University of Oslo

FYS-STK3155

Project 2:
Classification and Regression, from linear and logistic regression to neural networks

Laura Ruffoni
Leonardo Proetto

October 2024

Contents

Abstract

In machine learning, neural networks (NN) represent a really flexible tool that can adapt well to handle a wide range of tasks. Inspired by the interconnected structure of biological neurons, they have evolved into some of the most powerful tools in machine learning, enabling advanced applications such as image recognition, natural language processing, and complicated pattern analysis.

This project extends the analysis conducted in Project 1 to explore how neural networks are trained and how they perform in both simple and more complex scenarios, for both regression and classification tasks. One of the main goals of the project is to compare their performance against more basic methods such as Ordinary Least Squares (OLS) and Ridge regression for regression tasks, and Logistic Regression for classification. The goal is to understand when an increased model complexity benefits the performance of the models. Throughout the analysis, particular attention is given to understanding how different neural network architectures, activation functions and hyperparameters influence each model's performance.

First, we evaluated various optimization algorithms such as Gradient Descent, Stochastic Gradient Descent (SGD), Adagrad, RMSprop and Adam. In OLS regression over a relatively simple function, SGD performed the best, with a validation MSE = 1.0338, while for Ridge regression, the mini-batches approach with momentum yielded a validation MSE = 1.0275.

We then implemented our NN code and tested it on a simple function. It became clear that despite the increase in complexity, the gain in performance was minimal, making simpler approaches the preferred choice. On a more complex function, the NN performed well when its architecture was relatively simple. The best results were obtained with one hidden layer, 50 nodes, Adam as an optimization method and ReLU as the activation function. Over the validation set, an $R^2 = 0.9996$ and MSE = 0.0002 were achieved.

The comparison of Logistic Regression and NN on classification tasks highlighted how, in a real-world setting, both methods perform satisfactorily, with accuracy levels above 0.9.

Overall, this work clearly demonstrates that the choice of model should be tailored to the complexity of the problem at hand.

1 Introduction

In data analysis, selecting the appropriate model to effectively handle regression and classification tasks across varying levels of complexity is a central challenge. Equally crucial is the tuning of model parameters, which directly influences the accuracy of the chosen model. This project confronts these challenges directly, comparing traditional regression techniques with neural network models to identify when added model complexity enhances results and when it doesn't. We seek to reveal insights into how model choice, different optimization methods and tuning of hyperparameters impact efficiency in both regression and classification. We examine OLS and Ridge regression models, with a rigorous investigation of different optimization algorithms, alongside a Feed Forward Neural Network, comparing their abilities to fit different continuous functions. We also test the Logistic Regression and Neural Network capabilities in classification tasks.

Our analysis begins with a comprehensive examination of various optimization algorithms, including Gradient Descent, Stochastic Gradient Descent, Adagrad, RMSprop and Adam, focusing on their application to OLS and Ridge regression models. Each method is evaluated in terms of convergence efficiency, computational cost and suitability for different model complexities.

We also cover regularization method on the Neural Network. Further, to assess the regression capabilities of the neural network, we evaluate its performance on both a simple and a more complex function, aiming to identify conditions where neural networks outperform linear models, such as OLS and Ridge regression.

Finally, for classification tasks, we examine the predictive accuracy of the FFNN and compare it to that of Logistic Regression, using a binary dataset.

We extend our analysis by applying regularization techniques to the Neural Network models to mitigate overfitting, examining their impact on generalization for both regression and classification tasks. Throughout this study, we rigorously analyze the impact of key hyperparameters, including learning rates, regularization terms, and network architecture, on model performance. This investigation explores applications of Neural Networks and offers insights into model selection, the effectiveness of different optimization methods and the balance between model complexity and computational efficiency.

2 Methods

2.1 Linear Regression and Iterative Methods

2.1.1 Data Processing and plain Gradient Descent

The goal of most machine learning problems is to accurately estimate a real function by optimizing its parameters with respect to the data. As demonstrated in Project 1, the optimization process for algorithms like Ordinary Least Squares and Ridge regression can sometimes be solved analytically. However, in most cases, an analytical solution is not available. In such cases, iterative methods become the preferred approach, as they do not rely on deriving an exact mathematical formula for the solution [9].

In this context Gradient Descent represents a key algorithm that employs an iterative method to update the model's parameters. We implemented gradient descent to optimize the parameters of the OLS and Ridge regression models explored in Project 1.

The input data consists of 100 random samples uniformly distributed between 0 and 2.

```
# Creation of data
np.random.seed(67)
n = 100
X = 2 * np.random.rand(n, 1)
```

Firstly, the code is tested on a one-degree polynomial function without any noise.

```
y = 2 + 3 * X
```

Then, we generate the dependent variable y from a two-degree polynomial relationship. Gaussian noise is added to introduce variability, making the dataset more representative of real-world scenarios.

```
y = 4 + 3 * X + 2*X**2 + np.random.randn(n, 1)
```

With the help of `train_test_split()`, a Scikit-learn built-in function [2], the data is split into training data (70%), used to train the models, and test data (20%). From the training data, a validation set (20%) is also extracted. In this project, we will generate polynomial features in the training and validation datasets thanks to Scikit-learn built-in function `PolynomialFeatures()` [2]. The values of the parameters are initiated randomly.

The processed data serves as input for all the gradient methods we implemented. In each, we focus on tuning the learning rates. In particular the following learning rates are tested: [0.001, 0.01, 0.1, 1.0/np.max(EigValues)]. The last value is based on the eigenvalues of the Hessian matrix, which indicate the curvature of the loss function. This approach scales the learning rate inversely to the steepness of the loss function. We select the best learning rate based on the validation MSE.

Since we know the true function, we also check how well the optimized parameter approximates the true ones for each method. An important aspect of all gradient descent methods is determining when to stop the process to ideally reach a point close to the minimum of the cost function.

In our analysis the training is set to stop after a predefined number of 1000 epochs is reached or if all the gradient values reach values under 10^{-4}. The performance of the different optimization methods will also be analyzed over 100 epochs to assess the impact that the number of epochs has on the process.
The gradient measures the direction and magnitude of change in the cost function. This allows us to gradually reach the minimum [6]. If the cost function is convex, gradient methods ensure to get to the global minimum. Non-convex functions are very common in machine learning, if the initial guess for the parameters is not good enough the method is likely to converge in a local minimum. This highlights the sensitivity to the chosen initial condition [9]. In each epoch, gradient descent (GD) uses the entire training set to update the parameters, which can become computationally expensive in scenarios involving large datasets [9]. To overcome this issue, stochastic gradient descent and mini-batch gradient descent will be employed. Another limitation of GD is its sensitivity to the choice of learning rate, which will be addressed by exploring adaptive methods and momentum techniques.

Our analysis focuses firstly on plain gradient descent. Our `plain_GD()` function explores how the learning rate influences the optimization process by testing several values. To have a more comprehensive view of the results we plot the train and validation MSE for every learning rate.

2.1.2 Stochastic Gradient Descent

The stochastic gradient descent method is then employed. This technique helps enhance the efficiency of the training process, especially for large datasets. In fact, the parameters of the model are updated for each training sample individually and not over the entire dataset as in plain gradient descent. The stochastic process is determined by the instances chosen randomly at each iteration. The overall computational cost is reduced but at the cost of a reduced precision. In fact, calculating the gradient on one sample at the time introduces more variability in the updates that can cause the optimization path to oscillate, making it less precise [5]. We apply manually SGD in the function `plain_SGD_hand()` and compare the results against `plain_SGD()` function which employs Scikit-learn. However our analysis will focus on mini-batch GD, as it is considered more robust [2].

2.1.3 Mini-Batch Gradient Descent

Even though SGD technically refers to using a single example at a time to evaluate the gradient, the term SGD is used even when referring to Mini-Batch Gradient Descent [7]. This method computes the gradient descent over a small random subset of the training set. This helps reducing the computational cost and speeds up the convergence of the algorithm w.r.t. plain GD while reduces the noisiness typical of SGD [3] [5]. Mini-batch Gradient Descent is sensitive to the learning rate and the choice of mini-batch size. In this work different sizes [16, 32] for the batches have been tested to evaluate the effect on model performance and training efficiency. We implemented the mini-batch GD both manually (`plain_SGD_with_mini_batches()`) and using Scikit-learn (`plain_SGD_with_batches()`) [2]. Both methods, for every epoch, shuffle the training data and extract random mini batches of inputs which are used to train the model. Shuffling the data ensures that each mini-batch is a representative mix of the dataset. The preprocessing of the data and the evaluation of the best model are handled as in the plain GD case. The tuning of the learning rate is implemented in the functions and the best value is evaluated based on the validation MSE.

2.1.4 Momentum

Previously, the size of the step of the descent was simply the norm of the gradient multiplied by the learning rate. The method of momentum is designed to accelerate learning. It introduces a momentum parameter also known as velocity which is the direction and speed at which the parameters move through parameter space [3]. This enhances the optimization process by allowing it to traverse areas with differing gradients more effectively, leading to a faster convergence to the optimal solution. The step size increases when many successive gradients point in the same direction [3] [9]. Instead, if gradients are inconsistent the momentum might help mitigate drastic changes in directions, enhancing the smoothness of the process. In our work the momentum value is set to 0.9 as considered the most effective, based on the results obtained.

```
new_change = adjusted_eta * gradient + momentum * change
beta = beta - new_change
change = new_change
```

To evaluate momentum impact, we introduce it into each gradient method.

2.1.5 Adaptive algorithms

In the training process of a neural network the cost function can be very sensitive to changes in some directions of the parameter space, while being less sensitive in others. Parameters associated with steep gradients may be updated too aggressively, while those associated with gentler gradients may converge too slow [3].

