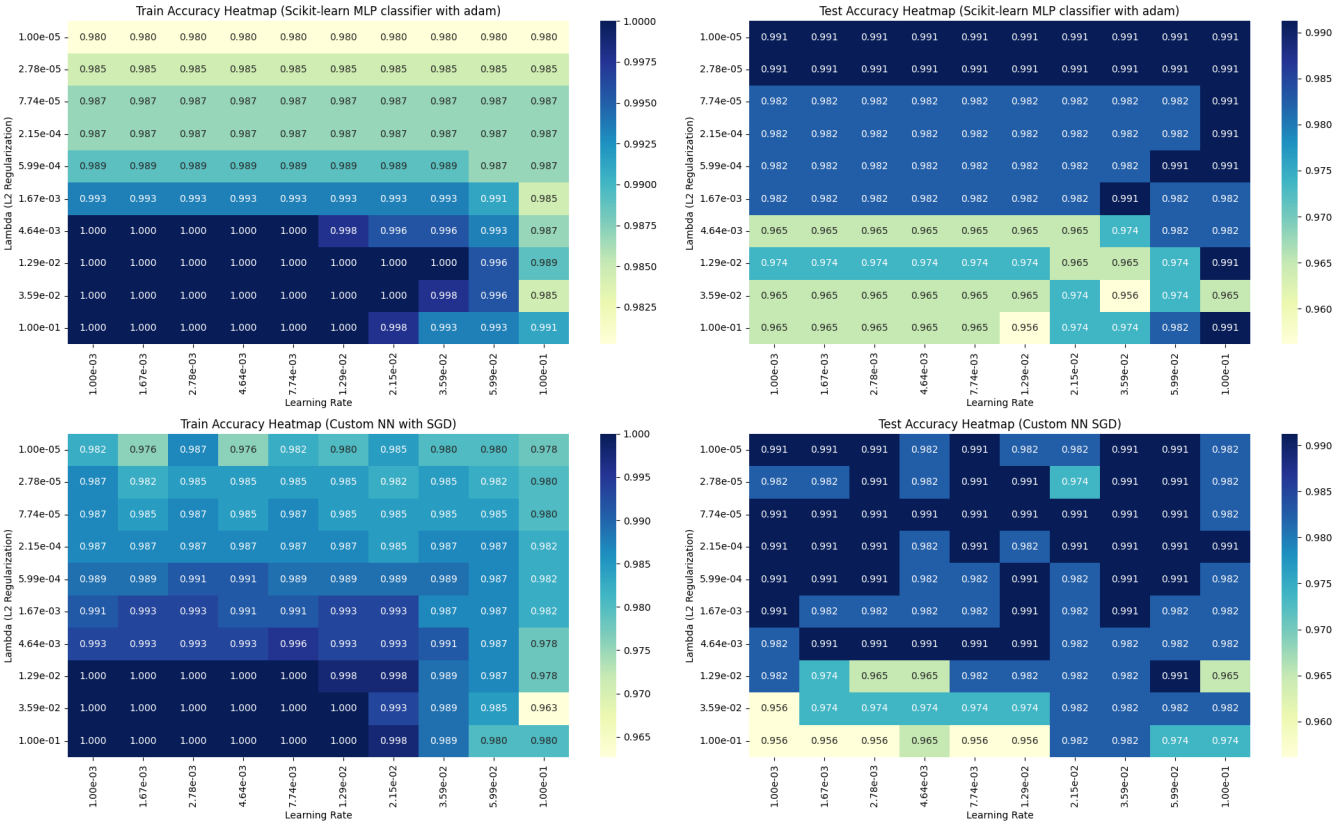


Figure 4.9: Neural Network with one hidden layer (50 nodes) with sigmoid activation. Heatmap with accuracies for different learning rates (η) and L2 regularization (λ) for both training and test data.

The models were trained for 100 epochs with a batch size of 10 for the custom neural network, applied to binary classification of the winsconsin breast cancer data. The heatmaps in Figure 4.9 show a maximum training accuracy of 1.000 and a test accuracy of 0.991 for both the scikit-learn MLPClassifier and the custom neural network. Both methods exhibit a similar

pattern, with better accuracy associated with higher lambda values and lower learning rates for the training data. However, for the test data, this pattern reverses, with lower lambdas and higher learning rates yielding better results, particularly for the scikit-learn method. The test accuracy for the custom neural network remains relatively uniform, with most hyperparameter combinations resulting in accuracies of either 0.991 or 0.982.