Adaptive learning rate algorithms optimize the gradient descent by automatically adjusting the learning rates during the training process based on the gradients. However, the initial value of the learning rate still influences the training process, so it requires careful tuning. These approaches obtain a faster convergence of the descent, preventing oscillations and overshooting leading to a smoother parameter update [3]. During this project we focused on the analysis of the following adaptive learning rate methods: Adagrad, RMSprop and Adam. All the functions used to implement adaptive methods are modified versions of those used for plain GD and SGD. The only change is in the parameter update step, where different strategies are applied to adjust the learning rates adaptively. Additionally, we compared the performance of these adaptive methods both with and without momentum. As said before, all the approaches have been implemented in OLS and Ridge Regression.

2.1.6 AdaGrad

The AdaGrad algorithm individually adapts the learning rate of all model parameters by decreasing it based on the sum of the squares of past gradients, making the updates smaller for parameters that have received larger updates [1].

```
accumulated_gradients += gradient ** 2
adjusted_eta = eta / (np.sqrt(accumulated_gradients) + 1e-8)
```

This means that when gradients are large, the learning rate for those parameters decreases significantly, whereas parameters with smaller gradients experience only a slight reduction in their learning rate. Consequently, this approach enables more effective updates in the directions where the loss landscape is less steep [3]. Adagrad converges quickly when applied to a convex function. A limitation is its tendency to apply an excessive decrease of the learning rate over time, making Adagrad less suitable in non-convex optimization problems as it would struggle to escape saddle points or local minimum. We applied Adagrad to both plain gradient descent and mini-batch gradient descent, and also provided versions of each with momentum incorporated.

2.1.7 RMSprop

RMSprop adapts Adagrad for non-convex settings by introducing a decay rate parameter that counteracts the rapid reduction of the learning rate. It uses a weighted moving average of squared gradients, which means it gives more importance to recent gradients while gradually forgetting older ones [3].

```
accumulated_gradients =  decay_rate * accumulated_gradients + (1 - decay_rate) *
    (gradient ** 2)
adjusted_eta = eta / (np.sqrt(accumulated_gradients) + 1e-8)
```

The decay rate parameter, set at 0.9, controls how quickly the influence of past gradients diminishes, balancing the focus between recent and older gradients. This adaptability helps maintain a more stable learning rate during training, preventing it from decreasing too quickly and allowing the method to adjust to changes in the loss function shape. As a result, RMSprop often achieves faster convergence compared to Adagrad. RMSprop is applied to both plain gradient descent and mini-batch gradient descent. Versions of each with momentum are incorporated.

2.1.8 Adam

Adam is a combination of the gradient descent with momentum algorithm and RMSprop [9]. It calculates adaptive learning rates for each parameter by tracking two moving averages: the mean (first moment) of the gradients, which represents the momentum, and the variance (second moment) of the gradients. Like RMSprop, Adam employs decay rates that adjust the learning rate for each parameter individually, taking into account both recent gradients and their squared values [3]. This combination of momentum and adaptive learning rates allows Adam to effectively adapt to changes during training, making it efficient and capable of converging quickly, even in noisy environments.

```
# Update biased first moment estimate
 first_moment = decay1 * first_moment + (1 - decay1) * gradient

# Update biased second raw moment estimate
second_moment = decay2 * second_moment + (1 - decay2) * (gradient ** 2)

 # Correct bias in first moment
first_moment_corrected = first_moment / (1 - decay1 ** time_step)
# Correct bias in second moment
second_moment_corrected = second_moment / (1 - decay2 ** time_step)

# Update beta
beta -= eta * first_moment_corrected / (np.sqrt(second_moment_corrected) + epsilon
    )
```

In our project the decay rate are set respectively at 0.9 and 0.999. As, before, Adam is implemented on both plain gradient descent and mini-batch GD.

2.1.9 Ridge Regularization Applied to Linear Regression

In Project 1, we explored Ridge regularization in depth, highlighting its advantages. We repeat the analysis with a range of lambda values [0.0001, 0.001, 0.01, 0.1] to examine its effects on the model's performance. Our functions have been updated to integrating Ridge regularization across various iterative methods. To provide a qualitative understanding, we plot the MSE for each lambda value. Consistent with previous analyses, we compare results across different approaches applied to both GD and SGD, using 1000 epochs for each setting.

2.2 Neural Networks

A neural network is a computational model that attempts to imitate the way a human brain works. It consists of interconnected layers of nodes (neurons) that process and learn from data. The organization and parameters of the connections between neurons determine the output [4]. A network learns by adjusting the weights W and biases b assigned to the connections, based on input data. Neural networks are designed to manage vast amounts of data, making them effective for solving complex problems. They are powerful tools for regression tasks, predicting continuous values by learning complex relationships in data. For classification tasks, they can be adapted by changing the output layer to assign each input to a class. In this project, we designed a neural network that is applied for regression analysis in the first part of the study. Later, this network is adapted for classification tasks, allowing us to analyze its performance across different predictive contexts.

2.2.1 Neural Networks for Regression

The structure of a neural network consists of an input layer, one or more hidden layers, and an output layer. The input layer receives the raw data, which is then passed through the hidden layers. Each hidden layer applies a linear transformation using weights W and biases b, computed as $z = W \cdot X + b$, where X is the input data. The output layer typically consists of a single neuron without any activation function or with a linear activation function $\hat{y} = a \cdot W + b$, allowing the model to predict continuous values.

First, we applied our code to a simple second-degree function, comparing results both with and without added noise.

The input data consists of 100 random samples uniformly distributed between 0 and 2, split into training, validation, and test sets using Scikit-learn's `train_test_split()` function [2]. After comparing results between networks trained on scaled versus unscaled data, we decided not to scale the inputs. In fact, scaling could produce low values that, in our case, distorted the results and led to predictions further from the actual values. Our analysis trains the network using MGD, so all the code is developed to efficiently handle batches of input data in each epoch. The function `create_layers_batch()` is the first employed.

```
def create_layers_batch(network_input_size, layer_output_sizes):
    layers = []
    i_size = network_input_size
    for layer_output_size in layer_output_sizes:
        W = np.random.randn(i_size, layer_output_size) * np.sqrt(2 / (i_size +
    layer_output_size))
        b = np.zeros(layer_output_size)
        layers.append((W, b))
        i_size = layer_output_size
    return layers
```

It takes as arguments the size of all the layers and creates a proper weight and bias matrix for each one. The weights are initialized following the Lecun initialization, which helps maintain stable gradients and improves training efficiency by scaling weights according to the layer's size. Zero values are set for biases.

Then the activation functions need to be specified. The output is sensitive to the choice of the number and size of hidden layers and which activation function do they apply. In this case, we compared the results by employing sigmoid and ReLU activation functions using different combinations of the number of layers and nodes. In the regression tasks the identity function is implemented at the output layer, that is equal to not having any activation at all in the last layer.

Neural networks depend on several hyperparameters. As previously discussed, the learning rate, batch size and epochs influence the training process.

Afterwards, we employ the `train_nn()` function which trains a Feed Forward Neural Network. For each epoch, it shuffles the training data and extracts randomly a batch. One iteration consists of one feed forward step and one backpropagation step [8]. This procedure is carried out by `backpropagation_batch()`. This function initiates a forward pass with `feed_forward_saver_batch()`, which stores intermediary layer inputs, activations, and the network's predictions.

```
def feed_forward_saver_batch(inputs, layers, activation_funcs):
    layer_inputs = []
    zs = []
    a = inputs
    for (W, b), activation_func in zip(layers, activation_funcs):
        layer_inputs.append(a)
        z = a @ W + b
        a = activation_func(z)
        zs.append(z)
    return layer_inputs, zs, a
```

This stored information is then used to calculate gradients during the backward pass. Starting from the output layer and moving backward through the network, the function computes the gradient of the cost function, with respect to the parameters in the neural network. The chain rule is implemented to calculate the gradient.

```
def backpropagation_batch(inputs, layers, activation_funcs, targets,
    activation_ders, cost_der):
    batch_size = inputs.shape[0]
    layer_inputs, zs, predict = feed_forward_saver_batch(inputs, layers,
    activation_funcs)
    layer_grads = [() for _ in layers]

    for i in reversed(range(len(layers))):
        layer_input, z, activation_der = layer_inputs[i], zs[i], activation_ders[i
    ]

        if i == len(layers) - 1:
            dC_da = cost_der(predict, targets)
        else:
            (W, b) = layers[i + 1][:2]
            dC_da = dC_dz @ W.T

        dC_dz = dC_da * activation_der(z)
        dC_dW = dC_dW = layer_input.T @ dC_dz / batch_size
        dC_db = np.mean(dC_dz, axis=0)

        layer_grads[i] = (dC_dW, dC_db)

    return layer_grads
```

In the list `layer_grads` are stored the new gradients for weights and biases that are returned to `train_nn()` and used to update their value.

```
def train_nn(inputs, targets, val_inputs, val_targets, layers, activation_funcs,
    activation_ders, learning_rate, epochs, batch_size):
    num_samples = inputs.shape[0]
    training_metrics = {'mse': [], 'r2': []}
    validation_metrics = {'mse': [], 'r2': []}

    for epoch in range(epochs):
```

```
        # Shuffle the training data
        permutation = np.random.permutation(num_samples)
        inputs_shuffled = inputs[permutation]
        targets_shuffled = targets[permutation]

        # Training over batches
        for i in range(0, num_samples, batch_size):
            input_batch = inputs_shuffled[i:i + batch_size]
            target_batch = targets_shuffled[i:i + batch_size]

            # Backpropagation
            layer_grads = backpropagation_batch(input_batch, layers,
    activation_funcs, target_batch, activation_ders, mse_der)

            # Update weights and biases
            for j, (dC_dW, dC_db) in enumerate(layer_grads):
                W, b = layers[j]
                layers[j] = (W - learning_rate * dC_dW, b - learning_rate * dC_db)


        # Calculate metrics on training set
        predictions_train = feed_forward_batch(inputs, layers, activation_funcs)
        mse_train = mse(predictions_train, targets)
        r2_train = r2_score(predictions_train, targets)
        training_metrics['mse'].append(mse_train)
        training_metrics['r2'].append(r2_train)

        # Calculate metrics on validation set
        predictions_val = feed_forward_batch(val_inputs, layers, activation_funcs)
        mse_val = mse(predictions_val, val_targets)
        r2_val = r2_score(predictions_val, val_targets)
        validation_metrics['mse'].append(mse_val)
        validation_metrics['r2'].append(r2_val)

    return layers, training_metrics, validation_metrics
```

In addition, to quantitatively assess the network's performance we compute the MSE and R^2 on both the training and validation. For this step we employ the `feed_forward_batch()` function. This function implements a forward pass using the final optimized layers and returns the output which corresponds to the predicted values.

```
def feed_forward_batch(inputs, layers, activation_funcs):
    a = inputs
    for (W, b), activation_func in zip(layers, activation_funcs):
        z = (a @ W) + b
        a = activation_func(z)
    return a
```

Lastly, to provide a qualitative view of the network's performance, two plots were created: one showing the training and validation MSE across epochs and another comparing the original data points, the true function and the neural network's predicted curve.