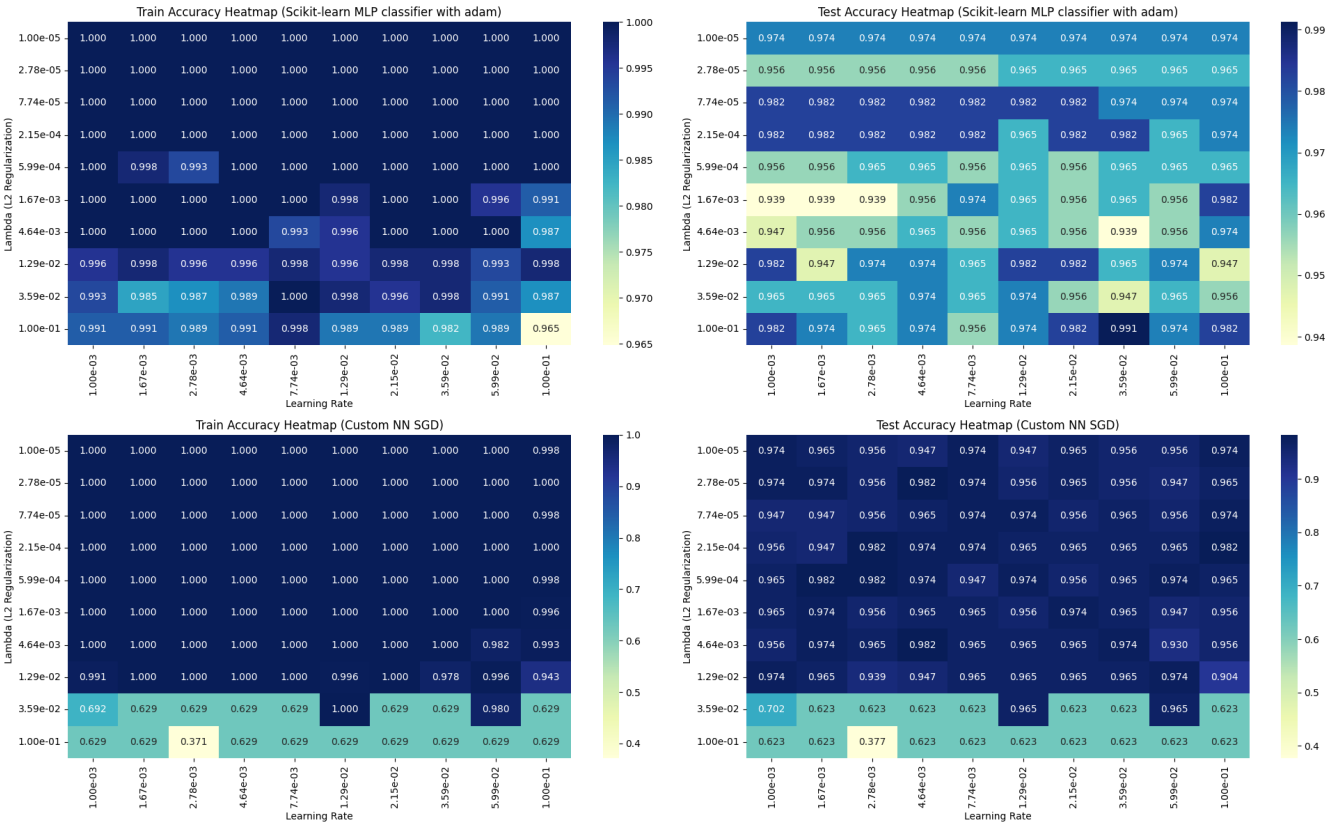


Figure 4.10: Neural Network with five hidden layers with ReLU activation for the hidden layers and sigmoid for the output layer. Heatmap with accuracies for different learning rates (η) and L2 regularization (λ).

Figure 4.10 presents the results of a more complex configuration of the neural network and the MLPClassifier. The networks was set up with five hidden layers (50, 20, 10, 30, 40 nodes) and ReLU activation for the hidden layers and sigmoid activation for the output layer. The model was trained for 100 epochs with a batch size of 10 for the costum neural network, applied to binary classification of the winsconsin breast cancer data. The training accuracy for both methods is predominantly at 1.000, particularly for the custom neural network. Higher values of λ lead to a decrease in accuracy, with the custom neural network reaching a minimum accuracy of 0.371.

For the test accuracy, the scikit-learn method achieved a maximum accuracy of 0.991, with optimal hyperparameters of $\lambda = 0.1$ and $\eta = 0.0359$. In contrast, the custom neural network dis-

played consistent test accuracy across most hyperparameter combinations, with values primarily between 0.956 and 0.982, showing slightly better performance at $\lambda = 0.0000774$.

4.5 Logistic Regression

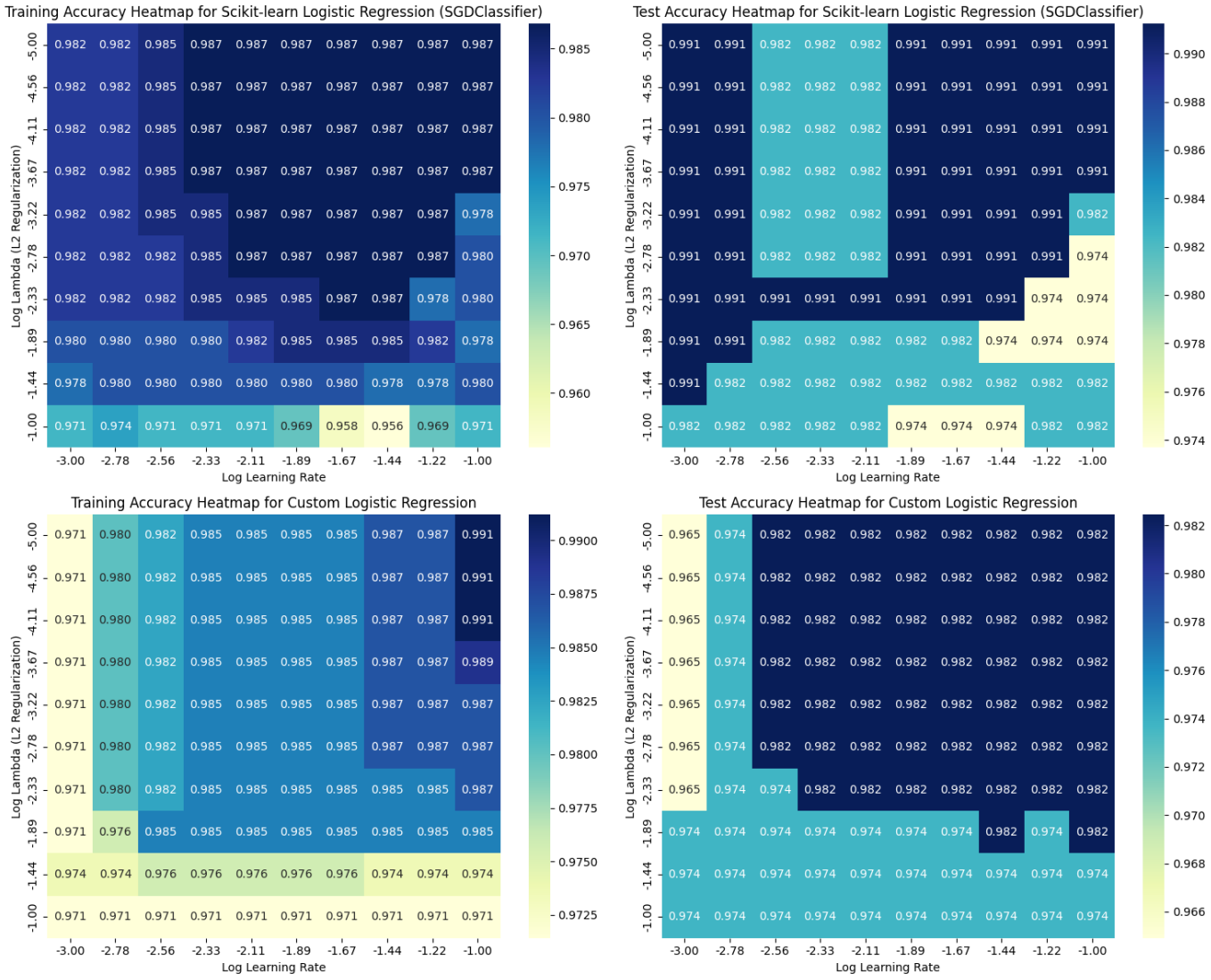


Figure 4.11: Logistic Regression: Heatmap with accuracies for different learning rates (η) and L2 regularization (λ).

Figure 4.11 shows the results of the grid search conducted on logistic regression, applied to binary classification of the Wisconsin breast cancer dataset. The model was trained for 100 epochs with a batch size of 10. The maximum training accuracy achieved was 0.987 for the scikit-learn method and 0.991 for the custom logistic regression method. Both methods demonstrated a similar pattern, with higher accuracy associated with lower λ and higher learning rates (η).

For test accuracy, the scikit-learn method reached a maximum of 0.991, while the custom logistic regression achieved 0.982. Again, higher η and lower λ resulted in improved accuracy. However, the scikit-learn method was more sensitive to higher learning rates, whereas the custom method was more sensitive to lower learning rates.

5 Discussion

5.1 Gradient Descent Regression

In this section, we analyzed the performance of the different GD methods in optimizing the linear regression models OLS and Ridge regression.

5.1.1 Plain Gradient Descent

The analysis presented in Figures 4.1a and 4.1b illustrates the performance of Gradient Descent (GD) with different learning rates (η). Specifically, GD performs exceptionally well when η = 0.1 and η = 0.01, with an MSE value of 0.01 and R^2 score of 1 for OLS (with λ = 0) and Ridge regression within the range $\lambda = [10^{-9}, 10^{-3}]$. These results are as expected as having a too small learning rate results in slow convergence. Regarding the regularization parameter λ, its impact was minimal except at higher values ($\lambda \geq 10^{-1}$). This suggests that for Ridge regression, a small regularization parameter does not significantly affect the model's ability to fit the data, while for a larger regularization parameter, it can introduce bias, penalizing large coefficients and potentially underfitting the model.

The momentum term incorporates information from previous iterations, helping to accelerate convergence. This is shown in Figure 4.2, where GD with momentum converged faster than plain GD for both OLS and Ridge regression. This effectively reduces the iterations required to reach a stable MSE. The results also indicated that the regularization parameter λ had minimal effect on convergence speed when momentum was used. This indicates that the penalty of large coefficients that λ introduces is minor compared to the influence of momentum on accelerating convergence.

5.1.2 Stochastic Gradient Descent

Following our analysis of gradient descent (GD), we looked at Stochastic Gradient Descent (SGD).

SGD differs from GD by updating parameters using a subset of data (mini-batches) rather than the entire dataset. Figure 4.3 compares the convergence of OLS regression using GD and SGD, both with momentum. SGD converged significantly faster in terms of iterations and achieved a stable low MSE earlier than GD. This is attributed to the frequent updates in SGD, which allow the optimizer to make quicker progress toward the minimum. Figure 4.3 aligns well with typical expectations for GD and SGD with momentum.

Similar to our analysis of GD, for SGD we looked at the effect of the hyperparameters of batch size, epochs, and regularization. Figure 4.4 illustrates the impact of batch size and epochs on the MSE for OLS and Ridge regression. Smaller batch sizes (e.g. 5, 10) led to faster convergence and lower MSE values compared to larger batch sizes (e.g. 100, 150). This is because smaller batches provide more frequent parameter updates, allowing the model to adjust more rapidly. Smaller batches can however introduce higher variance in the gradient estimates, which may cause the loss function to fluctuate more during training. Larger batches produce more stable gradient estimates, but may require more epochs to converge due to less frequent updates. Increasing the number of epochs generally improved model performance up to a point, after which the gains diminished. This suggests that after sufficient training, the model parameters stabilize, and additional epochs show diminishing returns in the reduction of MSE. Likewise with the other algorithms, the regularization term, λ shows an increase in MSE compared to OLS. The bigger the regularization term, the bigger the difference. This is as expected.

5.1.3 Adaptive learning rate algorithms: Adagrad, RMSprop and Adam

Adaptive learning rate algorithms adjust the learning rate during training based on the historical gradients. Figure 4.5 presents heatmaps showing the MSE for OLS and Ridge regression using Adagrad, RMSprop, and Adam, respectively.