All the procedure is repeated employing Scikit-learn function [2]:

```
mlp = MLPRegressor(
    hidden_layer_sizes=(10,),  # Single hidden layer with 10 neurons (like before)
    activation='logistic',     # 'logistic' corresponds to the sigmoid function
    solver='sgd',              # Stochastic Gradient Descent optimizer
    learning_rate_init=0.01,  # Initial learning rate
    max_iter=10000,              # Number of epochs
```

```
    batch_size=32,              # Minibatch size
    random_state = 32
)
```

As previously explained, the SGD approach with a fixed learning rate is effective but has several limitations, especially in complex functions. To address these limitations, the `train_nn_adam()` function employs the Adam optimizer. Its structure remains the same as `train_nn()` but the parameters are updated taking into account the first and second moment estimates. Our goal is to determine if this adaptive method can provide a beneficial alternative for training neural networks.

Again, we repeated this approach employing Scikit-learn function [2]:

```
mlp = MLPRegressor(
    hidden_layer_sizes=(10,),
    activation='logistic',
    solver='adam',
    learning_rate_init=0.01,
    max_iter=10000,
    batch_size=32,
    random_state=32
)
```

In the second part of our analysis, we used the same procedure, but the neural network was trained to approximate a more complex function: $y = \sin(X^2)\cdot e^{-\frac{X}{10}} + \frac{1}{3}X^3 + \log(1+X^2)\cdot\sin(\log(X^6))$. Different numbers and sizes of hidden layers have been tested and compared, providing both a quantitative and qualitative investigation.

2.2.2 Neural Networks for Classification

Neural networks can also be applied to classification tasks since they are universal and versatile estimators. The training procedure is equivalent to the pipeline used for regression task. Only the activation function for the output layer needs to be changed. If it is a binary classification task, the sigmoid is employed since the threshold sets the boundary between the two classes. When solving a multiclass problem, the softmax activation function is the best choice, but the classes need to be mutually exclusive [8]. The output layer will have a number of nodes equal to the number of classes. Additionally, the loss function also varies. Cross-Entropy Loss is employed and it measures the difference between two probability distributions: the true distribution of the labels and the predicted distribution output by the model.

To quantify the network predictive capability, several metrics are applied. Accuracy quantifies the proportion of correctly classified instances relative to the total number of points in the dataset. It is a straightforward measure but if there are imbalances in the size of the classes it may not fully capture the model's efficiency. Confusion matrices give a wider insight, illustrating the number of true positive, true negative, false positive, and false negative predictions for each class. In our analysis confusion matrices on the validation set are computed before and after training the network by Scikit-Learn's built-in function `confusion_matrix()` [2]. Moreover, we plot accuracy and cross-entropy over the training epochs and illustrated the original data points divided by the model's decision boundary.

To test our code we generate two datasets using the `make_classification()` function from `sklearn.datasets` [2]. In the first dataset, the two classes are perfectly separated, an accuracy of 1 is expected. In contrast, the second dataset contains some overlapping points between the classes. Also, we compared our results to a code implementing Scikit-learn's `MLPClassifier()` [2]. In this example, a single hidden layer with 8 nodes and a sigmoid activation function is specified. Since the classification task is binary, all networks employ the sigmoid function in the output layer. The

training is conducted over 1,000 epochs with a learning rate of 0.01 and 32 as batch size.

The analysis is then extended to a real-world example using the Wisconsin Breast Cancer dataset [2], which contains 569 samples characterized by 30 features. The binary target variable classifies tumors as either malignant or benign. The data is loaded, splitted and scaled. Since the input data has many features, scaling is useful as it ensures that all features contribute equally to the model's learning process. As before, the number of nodes, layers, and the activation functions are specified. The parameters are initialized according to the number of nodes for each layer using `create_layers_batch()`. We compute the initial accuracy and confusion matrix based on the validation set. Training is conducted through the `train_nn_classification()` function, where each iteration involves a forward and backward pass respectively via `feed_forward_saver_batch()` and `backpropagation_batch()`. These functions are adapted from those used in regression but modified for classification, as previously mentioned. After optimizing the parameters, the validation accuracy and confusion matrix are recalculated. Predictions are generated with the `feed_forward_batch()` function. Given the high dimensionality of the input features, it is not possible to plot the original data point divided by the optimized network's boundary. Results where the learning rate is set at 0.01, epochs at 100 and 16 as batch size are analyzed and compared to those obtained using Scikit-Learn's function `MLPClassifier()` [2].

2.2.3 L2 Regularization Applied to Neural Networks

In this project, L2 regularization is incorporated into neural network both for regression and classification tasks. Regularization is applied during backpropagation in the `backpropagation_batch()` function, where a term proportional to W is added to the weight gradients, with λ controlling the regularization strength. A higher λ applies a stronger penalty on large weights, promoting a balance between fitting the training data and maintaining smaller weights which helps it generalize better to new data. Additionally, given the need to tune numerous hyperparameters, we conduct a quantitative and qualitative analysis of the impact of various λ values and learning rates on model performance. We test values of $\lambda = [0.0, 0.001, 0.01, 0.1, 1, 10]$ and learning rates $[0.0001, 0.001, 0.01, 0.1]$, computing performance metrics across these combinations. The results are visualized through heatmaps.

2.3 Logistic Regression

Logistic Regression is another powerful tool for classification problems. Logistic regression models the relationship between one or more independent variables and a binary dependent variable by estimating probabilities using the sigmoid function, represented by:

$$P(Y = 1 \mid X) = \frac{1}{1 + e^{-\beta_0 - \beta_1 X}}$$

It is mostly used for binary classification but it can be extended to multi-class tasks by a one-vs-all approach.

To optimize the model's weights, we implemente gradient descent, stochastic gradient descent, and mini-batch gradient descent, respectively using the functions `logistic_regression_gd()` , `logistic_regression_sgd()` and `logistic_regression_mgd()`. Furthermore, we compare the results against Scikit's learn `LogisticRegression()` function [2]. The cost function for logistic regression is binary cross entropy which penalizes predictions that diverge from the true class probabilities. As in our neural network classification tests, we evaluate the logistic regression models on two datasets: one with linearly separable classes and one with overlapping classes. The data is split

into training, validation and test sets. We train the models using a learning rate of 0.01 and 1000 epochs. Validation accuracy and confusion matrices are computed before and after training, with cost history tracked across epochs. In addition a plot showing the original points and the model's boundary is displayed.

Moreover, we proceed to extend our analysis on real cancer data. The Wisconsin Breast Cancer was loaded from `sklearn.datasets` [2]. Following the approach used in our neural network study, the data was split and scaled appropriately. We then applied our GD, SGD, and MGD functions to this dataset, setting the number of epochs to 1000 and the learning rate to 0.01, matching the parameters used in the neural network analysis to enable a direct comparison of the results.

2.3.1 Ridge Regularization Applied to Logistic Regression

In logistic regression, the Ridge penalty modifies the model's loss function by adding a regularization term. This penalty, added to the gradient during training, helps control overfitting. Our GD, SGD and MGD functions are updated to implement Ridge regularization, and the Scikit-learn function also applies this penalty term. We test, on 1000 epochs, values of $\lambda = [0.0, 0.001, 0.01, 0.1, 1, 10]$ and learning rates $[0.0001, 0.001, 0.01, 0.1]$, computing the loss metric across these combinations. The results are visualized through a heatmap.

3 Results

3.1 Regression

3.1.1 Iterative Methods

Here below are presented the results of the analysis regarding the different Gradient Descent methods. In tables 1 and 2 are discussed the outcomes concerning various plain Gradient Descent techniques, including momentum, RMSprop, Adagrad and Adam. The results relative to the Stochastic Gradient Descent and all alternative methods are instead listed in charts 3 and 4. These results regard the analysis of the gradient descent methods in the Ordinary Least Square regression.

In table 1, considering the overall performance based on the MSE, on a number of epochs = 1000, the plain Gradient Descent has an overall good performance (MSE = 1.0462) that does not significantly differ from all other methods, even from the adaptive methods. Adding the momentum to the process raises the MSE to 1.0531, suggesting that momentum here cannot effectively help in improving the training process. In both algorithms the learning rate is picked at 0.001, this can be justified by the high number of epochs that allow the process to get to the minimum with smaller steps and be more precise. Taking into account the adaptive methods, the overall performance with respect to the GD with or without momentum is not significantly enhanced. Adagrad with momentum is the best performing algorithm with a MSE of 1.0338. When Adagrad is employed, momentum seems to work as expected. The same does not occur for RMS prop that performs better without momentum. This analysis demonstrates how adaptive methods generally begin with higher learning rates, to dynamically adjust them throughout the training process. The fact that these methods do not significantly enhance the performance might be due to the relatively simple task, as they well adapt to large-scale problems. It is interesting to also consider the results obtained with a number of epochs = 100, showed in Table 2. Considering the plain Gradient descent, the MSE is slightly higher and the learning rate is sensibly higher. This is expected as keeping a lower number of epochs will favor a faster but less precise descent to the minimum. The striking change observable with respect to the previous results concerns Adagrad. It is observable how the performance gets significantly worse, this can be explained by the fact that the algorithm does not fully adapt and it

is not able to converge towards the minimum. Which is what happens when 1000 epochs are used. The quality of the gradient descent method is assessed also focusing on the values of the optimized parameters. Generally all methods obtain values close to the parameters of the real functions. Gradient Descent and Adam perform the best. In this case, the analysis conducted with 100 epochs yields poorer results, as the values of the optimized parameters deviate more compared to those obtained with 1000 epochs. This demonstrates that in iterative methods, increasing the number of iterations, within computational limits, allows for a slower yet more accurate optimization process. If the focus of the analysis is obtaining more accurate results, increasing the number of epochs seems appropriate.