In Figure 4.5a we can see that Adagrad converged more slowly compared to RMSprop and Adam and did not converge even after 400 epochs. Adagrad adapts the learning rate by dividing it by the square root of the accumulated squared gradients, and therefore may have reduced the learning rate too much over time. The high MSE at lower epochs suggests that Adagrad may be less suitable for this regression task due to its conservative adjustment of the learning rate, particularly for convex functions where larger learning rate adjustments might be beneficial.

RMSprop modifies Adagrad by introducing a decay factor to limit the accumulation of past gradients, preventing the learning rate from becoming too small. The effect of this can be shown in Figure 4.5b where RMSprop converged quickly reaching low MSE values by 100 epochs and maintaining stable results across different values of the regularization parameter λ.

Adam combines the ideas of momentum and RMSprop by maintaining running averages of both the gradients and their squares. As shown in 4.5c, Adam also converged quickly performing similarly to RMSprop, where it reached low MSE values by around 80 epochs, with little variation across the λ values.

5.1.4 Automatic gradient calculation: Autograd

Table 4.1 shows that methods that involve SGD have a noticeable difference when it comes to analytical gradient and Autograd. Analytical gradients provide exact calculations, while Autograd, though convenient, may introduce slight numerical inconsistencies. These minor discrepancies in gradient calculations likely contributed to the observed differences in MSE, especially in methods sensitive to precise gradient updates, such as those involving adaptive learning rates and momentum. Adagrad, however, is less affected due to its accumulation of squared gradients over time, which stabilizes the learning rate and smooths out minor discrepancies, making it more resilient to small numerical differences.

5.2 Neural Network Regression and OLS

The ordinary least square method produced good results, with a train MSE of 0.0103 and R^2 of 0.9975. For the test data, the MSE was 0.0130 and R^2 was 0.9968 (Table 4.2). The model shows good predictive power and generalization. This level of performance is expected for a fit of a simple second-order polynomial, demonstrating how model simplicity can work to our advantage, when the problem allows it.

The simple neural network with one hidden layer containing 50 nodes, using sigmoid activation for the hidden layer and a linear activation for the output layer, performed well in regression tasks. The choice of a linear output activation was made to allow the model to fit a continuous function. The weights were initalized randomly from a normal distribution following He normal initialization [7]. This was done to fascilitate fast learning. The biases were initialized to zeros for simplicity, ensuring an unbiased initialization. Figure 4.6 shows stable model performance across most η and λ combination. The difference in train and test results was minimal, indicating good generalisation capabilities. The highest learning rate (η) in the grid 0.1 gave the

poorest results for both MSE and R^2, while changes in regularisation (λ) did minimal to model performance, although the highest $\lambda = 0.1$ did reduce some performance across most learning rates.

The heatmap for the MLPRegressor with the same configuration, as shown in Appendix III, displays similar results.

5.3 Testing Different Activation Functions

Choosing the right activation function is important in tailoring the architecture to the problem at hand. The results for the ReLU activation showed slightly better performance than the sigmoid activation, with MSE = 0.0123 and $R^2 = 0.9970$ for the test data with hypermarameters $\eta = 0.00464$ and $\lambda = 0.000215$ (Figure 4.7, Table 4.2). The train data showed slightly better fit with MSE = 0.0101 and $R^2 = 0.9976$ with the hyperparameters $\eta = 0.0359$ and $\lambda = 0.000599$. The results show that the model generalized well and has strong predictive power. However, the difference in hyperparameters may be a sign that the model is trying to adjust for potential overfitting. Cross validation may be necessary to investigate this further. Overall the model performed well for all the λ values and most learning rates with MSE less than 0.02 and R^2 over 0.99. The exception was $\eta = 0.1$ which had varying results for different λs getting as low as R^2 0.8295 and as high as MSE = 0.6961 for $\lambda = 0.0002$. This may indicate that a threshold was reached for the learning rate in relation to the dying ReLU effect as explained in the theory section of the report.

The Leaky ReLU activation gave similar, good results (Figure 4.8, Table 4.2). Although for the train data the best MSE and R^2 was slightly worse with MSE = 0.0122 and $R^2 = 0.9970$ while for the test data it stayed the same as for ReLU activation. The heatmaps show stable results across most λs and learning rates, even at $\eta = 0.1$. This may be due to the fact that leaky ReLU is designed to mitigate the issue of dead nodes caused by the dying ReLU effect. However, with $\eta = 0.1$ and $\lambda = 0.1$ we get drastically worse results with MSE = 3.1851 and $R^2 = 0.2199$. At $\lambda = 0.0002$ and $\eta = 0.1$ the MSE is 0.6872 and $R^2 = 0.8317$. The results show a similar performance to ReLU activation with better performance at higher learning rates and across more λ values.

5.4 Neural Network Classification

The results for the simple neural network with one hidden layer (50 nodes) and sigmoid activation on the classification problem, is shown in Figure 4.9. The network performance on a

classification problem is generally very good with a maximum accuracy of 1 on the training data and 0.991 on the test data for both scikit-learn MLPClassifier (with Adam) and the custom neural network (with SGD). While the overall performance was good, the best results for the training data were obtained with a high λ and low learning rate. For the test data this was reversed, showing better results with lower λs and higher learning rates. This may indicate slight overfitting, where the model adjusts the hyperparameters to better fit the test data, leading to improved test performance at the expense of training performance. Additionally, the Scikit-learn method consistently yielded good results across all learning rates for the two lowest values of λ.