	Mean Squared Error (MSE)	Learning rate	Beta0	Beta1	Beta2
GD	1.0462	0.001	3.989	2.997	1.943
GD with momentum	1.0531	0.001	3.914	3.195	1.852
GD Adagrad	1.0682	0.1	3.64	3.662	1.688
GD Adagrad with momentum	**1.0338**	**0.108**	4.207	2.413	2.213
GD RMSprop	1.0503	0.01	3.905	3.238	1.84
GD RMSprop with momentum	1.0543	0.01	3.903	3.225	1.839
GD Adam	1.0438	0.108	4.018	2.92	1.979

Table 1: Summary of OLS regression with MSE, Learning rate and optimized parameters for various Gradient Descent Optimization Methods using 1000 epochs. The best performing method is highlighted in blue.

	Mean Squared Error (MSE)	Learning rate	Beta0	Beta1	Beta2
GD	1.0334	0.108	4.347	2.049	2.378
GD with momentum	1.0449	0.1	3.906	3.235	1.854
GD Adagrad	6.9358	0.1	2.934	3.115	1.187
GD Adagrad with momentum	1.0497	0.108	4.3	1.623	2.67
GD RMSprop	**1.0314**	**0.108**	4.155	2.69	2.176
GD RMSprop with momentum	1.047	0.01	3.963	3.046	1.925
GD Adam	1.064	0.1	3.703	3.663	1.675

Table 2: Summary of OLS regression with MSE, Learning rate and optimized parameters for various Gradient Descent Optimization Methods using 100 epochs. The best performing method is highlighted in blue.

In table 3, the results based on the MSE indicate as best performing the SGD algorithm implemented by Scikit learn with an MSE = 1.0302. The manually implemented SGD performance is worse, and it is enhanced when mini-batches are employed. Furthermore adding the momentum seems to increase the performance even if slightly. Also in this setting, adaptive methods generally perform worse for the same reason presented before. Among them, Adam seems to be the most effective algorithm, which led to an MSE value of 1.0335. Evaluating the different algorithms based on the values of the parameters does not offer any particular insight. The values resemble the original parameters but no one of the methods particularly stands out.

	Mean Squared Error (MSE)	Learning rate	Beta0	Beta1	Beta2
SGD	1.0529	0.001	3.906	3.217	1.844
SGD Scikit	1.0302	0.01	4.106	2.703	2.094
MGD	1.0396	0.108	3.926	3.242	1.881
MGD with momentum	1.0385	0.1	3.908	3.241	1.893
MGD Scikit	1.0419	0.001	3.999	2.98	1.597
MGD Adagrad	1.0432	0.1	3.998	2.936	2.978
MGD Adagrad with momentum	1.0532	0.1	3.917	3.186	1.856
MGD RMSprop	1.0531	0.01	3.934	3.141	1.874
MGD RMSprop with momentum	1.0476	0.01	3.919	3.22	1.851
MGD Adam	1.0335	0.01	4.047	2.654	2.137

Table 3: Summary of OLS regression with MSE, Learning rate and optimized parameters for various Stochastic Gradient Descent Optimization Methods using 1000 epochs and batch size = 32. The best performing method is highlighted in blue.

We present now the results of an analysis whose purpose was understanding how different batch sizes could affect the efficiency of the different optimization methods. In table 4 are presented the results relative to the OLS analysis with MGD and a batch size of 16. The performance of the different methods does not change drastically even if it is clear that for most of the methods having a larger batch size helps the analysis. This might be due to the fact that larger batch sizes provide a more representative sample of the dataset. The only method that improves its performance is MGD with Adagrad and momentum, with an MSE = 1.0377.

	Mean Squared Error (MSE)	Learning rate	Beta0	Beta1	Beta2
MGD	1.0457	0.01	3.996	2.976	1.953
MGD with momentum	1.0431	0.01	3.909	3.131	1.861
MGD Scikit	1.0465	0.001	3.996	2.976	1.951
MGD Adagrad	1.0451	0.1	3.4	2.967	1.958
MGD Adagrad with momentum	1.0377	0.001	3.914	2.648	2.211
MGD RMSprop	1.0532	0.01	3.929	3.162	1.864
MGD RMSprop with momentum	1.0521	0.01	3.906	3.241	1.834
MGD Adam	1.0397	0.1	3.916	3.254	1.881

Table 4: Summary of OLS regression with MSE, Learning rate and optimized parameters for various Stochastic Gradient Descent Optimization Methods using 1000 epochs and batch size = 16. The best performing method is highlighted in blue.

The following results regard the analysis of the gradient descent methods in the Ridge regression. In this Analysis we show the overall accuracy of each model tested on various learning rates and different lambda values for the regularization of Ridge. Table 5 indicates GD with RMSprop as the best performing method with an MSE = 1.0327, a lambda of 0.01 and a learning rate of 0.1. Overall it is clear how adding the momentum does not favor particularly more precise results, except for Adagrad. It is interesting to see how the analysis that included Adagrad and Adam as gradient descent algorithms required a low regularization. This could be reasonable, as Adagrad and Adam frequently adjust parameter values to avoid overshooting, which reduces the need for strong regularization. Plain gradient descent performs overall well and as expected tends to generally pick lower values of learning rate. For this algorithm a relatively higher regularization is preferred, in continuity with what stated before. Overall, Table 5 suggests that the choice of gradient descent variant, adaptive learning rates, and regularization strength should be carefully balanced based on the complexity of the data and model. For simpler tasks like this, adaptive methods do not have a significant advantage, while standard GD methods benefit minimally from added features like momentum or high regularization. The similarity between the optimized parameters from the

Ridge regression and those of the actual function [4,3,2] suggests that all the algorithms allowed an accurate estimation of the underlying relationships. In Ridge this similarity towards the real parameters is reduced compared to the OLS analysis. This result is expected as Ridge applies a regularization as discussed in Project 1.

	Mean Squared Error (MSE)	Lambda	Learning rate	Beta0	Beta1	Beta2
GD	1.0461	0.001	0.01	3.985	2.989	1.949
GD with momentum	1.0482	0.01	0.01	3.946	2.95	1.982
GD Adagrad	1.0683	0.0001	0.1	3.662	1.689	1.068
GD Adagrad with momentum	1.0339	0.0001	0.108	4.206	2.414	2.213
GD RMSprop	**1.0327**	**0.01**	0.1	3.995	3.002	2.031
GD RMSprop with momentum	1.0482	0.01	0.001	3.945	2.95	1.981
GD Adam	1.0438	0.0001	0.108	4.017	2.919	1.979

Table 5: Summary of Ridge regression with MSE, Learning rate and optimized parameters for various Gradient Descent Optimization Methods using 1000 epochs. The best performing method is highlighted in blue.

In table 6 are listed the results of the Ridge regression analysis implemented with Stochastic Gradient Descent and alternative methods. Also here plain Stochastic gradient descent performs well, better than adaptive gradient descent methods. It has a MSE = 1.0474. Implementing the mini-batches approach and momentum results in an improvement in the performance of the model. No tendency is observable in the choice of lambda values, generally weak regularizations are preferred, demonstrating how this gradient descent methods effectively allow a fine-tuning of the parameters. The Scikit-implemented methods perform worse in the SGD algorithm, with respect to manually-implemented methods, but better when the mini-batches approach is included. Given the simplicity of the task, it is reasonable to conclude that adding momentum or employing mini-batch methods do enhance the performance of the mode only slightly. The adaptive gradient descent methods fail to improve the results. In the optimization of the parameters all methods perform similarly.

	Mean Squared Error (MSE)	Lambda	Learning rate	Beta0	Beta1	Beta2
SGD	1.0474	0.01	0.001	3.641	2.545	2.281
SGD Scikit	1.0303	0.0001	0.01	4.043	2.653	2.139
MGD	1.0284	0.01	0.108	3.66	2.563	2.308
MGD with momentum	**1.0275**	**0.01**	0.1	3.718	2.628	2.384
MGD Scikit	1.0323	0.01	0.1	4.043	2.653	2.14
MGD Adagrad	1.0432	0.0001	0.1	3.62	2.595	2.258
MGD Adagrad with momentum	1.0487	0.01	0.1	3.641	2.546	2.281
MGD RMSprop	1.0499	0.01	0.01	3.713	2.617	2.312
MGD RMSprop with momentum	1.0326	0.01	0.1	3.793	2.642	2.382
MGD Adam	1.0336	0.0001	0.01	3.642	2.546	2.283

Table 6: Summary of Ridge regression with MSE, Learning rate and optimized parameters for various Stochastic Gradient Descent Optimization Methods using 1000 epochs and. The best performing method is highlighted in blue.

3.1.2 Neural Networks

We present the results from our trained Neural Network, focusing on its performance across various scenarios. Specifically, we evaluate the network on a simple function, both with and without noise, as well as on a more complex function. The following four cases are examined:

1. A Neural Network optimized using MGD.
2. A Neural Network optimized using MGD, implemented with Scikit-learn.
3. A Neural Network optimized using the Adam algorithm.
4. A Neural Network optimized using Adam, implemented with Scikit-learn.

For each case analyzed, we conducted multiple tests to adjust the Neural Network's parameters, exploring various configurations. Specifically, we tested different numbers of hidden layers, nodes per layer, and activation functions to assess their impact on performance. Table A presents the various tests performed for each case. We began by analyzing the network with 1 hidden layer, progressively increasing up to 4 layers. For each configuration, we tested performance using different combinations of activation functions: only the Sigmoid, only the ReLu or a combination of both. We will focus on the MSE values obtained on the validation set. When Scikit-learn is employed to implement the neural network, only one single activation function at the time is tested. For clarity and brevity, only part of the total results are presented here. However, charts displaying the complete set of outcomes can be found in the supplementary material accompanying this project. These charts correspond to the various analyses, labeled A to H, as detailed in Table 7.

	N° Layer	N° Nodes	Activation Function
A	1	50	Sigmoid
B	1	50	ReLu
C	2	50,30	Sigmoid
D	2	50,30	Relu + Sig.
E	3	50,30,30	Sigmoid
F	3	50,30,30	ReLu+Sig
G	4	50,30,30,40	Sigmoid
H	4	50,30,30,40	ReLu+Sig

Table 7: Summary of all the different neural network's architectures tested.

3.1.3 Neural Networks Applied on Simple Function

In evaluating the Neural Network on the simple function (without noise) as discussed in the methods section, we observe notable performance differences with different configurations. Evaluating the model on a noise-free function provides a baseline for its performance, that will help us understand the Neural Network behavior in an ideal scenario. From the results, using MGD as an optimization method, it is evident that even a simple Neural Network with just one hidden layer achieves good levels of accuracy, producing an MSE of 0.0972 and R^2 of 0.9943 with the Sigmoid activation function, and an improved MSE of 0.0108 and R^2 of 0.9994 with ReLU. Increasing the number of hidden layers enhances the network's efficiency, with the best performance achieved using 4 layers and a combination of ReLU and Sigmoid activations, yielding an MSE of 0.0006 and R^2 of 1.0. A similar performance was obtained with a 2-layer network, also using a mixed activation approach, with an MSE of 0.0007 and R^2 of 1.0. Figure 1 shows how the predicted values differ from the original data points, with this latter configuration for the Neural Network. Given the simplicity of the function and the absence of noise, it is expected that even a basic Neural Network architecture is sufficient to capture the underlying patterns. In these scenarios, it may be advantageous to use a simpler network configuration to reduce computational cost and processing time. Additionally, the

results clearly demonstrate that combining activation functions generally improves performance even if slightly., highlighting the potential benefits of using mixed activations for certain tasks.

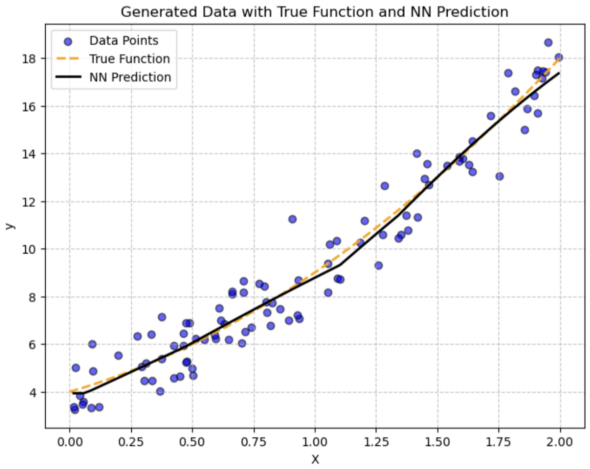

Figure 1: Scatterplot showing original data points in blue. Dashed orange line rapresents the true function. In black the NN predictions.

The Neural Network with MGD implemented with Scikit-learn yielded similar results. The overall performance is good, reaching its best when 1 single hidden layer is employed, with an MSE = 0.0040 and $R^2 = 0.9998$. In this case, it is evident that simpler networks are more effective, as performance deteriorates with an increasing number of hidden layers. Specifically, the network reaches an MSE of 0.0328 with three hidden layers, and with four layers, performance significantly declines, resulting in an MSE of 20.1846 and a negative R^2 value. This indicates that the additional complexity introduced by more layers may lead to overfitting or difficulties in training, reinforcing the idea that simpler architectures can be more suitable for this task.

By substituting the Adam algorithm for MGD, several clear trends emerge. Firstly, there is a noticeable improvement in performance. In this noise-free scenario, the manually implemented Adam method achieves an MSE of 0 and an R^2 of 1 when the Neural Network consists of a single hidden layer with the sigmoid activation function. Figure 2 shows how well the Neural Network can effectively approximate the true function. In contrast, when using Scikit-learn, the performance of the one-layer network results in an MSE of 0.0037 and an R^2 of 0.9998. With the Adam optimizer, it becomes even more apparent that simpler networks outperform more complex architectures, as performance declines rapidly with the addition of layers. Additionally, combining activation functions slightly enhances the results as previously seen.

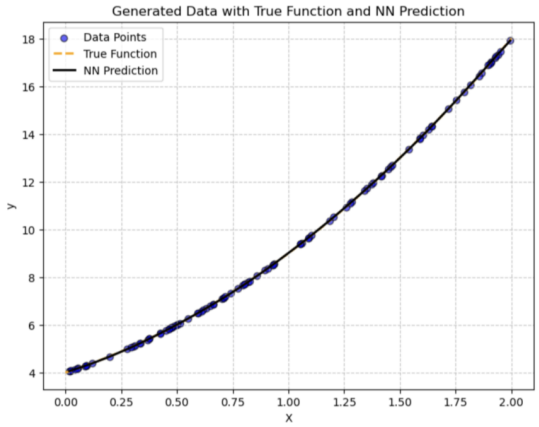

Figure 2: Scatterplot showing original data points in blue. Dashed orange line represents the true function. In black the NN predictions.

We now present the results achieved after introducing noise into the data, allowing us to assess how well the model generalizes in noisy scenarios. As expected, the Neural Network's performance declined in the presence of noise. First, we consider the case where MGD is used as the optimization technique. When manually implemented with a simple configuration featuring only one hidden layer, the algorithm performed well, achieving an MSE of 0.9977 and an R^2 of 0.9401 when employing the ReLu activation function. Generally, it appears that ReLu outperforms Sigmoid in these regression tasks, while using a combination of activation functions tends to yield better results. As the number of hidden layers increases, the performance does not show a clear trend. For networks with three or four layers and a combination of activation functions, the MSE values are 1.1154 and 0.9958, respectively, with corresponding R^2 values of 0.9312 and 0.9418. Given that the increase in performance may be limited or even absent in these cases, it remains evident that a simpler network configuration is preferable. Figure 3 shows how the network approximate the function.

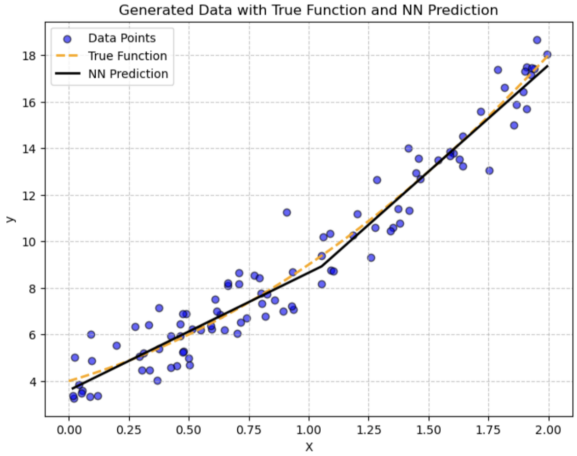

Figure 3: Scatterplot showing original data points in blue. Dashed orange line represents the true function. In black the NN predictions.

When Adam is used as the gradient descent method, the results decline in overall performance.

This deterioration may be attributed to the limited number of data points available to the model and the presence of noise, which could lead to some degree of overfitting. Figure 4 illustrates how the Neural Network closely follows the data points but struggles to replicate the true function effectively. This specific graph was generated using a network configuration with one hidden layer and the sigmoid activation function, resulting in an MSE of 2.0633 and an R^2 of 0.8851. In contrast, when the sigmoid function is replaced with ReLu in this configuration, the model demonstrates improved performance, achieving an MSE of 0.9966 and an R^2 of 0.9420. However, for more complex network architectures, the performance deteriorates, with MSE values ranging between 2 and 3. Specifically, when employing four hidden layers and a mix of activation functions, the MSE escalates to 22.0235, accompanied by significantly low negative R^2 values. As observed previously, when Scikit-learn is used to implement the Neural Network with Adam, we see similar performance in simple models, but a more pronounced trend of deteriorating performance as additional hidden layers are introduced.

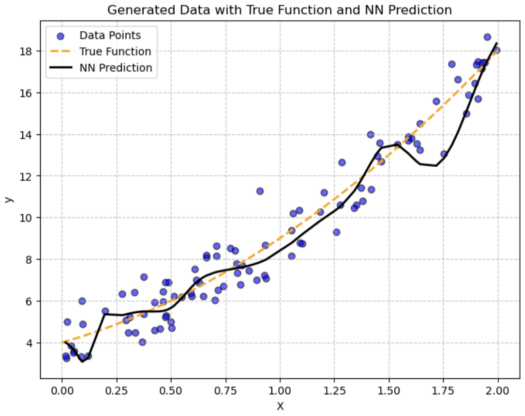

Figure 4: Scatterplot showing original data points in blue. Dashed orange line represents the true function. In black the NN predictions.

We now proceed to compare the performance of the Neural Network with that of OLS regression methods applied to the simple function. In particular, we will consider OLS regression utilizing MGD alongside various adaptive optimization methods.