Figure 4.10 shows the results of a more complex configuration of the neural network and the MLPClassifier. These networks are configured with five hidden layers with ReLU activation and sigmoid for the output layer. The training accuracy for both methods demonstrates high consistency, with many values equal to 1.000, especially for the custom neural network. However, higher values of λ significantly reduce accuracy, with the custom neural network dropping as low as 0.371. This is to be expected with a more complex model and shows the regularization driving down the accuracy.

In terms of test accuracy, the scikit-learn method yielded varied results, ranging from 0.939 to 0.991 across all hyperparameter configurations, with the highest accuracy of 0.991 achieved at optimal hyperparameters of $\lambda = 0.1$ and $\eta = 0.0359$. In contrast, the custom neural network showed a similar distribution of results across most hyperparameter combinations, with test accuracies predominantly ranging from 0.947 to 0.982, and a slight preference for $\lambda = 0.0000774$. However, for the two highest λ values, the custom model performed poorly, with accuracy dropping as low as 0.377.

5.5 Logistic Regression

The logistic regression with stochastic gradient descent (see Figure 4.11) shows good performance on the classification problem with a best train accuracy of 0.987 for the Scikit-Learn method and 0.991 for our logistic regression implementation. The scikit-learn method is more stable across the hyperparameters, with lower λ values and higher learning rates yielding the best results. This stability may be attributed to the robust optimization algorithms employed within the Scikit-Learn framework. A similar pattern can be observed for the custom model.

For the test data, both models did well, with a maximum accuracy of 0.991 for the scikit-learn method and 0.982 for the custom model. A similar pattern can be seen for the test data with lower λs and higher learning rates showing the best results. Both models seem stable

for multiple hyperparameters. The stability of both models across multiple hyperparameter combinations reinforces their reliability in practical applications.

5.6 Evaluation of the Various Algorithms

In this section, the various algorithms employed in our analysis, including Ordinary Least Squares (OLS), logistic regression, and neural networks are evaluated and compared. The evaluations are based on training and test accuracy, mean squared error (MSE), and R^2 values.

5.6.1 Ordinary Least Squares (OLS)

OLS demonstrated good predictive power, with a training MSE of 0.0103 and an R^2 of 0.9975, indicating that the model captures a substantial portion of the variance in the training data. Test data results were similarly robust, with an MSE of 0.0130 and an R^2 of 0.9968. This high performance suggests that OLS is well-suited for the dataset, likely benefiting from its simplicity.

5.6.2 Neural Networks

The neural network models, including a simpler configuration with one hidden layer and a more complex architecture with five hidden layers, exhibited varied performance. The simpler network achieved a maximum training accuracy of 1.0 and a test accuracy of 0.991, suggesting strong performance but also raising concerns about potential overfitting given the high training accuracy.

Conversely, the more complex model faced challenges with significant drops in accuracy at higher regularization values; the custom implementation fell to 0.371 under the highest λ, while the scikit-learn maintained relatively high accuracies across the hyperparameters. This contrast highlights the challenges of training deeper networks, where overfitting can become a critical issue, requiring careful hyperparameter tuning. In this context, the Adam optimizer may be particularly useful, as its ability to combine gradient and momentum information helps navigate the complex hyperparameter space. Despite these challenges, the custom model maintained a test accuracy between 0.956 and 0.982 across most configurations.

5.6.3 Logistic Regression

Logistic regression performed reliably in the classification task, with the custom implementation achieving a training accuracy of 0.991, compared to 0.987 for the Scikit-Learn version. Both models showed stable performance across various hyperparameters, favoring lower regularization parameters and higher learning rates. For test accuracy, the custom model reached 0.982, slightly below Scikit-Learn's 0.991, but still reflecting strong overall performance. The consistency across both methods indicates that logistic regression is a solid choice for this type of binary classification.

5.6.4 Comparison of Performance

OLS was the most straightforward and effective method for regression tasks, while logistic regression exhibited robust capabilities in classification. The neural networks, particularly the custom implementation, displayed significant potential but required nuanced hyperparameter tuning to mitigate overfitting.

In conclusion, each algorithm presents distinct advantages, with the choice of method depending on the specific requirements of the task. OLS remains a strong candidate for simple regression problems, logistic regression proves reliable for classification, and neural networks offer flexibility for more complex relationships, albeit with a need for careful design and validation to ensure optimal performance.

6 Conclusion

This project demonstrated the effectiveness of gradient descent techniques and neural networks in addressing regression and classification problems. For linear regression, the plain gradient descent (GD) and stochastic gradient descent (SGD) methods were implemented and tuned to minimize the mean squared error (MSE), with SGD proving particularly efficient. Additionally, adaptive algorithms like Adam and RMSprop offered faster convergence, making them valuable for complex optimization challenges. The neural network regression with SGD achieved robust performance, especially with ReLU and Leaky ReLU activations, balancing accuracy and stability. In the classification problem on the Wisconsin Breast Cancer dataset, neural networks and logistic regression models were explored, each showing high accuracy. Logistic regression provided reliable results with simpler implementation, while neural networks demonstrated adaptability to various architectures and regularization. Overall, this study highlighted

the strengths and limitations of each method, emphasizing the importance of hyperparameter tuning and algorithm choice based on the problem's complexity. The findings emphasize that while traditional methods like OLS and logistic regression remain effective, neural networks offer flexibility and depth, particularly for nonlinear and large-scale datasets. Future work could explore larger-scale datasets and experiment with resampling techniques to further understand model performance and improve hyperparameter tuning.