As already discussed, in general, OLS achieves a MSE of approximately 1, which is consistent with the performance of the Neural Network when MGD is employed. However, the Neural Network exhibits a tendency to oscillate more in its results compared to OLS. Notably, when the Adam optimizer is utilized, the performance of the Neural Network deteriorates significantly, leading to a higher MSE and allowing OLS to outperform it. The stability of OLS under these conditions highlights its robustness for linear relationships. Neural networks have proved to be able to yield satisfactory results even for simple scenarios, however their complexity would probably be unnecessary. Indeed, OLS provides a clear interpretation of model coefficients, therefore when faced with simpler tasks the advantages of OLS make it a more efficient modeling approach.

3.1.4 Neural Networks Applied on Complex Function

As emerged from the previous results a Neural Network might not be the preferable solution for simpler problems as models such as linear regression might provide a better representation of the underlying relationships. It is interesting to understand whether the network can adapt effectively

to increased complexity, potentially revealing strength in scenarios where linear models struggle to replicate the real function. The following results will explore the network's performance on the complex function described in the method's section.

The first analysis conducted focuses on the manually implemented Neural Network with MGD optimization. When using a single hidden layer, the network performs well with the ReLu activation function, achieving an MSE on the validation set of 0.0244 and an R^2 of 0.9563. However, when the sigmoid activation function is applied, the performance declines, with an MSE of 0.2640 and an R^2 of 0.5264, indicating the limitations of the sigmoid in this case. This may be due to the compressive nature of the sigmoid, which maps all input values to a range between 0 and 1.

Figure 5 illustrates how the single-layer Neural Network approximates the true function. While the network replicates the function with reasonable accuracy, a more complex network might enhance the performance. Indeed, networks with two hidden layers show limited improvement. However, when the number of hidden layers is increased to three or four, the model attains a notable increase in accuracy. The most effective configuration is the Neural Network with 4 hidden layers and the ReLu as only activation function, which has an MSE = 0.0048 and R^2 = 0.9914.The sigmoid performs worse also in this case and using a combination of the two functions does not improve the results. Figure 6 illustrates how well this complex network approximates the true function, supporting the previous observation that increased model complexity can enhance performance in capturing the underlying relationship.

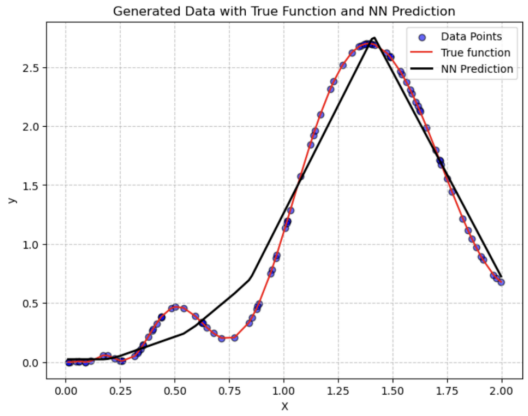

Figure 5: Scatterplot showing original data points in blue. Red line represents the true function. In black the NN predictions.

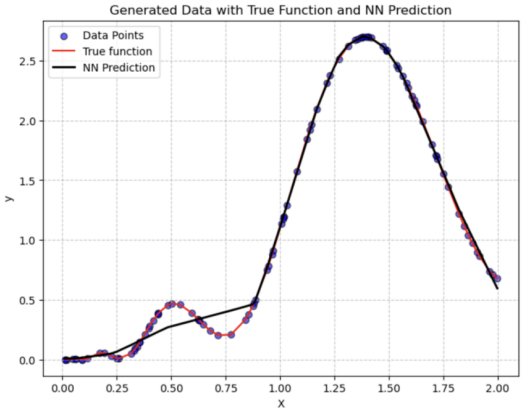

Figure 6: Scatterplot showing original data points in blue. Red line represents the true function. In black the NN predictions.

When using Scikit-learn to implement the neural network with mini-batch gradient descent as the optimization method, the results differ significantly. In this case, increasing the network complexity does not enhance accuracy. The lowest MSE is achieved with a configuration that includes two hidden layers, yielding an MSE of 0.3013 and an R^2 value of 0.4594.

The following results regard the implementation of the Neural Network using Adam as an optimization method. Overall, Adam appears to enhance the performance of simpler networks, allowing them to achieve good results more quickly. Employing a high number of hidden layers tends to be slightly less efficient. The best outcome is achieved when 1 hidden layer is employed and sigmoid is used as an activation function. The MSE on the validation set is = 0.0002 and the R^2 = 0.9996. ReLu in contrast with what has been seen so far does not increase the effectiveness of the process.

Figure 7 highlights how well the Neural Network models the true relationship present in the data.

Compared to the previous plots, here is clear how Adam offers a further advantage in optimizing the neural network's capacity to fit the underlying function. The model's ability to closely follow the true relationship indicates that the Adam optimizer effectively minimizes the loss function, leading to a more accurate representation of the data. While complexity can be beneficial in certain contexts, a well-tuned model with fewer hidden layers seems to often yields superior results.

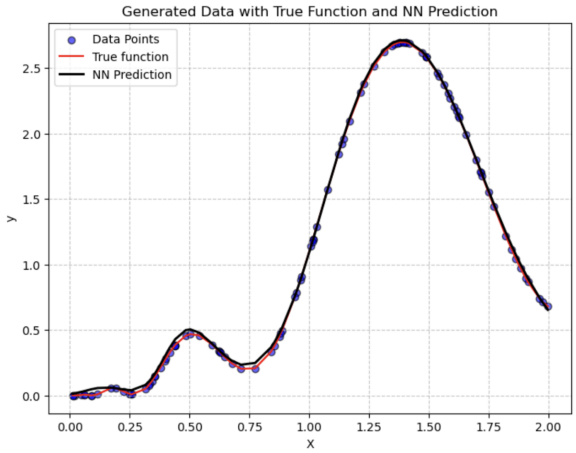

Figure 7: Scatterplot showing original data points in blue. Red line represents the true function. In black the NN predictions.

When Scikit-learn is employed to implement the Neural Network with Adam the performance is not enhanced and simpler configurations are preferred.

We now present the results obtained by applying the L2 Regularization to our Neural Network. In this case, a Neural Network with 1 hidden layer of 50 nodes and ReLu as activation function is employed. Figure 8 shows how the validation MSE changes according to different lambda values and learning rates. It's clear how bigger values of learning rate allow a more efficient process. It is also clear how strong regularizations are less favored. The peak performance, with an MSE = 1.5068 is reached with a learning rate of 0.1 and a lambda value of 0.001. We assessed how a low regularization is preferred. Next, we compare these results to those of the same neural network architecture implemented without L2 regularization. This comparison aims to evaluate the impact of regularization and it will help understand if a slight regularization can be beneficial over no regularization at all. Both approaches are tested on the simple function $y = 4 + 3X + 2X^2$ with added random noise. In the previous analysis, the neural network without regularization achieved an MSE of 0.9977, indicating that regularization may not have been beneficial in this case. This could be attributed to the simplicity of the task or other factors, and it is certainly an area that deserves further investigation in our future studies.

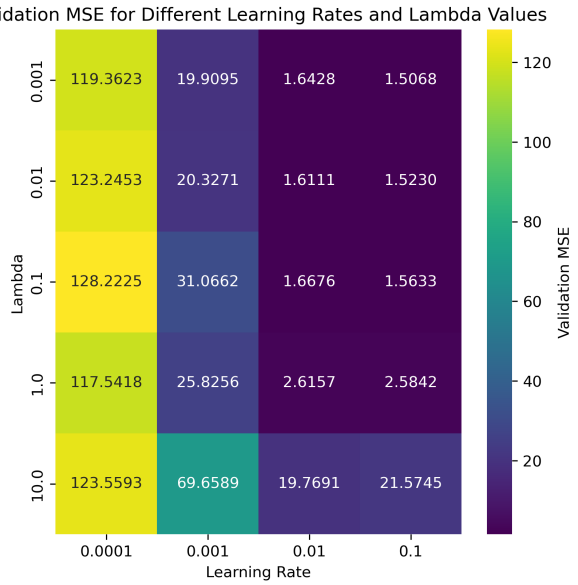

Figure 8: This heatmap displays the validation cross entropy of the Logistic Regression model across different values of the regularization parameter (lambda) and learning rates. Each cell indicates the binary cross entropy achieved with a specific combination of lambda and learning rate.

3.1.5 Final evaluation on Test Set

We then evaluated the neural network trained on the complex function using the Adam optimization method. This model has one hidden layer with 50 nodes and utilizes the ReLu activation function. When applied to the test data, the model achieved a MSE of 0.0081 on unseen data. This result indicates that the neural network is effectively capturing the underlying patterns of the complex function. The low MSE suggests that the model is well-tuned and capable of accurately predicting outcomes beyond the training dataset.

Further analysis could explore the impact of different hyperparameter settings on this performance, as well as a comparison with other optimization methods to ascertain the robustness of these findings.

3.2 Classification

In this section, we present the results of our classification experiments conducted with neural networks and logistic regression. We begin by testing both algorithms on two synthetic datasets. Each dataset consists of two distinct classes: in the first dataset, the classes are perfectly separable. As expected, both the neural network and logistic regression models achieved an accuracy of 1, regardless of whether Ridge regularization is applied, with confusion matrices confirming that all points were correctly classified.

The second dataset contained slightly overlapping classes, resulting in a few misclassifications and a reduction in accuracy, which aligns with our expectations (Figure 9).

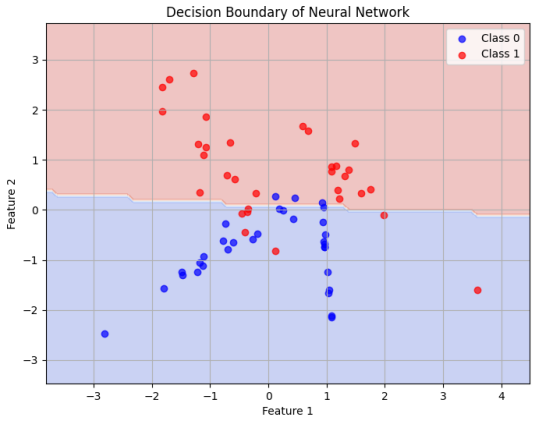

Figure 9: This plot shows the decision boundary created by the neural network, with blue points representing Class 0 and red representing Class 1. The shaded regions indicate the predicted classes based on the two input features.

Next, we apply our models to a practical, real-world dataset: the Wisconsin Breast Cancer dataset, which involves a binary classification task.

3.2.1 Neural Networks

First, we present the results of our neural network, tested across various combinations of nodes and layers. Performance either decreased or remained the same as when using a single hidden layer with 2, 3, or 4 nodes, so we focused our analysis on simpler neural network structures. Since the Scikit-learn function allows the specification of only one type of activation function for both the hidden and output layers, we use the sigmoid function for the hidden layers as well to be able to directly compare the results.

The batch size is set at 32, the epochs to 100 and the learning rate is 0.01. Before training, the initial validation accuracies were respectively 0.6044, 0.6044, and 0.4176 and the confusion matrices showed many misclassifications.

After training, the accuracy increases to exactly 0.9560 for all tests. The confusion matrix for the configuration with 2 nodes becomes:

```
Confusion Matrix after Training:
          Predicted 0  Predicted 1
Actual 0           32            4
Actual 1            0           53
```

For the tests with 3 and 4 nodes:

```
Confusion Matrix after Training:
          Predicted 0  Predicted 1
Actual 0           32            3
Actual 1            1           53
```

These results indicate that, according to the accuracy, a very simple network structure with two nodes can yield optimal classification results and the logistic function is a good choice for activation.

Similar results for tests having 2 and 3 nodes were found using Scikit-learn's function. The final validation accuracy is 0.9560 but there is a slight difference in the confusion matrix.

```
Confusion Matrix after Training:
          Predicted 0  Predicted 1
Actual 0           32            2
Actual 1            2           53
```

For 4 nodes test, the final accuracy increases up to 0.9670.

```
Confusion Matrix after Training:
          Predicted 0  Predicted 1
Actual 0           32            2
Actual 1            1           54
```

In this case, using 4 nodes in the hidden layer may be more appropriate in terms of accuracy. However, since the improvement is not drastic and 2 nodes still perform well, it might be preferable to maintain a simpler structure. We also applied a neural network with L2 regularization, testing with several combinations of lambda and learning rates to observe how accuracy changes.

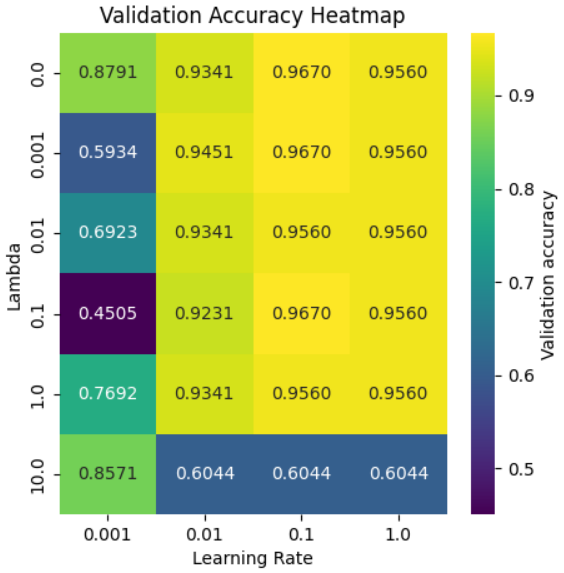

Figure 10: This heatmap displays the validation accuracy of the neural network across different values of the regularization parameter (lambda) and learning rates. Each cell indicates the accuracy achieved with a specific combination of lambda and learning rate.

From Figure 10, it is evident that the overall accuracy of our model is quite high. It is clear that accuracy increases with higher learning rates and that a low lambda value or no regularization at all is preferable. For lambda values of 0.1 and 0.001 with a learning rate of 0.1, the model's performance is equal to the one achieved when no regularization is applied using Scikit-learn. While L2 regularization can improve model generalization, our findings suggest that for this particular setup lower regularization (or none) combined with appropriate learning rates yields optimal performance without significant loss in accuracy.

3.2.2 Logistic Regression

In addition, we applied Logistic Regression to both the synthetic datasets to test our code and to the Wisconsin Breast Cancer dataset. We implemented Gradient Descent, Stochastic Gradient Descent, and Mini-batch Gradient Descent.

As expected, the accuracy is 1 for the perfectly separated classes dataset. In the second case, the accuracy decreases slightly. Since the boundaries are linear there are some misclassified points (Figure 11).

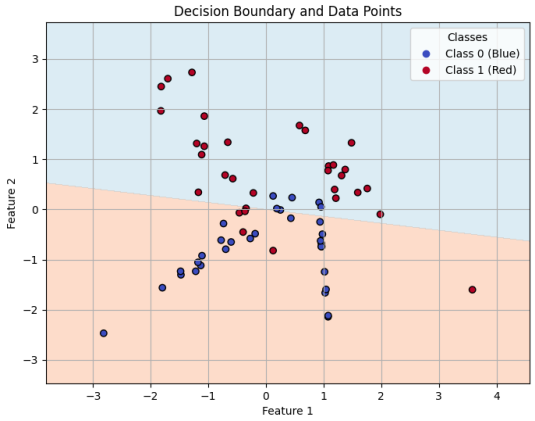

Figure 11: This plot shows the decision boundary created by Logistic Regression, with blue points representing Class 0 and red representing Class 1. The shaded regions indicate the predicted classes based on the two input features.

Applying L2 Regularization yields, in bith synthetic datasets, the same results. When the models are applied to the real-world dataset, the accuracies after training are as follows: GD achieves an accuracy of 0.9231, which is relatively lower compared to the others. In contrast, both SGD and its Scikit-learn implementation reach an accuracy of 0.9670, indicating strong performance. Mini-batch GD also performs well, with an accuracy of 0.9560. These results suggest that while GD is less effective, SGD and MGD provide more reliable predictions. An accurate analysis of the influence of the penalty term lambda, combined with the tuning of the learning rate, has been performed for each method. We plot a heatmap showing the loss function across all combinations, which can be found in the additional material. Here, we focus on the analysis of Mini-batch Gradient Descent (Figure 12):

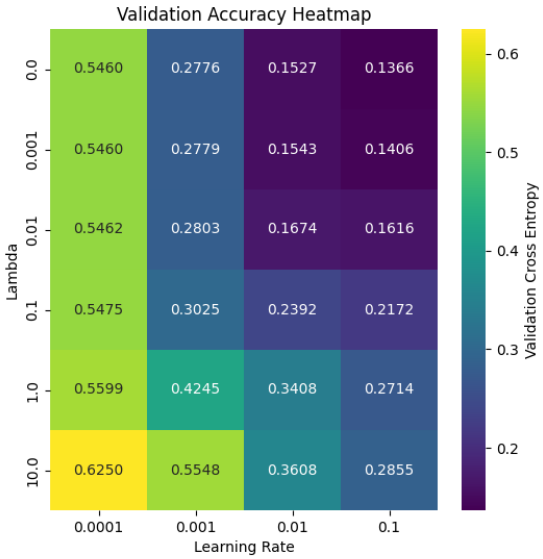

Figure 12: This heatmap displays the validation cross entropy of the Logistic Regression model across different values of the regularization parameter (lambda) and learning rates. Each cell indicates the binary cross entropy achieved with a specific combination of lambda and learning rate.

Lower values are preferable, and it is evident that models with lower regularization or no regularization at all perform better. In contrast, higher learning rates contribute to improved model performance. Similar behavior is found on GD investigation but on the other hand, SGD prefers a lower learning rate (0.001) to minimize the cross entropy.

3.2.3 Final evaluation on Test Set

All Logistic Regression results are obtained by setting the hyperparameters to the same values used in the neural network analysis. Among the networks, only the one with 4 nodes implemented using Scikit-learn, achieves the same accuracy as the Logistic Regression models with SGD.

We apply the test set to both models. The Scikit-learn model with 4 nodes and the Logistic Regression model yields the same results, exhibiting a low accuracy of 0.3772, which is unexpected since these models are selected based on their performance on the validation set. The low accuracy observed in both the Logistic Regression model and the neural network with 4 nodes might be attributed to a small sample size.

The test confusion matrix reveals that the model fails to correctly identify any actual positive instances, predicting all as negative (0 true positives). While it accurately predicts all 43 actual negative instances as negative, the lack of correct predictions for positive cases highlights a significant issue in the model's performance on the test set.

```
 Confusion Matrix after Training:
          Predicted 0  Predicted 1
Actual 0           43            0
Actual 1            71            0
```

4 Discussion

The presented results offer valuable insights on model selection and hyperparameters tuning, both in regression and in classification tasks. We have demonstrated more clearly how neural networks compare to traditional regression methods, such as OLS and Ridge, as well as to logistic regression in classification scenarios. Generally it is clear how the choice of the model, its architecture, its optimization methods and the hyperparameters are strongly tied to the complexity of the task. For the relatively simple tasks investigated, the neural network models did not outperform simpler algorithms, suggesting that less complex architectures can achieve comparable performance when task complexity is low and may be preferable due to their greater interpretability. Additionally, this analysis indicates that regularization methods, though effective in Project 1, did not yield the same benefits here, possibly due to the limited complexity of the tasks. Further investigation is required to fully understand the conditions under which regularization can contribute to neural network performance.

Different optimization methods have been tested in OLS and Ridge regression, with no clear preference emerging for any specific method. All methods were able to estimate the real function parameters well. Adaptive methods have been evaluated as efficient, but remaining context-dependent in performance. A higher number of iterations has been proved to increase the accuracy of the model at a higher computational cost. Additionally, using larger mini-batch sizes has led to more representative sampling of the dataset, resulting in enhanced model performance.