Bibliography

1. Goodfellow I, Bengio Y and Courville A. Deep Learning. p. 80-82, 170-171, 290-296. MIT Press, 2016
2. Géron A. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. p. 118-121. O'Reilly Media, 2019
3. Hjorth-Jensen M. Week 39: Optimization and Gradient Methods. 2024. Available from: https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week39.ipynb
4. Hjorth-Jensen M. Week 42 Constructing a Neural Network code with examples. 2024. Available from: https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week42.ipynb
5. Hjorth-Jensen M. Week 35: From Ordinary Linear Regression to Ridge and Lasso Regression. 2024. Available from: https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week35.ipynb
6. Hjorth-Jensen M. Resampling Methods. 2024. Available from: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter3.html#more-on-rescaling-data
7. Datta L. A Survey on Activation Functions and their relation with Xavier and He Normal Initialization. arXiv.org. 2020. Available from: https://arxiv.org/abs/2004.06632 [Accessed on: 2024 Nov 4]

Appendix

I Appendix A

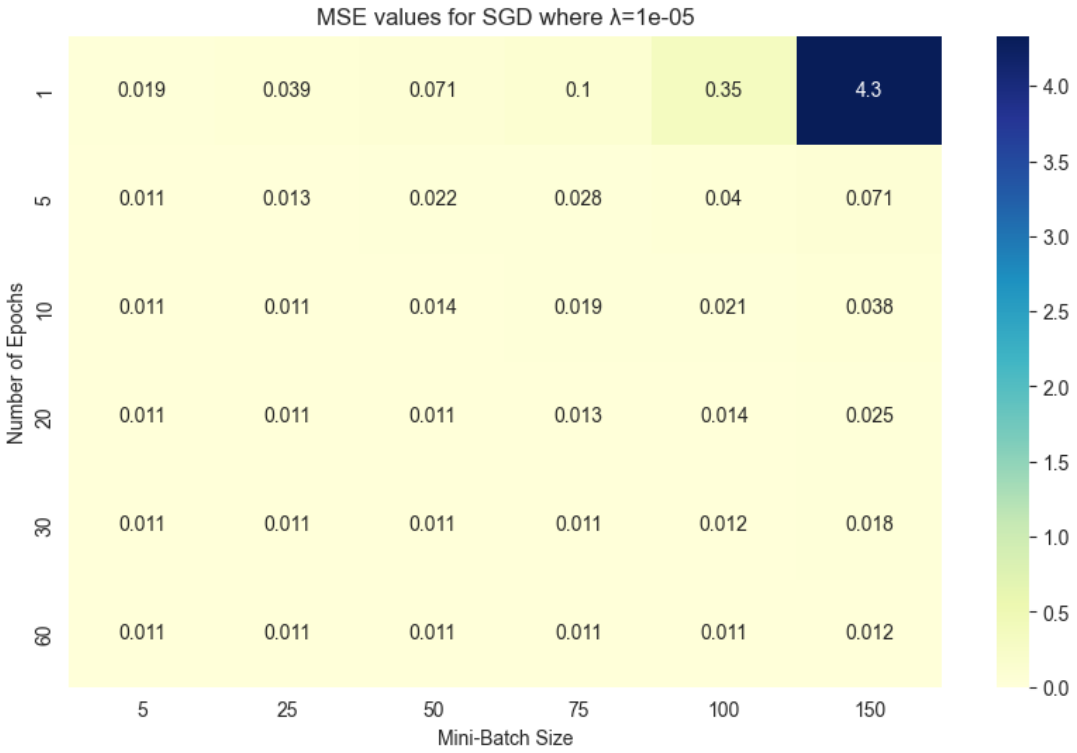


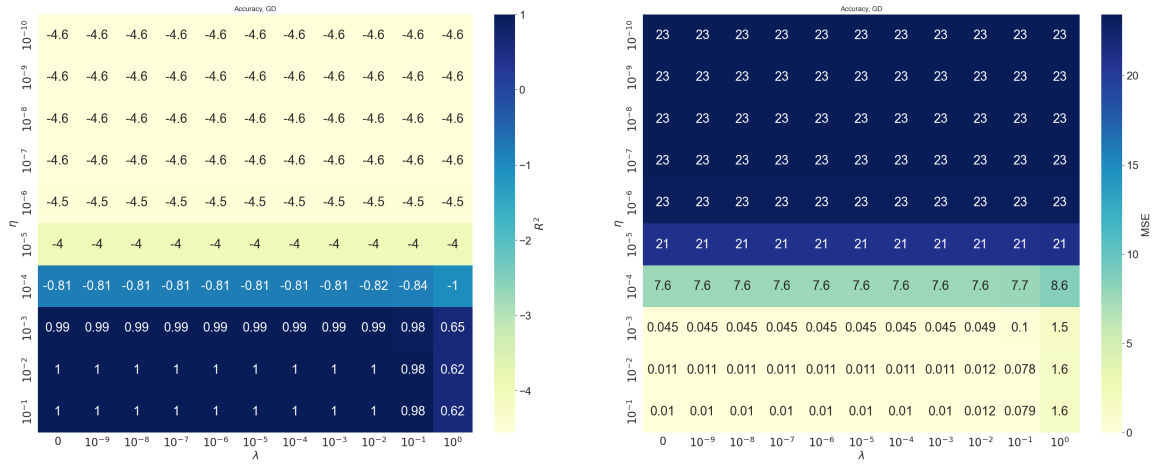
Figure I.1: OLS

Figure I.2: OLS2

Figure I.3: OLS3

II Appendix B - Autograd

.1 Autograd - Gradient Descent



(a) Heatmap for R^2

(b) Heatmap for MSE

Figure II.1: Heatmap for R^2 and MSE using plain gradient descent, with the learning rate (η) as a function of the hyperparameter (λ), for n = 400 and iterations = 2000.

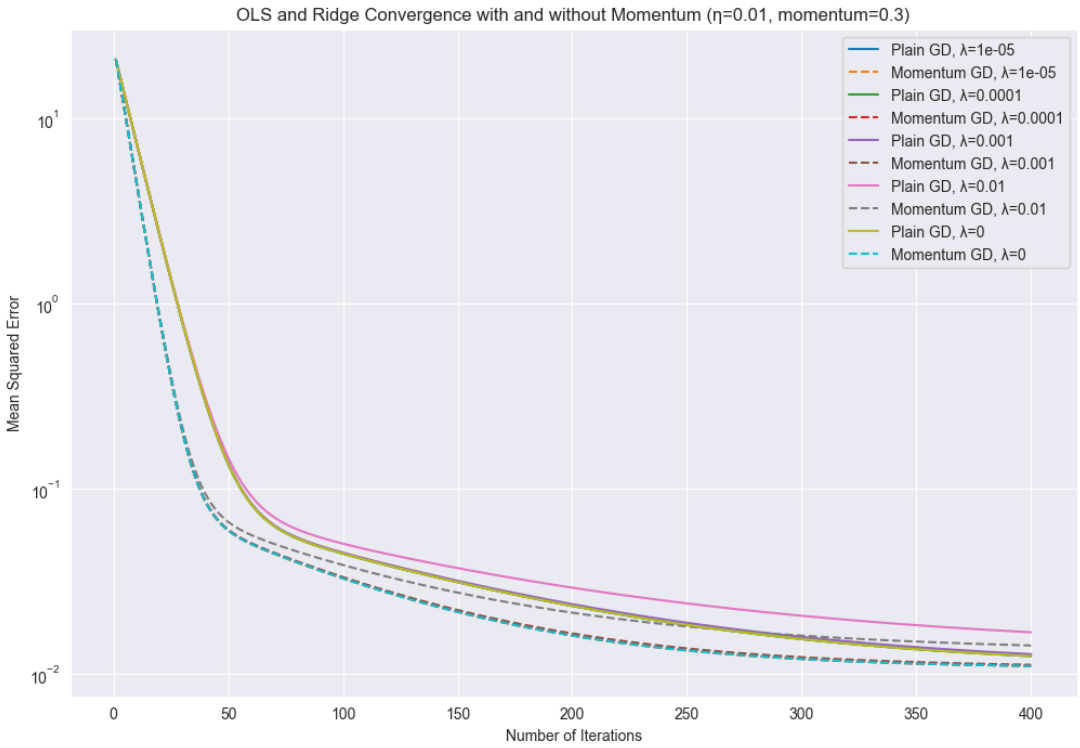


Figure II.2: Comparing the convergence of OLS ($\lambda = 0$) and Ridge regression for various λ values using plain gradient descent (GD) and gradient descent with momentum. The learning rate was set to 0.01, with momentum = 0.3 and initial change set to 0.0.

.2 Autograd - Stochastic Gradient Descent

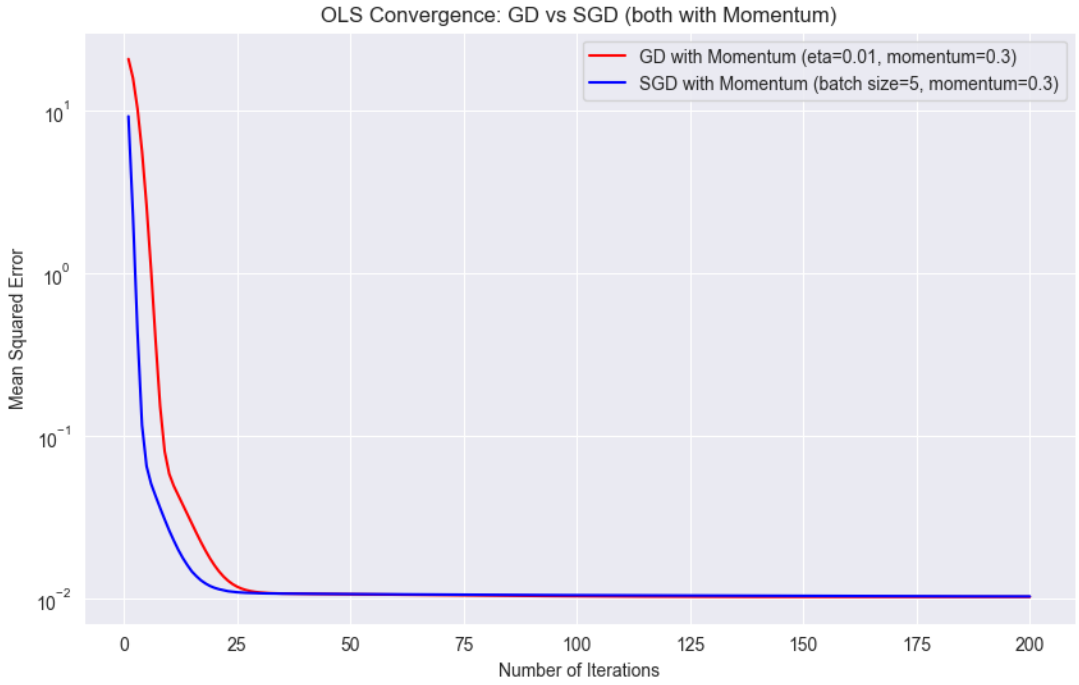
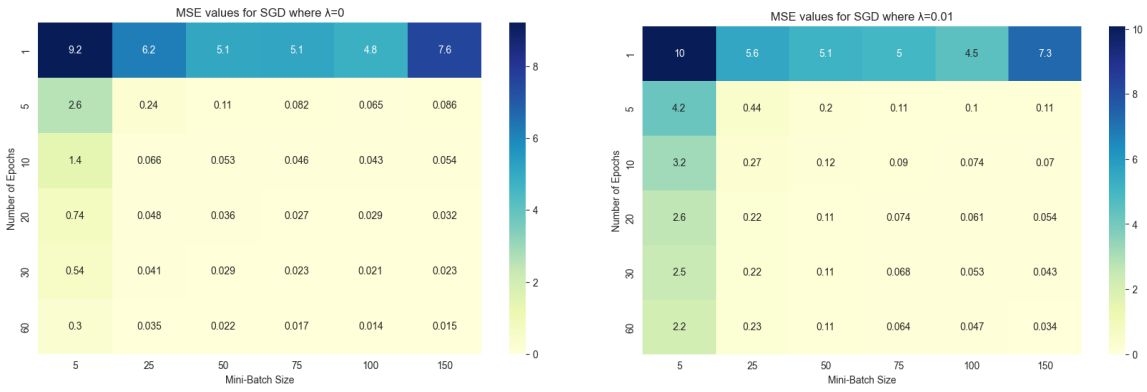