Our experiments with neural networks revealed that they require a thorough training process, with careful consideration of all potential parameters and architectural choices. Unlike simpler models, neural networks exhibited variable behavior across training conditions, with no consistently optimal configuration across all tasks. For instance, changes to the activation function or optimization method yielded different outcomes depending on the network's complexity, improving performance in some scenarios while deteriorating it in others. This underscores the importance of tailoring hyperparameters and architecture to the specific context of each task, as no single configuration universally optimizes network performance. Generally both in regression and in classification higher learning rates are clearly favorable. In regression or classification a significantly different number of nodes per hidden layer were employed to achieve satisfactory results.

During regression tasks, more complex neural networks demonstrated a strong ability to fit the original function compared to some simpler networks with varying configurations. However, a well-tuned configuration of simpler networks was often able to achieve even better fitting performance. Therefore, simpler models are generally preferred due to their interpretability and lower risk of overfitting. This behavior was also observed in the case of the complex function, indicating that neural networks possess significant potential and can be effectively utilized in much more complex scenarios.

Generally, the same considerations apply to classification tasks. Simpler network architectures tend to classify cases as effectively as logistic regression models, likely due to the abundance of features present in the dataset. It would be interesting to investigate whether a reduced number of features could impact performance and maybe favor neural networks. All SGD demonstrating slightly better performance compared to both mini-batch and full-batch Gradient Descent approaches.

The application of the best-performing algorithms on the test data yielded satisfactory results for regression tasks, although the accuracy in classification tasks was less impressive. Given the limited size of the dataset, it will be interesting to investigate whether these results improve with a larger number of data points.

5 Conclusion

This project highlighted how simpler models both in regression and classification are often the preferred choice in less complex scenarios. This preference is due not only to their comparable performance to neural networks but also to their greater interpretability and lower computational and time requirements. Additionally, it has been demonstrated that neural networks exhibit less predictable behavior during training. Exploring various architectures and parameters necessitates more extensive experimentation.

In more detail, the best optimization algorithm for OLS linear regression was found to be SGD, implemented using Scikit-learn. For Ridge regression, MGD with momentum resulted in improved performance. Regarding the complex function, the optimal Neural Network architecture was identified as having one hidden layer with 50 nodes, utilizing Adam as the optimization method and ReLu as the activation function. The performance of this model on the test data yielded a MSE of 0.0081.

For classification tasks in real-world settings, all models, both logistic regression and neural networks, performed effectively, achieving accuracy levels above 0.9. The logistic regression model implemented with SGD achieved the highest accuracy (0.9670), matching the performance of the neural network with a four-node hidden layer.

The key takeaway from this project is clearly that a good understanding of the data and the problem to be solved is essential. As remarked also in Project 1 parameters tuning and model architecture needs to be fine-tuned according to the characteristics of each model and to the complexity of the task. Moreover it has been particularly interesting to understand and explore different optimization methods, neural networks and their training process both in regression and classification.

A possible extension of this analysis could definitely head toward a more complex task, for which we expect to see neural networks outperform in a more decisively way simpler algorithms. Furthermore, there are numerous architectural and parameter settings that were not explored in this project, which could enhance the analysis. These include varying the number of nodes in each hidden layer and experimenting with different activation functions. Another aspect that deserves more in depth analysis is the regularization applied to Neural Networks as it did not produce the expected results. Throughout the experiments, lower levels of regularization were consistently favored, not significantly reducing model complexity and failing to outperform configurations without regularization.

6 Appendix

6.1 Source Code

All codes are available in the GitHub repository: Project-2-Proetto-Ruffoni-FYS-STK3155.

Due to the presence of multiple random initializations and stochastic processes, the results may vary slightly across different runs. However, we have made every effort to control for these variations by setting a seed throughout the code to ensure reproducibility as much as possible.

6.1.1 Linear Regression: Ordinary Least Squares

For the implementation and testing of Linear Regression using Ordinary Least Squares, refer to the following resources:

Test Code for OLS
Data Analysis Code for OLS

6.1.2 Linear Regression: Ridge Regression

For further analysis using Ridge Regression, see the complete project repository:
Test Code with Ridge Regularization
Data Analysis Code for Ridge

6.1.3 Regression: Neural Network

For additional exploration of Neural Network Classification, please refer to the full project repository:
Test Code Neural Network
Data Analysis Code for Neural Network
Test Code Neural Network with L2 Regularization
Data Analysis Code for Neural Network with L2 Regularization
Data Analysis Code on a Simple Function
Data Analysis Code on a more Complex Function

6.1.4 Classification: Neural Network

For further analysis using Neural Network Regression, see the complete project repository:
Test Code Neural Network
Breast Cancer Dataset Analysis
Test Code with L2 regularization
Breast Cancer Dataset Analysis with L2 regularization

6.1.5 Classification: Logistic Regression

For more detailed analysis using Logistic Regression, consult the complete project repository:
Test Code Logistic Regression and Ridge Regularization
Breast Cancer Dataset Analysis Logistic Regression and Ridge Regularization

6.2 Additional Tables

	SGD MSE	SGD R^2	SGD-Scikit MSE	SGD-Scikit R^2	ADAM MSE	ADAM R^2	ADAM-Scikit MSE	ADAM-Scikit R^2
Analysis A	0.0972	0.9943	0.004	0.9998	0	1	0.0037	0.9998
Analysis B	0.0108	0.9994	/	/	0.0016	0.9999	/	/
Analysis C	0.0059	0.9997	0.0055	0.9997	0.0001	1	18.9465	-0.1067
Analysis D	0.0007	1	/	/	0.0019	0.9999	/	/
Analysis E	0.0051	0.9997	0.0328	0.9981	0.2114	0.9874	18.3815	-0.0737
Analysis F	0.0051	0.9997	/	/	0.0049	0.9997	/	/
Analysis G	0.0133	0.9992	20.1846	-0.179	20.3586	low negative value	18.2791	-0.0677
Analysis H	0.0006	1	/	/	18.5935	low negative value	/	/

Table 8: Results of Regression Neural Network applied on simple function without noise.

	SGD MSE	SGD R^2	SGD-Scikit MSE	SGD-Scikit R^2	ADAM MSE	ADAM R^2	ADAM-Scikit MSE	ADAM-Scikit R^2
Analysis A	1.3309	0.9164	1.0147	0.9445	2.0633	0.8851	1.095	0.9475
Analysis B	0.9977	0.9401	/	/	0.9966	0.942	/	/
Analysis C	1.0345	0.9372	1.0147	0.9445	2.1645	0.875	20.3294	-0.1117
Analysis D	0.9842	0.9404	/	/	2.8943	0.8324	/	/
Analysis E	1.0283	0.9374	1.1512	0.937	2.9211	0.8361	19.7027	-0.0774
Analysis F	1.1154	0.9313	/	/	2.9211	0.8611	/	/
Analysis G	1.022	0.9381	21.5881	-0.1805	2.8275	0.8399	19.4894	-0.0658
Analysis H	0.9958	0.9418	/	/	22.0235	low negative value	/	/

Table 9: Results of Regression Neural Network applied on simple function with noise.

	SGD MSE	SGD R^2	SGD-Scikit MSE	SGD-Scikit R^2	ADAM MSE	ADAM R^2	ADAM-Scikit MSE	ADAM-Scikit R^2
Analysis A	0.0244	0.09563	0.4301	0.2283	0.013	0.9734	0.2988	0.4638
Analysis B	0.0264	0.5264	/	/	0.0002	0.9996	/	/
Analysis C	0.2643	0.5258	0.3013	0.4594	0.2643	0.5258	0.1944	0.6511
Analysis D	0.2279	0.591	/	/	0.0109	0.998	/	/
Analysis E	0.2658	0.5231	0.7853	-0.4092	0.0005	0.9991	0.7953	-0.427
Analysis F	0.09	0.9838	/	/	0.8126	0	/	/
Analysis G	0.7864	-0.4112	0.83881	-0.0018	0.76188	low negative value	0.77	-0.3816
Analysis H	0.0134	0.9759	/	/	0.0087	0.9832	/	/

Table 10: Results of Regression Neural Network applied on complex function.

6.3 ChatGPT Usage

As for Project 1, ChatGPT was used to improve the clarity and precision of the language, as English is not the authors' native language. Its assistance was limited to suggesting synonyms, refining sentence structures, and occasionally verifying the correctness of various method implementations. ChatGPT was never used to generate paragraphs or contribute directly to the text, nor was it involved in the creation of functions implemented in the analysis. The code was adapted and modified based on materials provided in lecture notes, assignments, and the code developed in Project 1. ChatGPT's use has been sparse and occasional. Link to ChatGPT conversation

References

[1] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

[2] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. "Deep Learning". In: *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016. Chap. 5.

[4] Mohaiminul Islam, Guorong Chen, and Shangzhu Jin. "An Overview of Neural Network". In: 5 (June 2019), p. 05. DOI: 10.11648/j.ajnna.20190501.12.

[5] Mert Alagözlü. "Stochastic Gradient Descent Variants and Applications". In: (Feb. 2022). DOI: 10.13140/RG.2.2.12528.53767.

[6] Atharva Tapkir. "A Comprehensive Overview of Gradient Descent and its Optimization Algorithms". In: *IARJSET* 10 (Nov. 2023). DOI: 10.17148/IARJSET.2023.101106.

[7] Morten Hjorth-Jensen. *Gradient descent methods and start Neural networks*. URL: https://github.com/CompPhysics/MachineLearning/blob/9c19ea9bfdb1ed64ab2106e43d19b4b75c324802/doc/LectureNotes/week40.ipynb.

[8] Morten Hjorth-Jensen. *Logistic Regression and Optimization, reminders from week 38 and week 40*. URL: https://github.com/CompPhysics/MachineLearning/blob/9c19ea9bfdb1ed64ab2106e43d19b4b7
doc/LectureNotes/week42.ipynb.

[9] Morten Hjorth-Jensen. *Optimization and Gradient Methods*. URL: https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week39.ipynb.