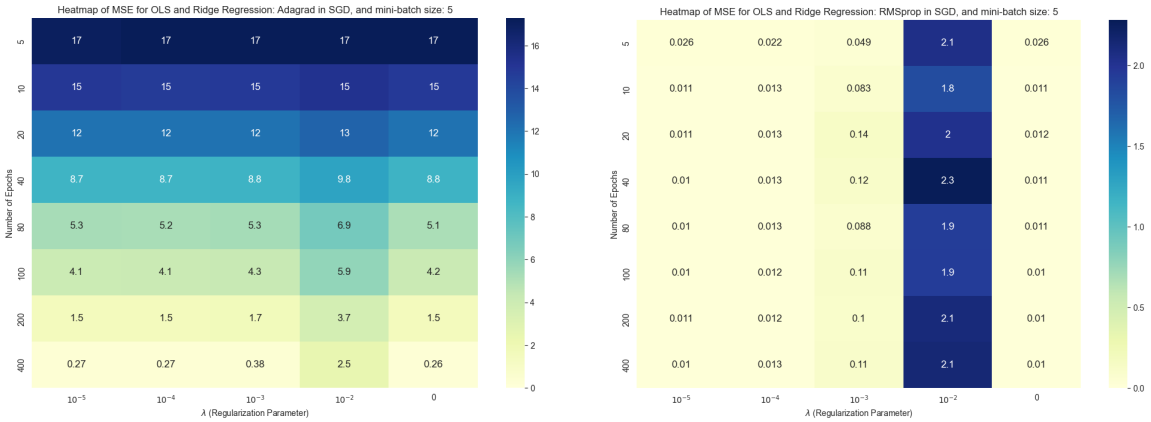
Figure II.3: Comparing the convergence of OLS for GD and SGD, both with momentum set to 0.3 and initial change = 0. For SGD, the mini-batch size is set to 5, and for GD the learning rate to 0.01



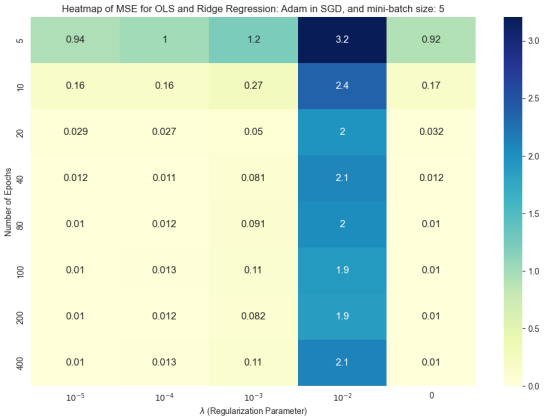
(a) OLS **(b)** Ridge regression where lambda = 0.01

Figure II.4: Two heatmaps to see the effect of the regularization term λ, mini-batch sizes and number of epochs on both OLS and ridge regression. Here the $\lambda = 0$ represents OLS and $\lambda = 0.01$ ridge regression

.3 Adaptive learning rate algorithms: Adagrad, RMSprop and Adam



(a) Heatmap for Adagrad **(b)** Heatmap for RMSprop



(c) Heatmap for Adam

Figure II.5: Heatmaps showing the MSE for OLS and Ridge Regression using Adagrad, RMSprop and ADAM in SGD, as a function of the regularization parameter λ and number of epochs, with a mini-batch size of 5. When λ is 0, the results represent OLS.

III Appendix C - MLPRegressor Heatmap

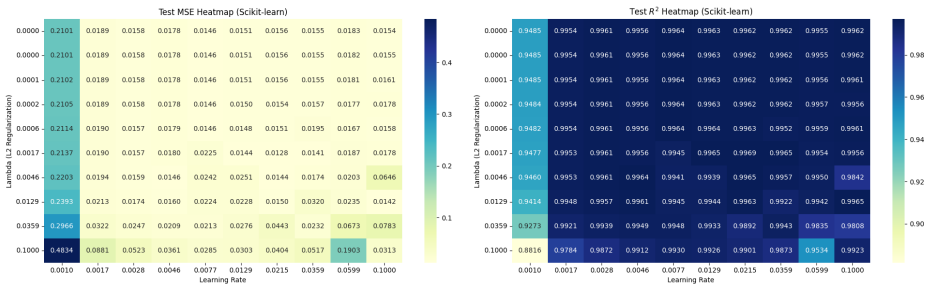


Figure III.1: Scikit-learn Neural Network with one hidden layer (50 nodes) and logistic activation: Heatmap for test MSE and R^2 for different learning rates (η) and L2 regularization (λ).