FYS-STK Project 2

Herman Nikolai Scheele, Theodor Jaarvik, Elaha Ahmadi

October 2024

Abstract

In this project, we explore and critically evaluate various machine learning algorithms for regression and classification tasks. We implement our own versions of gradient descent (GD) and stochastic gradient descent (SGD) algorithms, including enhancements like momentum, Adagrad, RMSprop, and Adam optimizers. These methods are applied to linear regression models such as Ordinary Least Squares (OLS) and Ridge regression. We also develop a custom feed-forward neural network (FFNN) from scratch for both regression and classification problems, experimenting with different activation functions and network architectures. For classification, we implement logistic regression using SGD and compare it with our neural network. Our main findings indicate that traditional linear regression models optimized with GD variants perform efficiently and accurately for regression tasks. In contrast, the FFNN demonstrates superior performance in classification tasks due to its ability to capture complex non-linear patterns. These results underscore the importance of selecting appropriate algorithms based on the nature of the problem and provide insights into the strengths and limitations of each method.

1 Introduction

Machine learning has become an integral part of data analysis and predictive modeling, offering powerful tools for tackling a wide range of problems in science, engineering, and industry. Regression and classification are fundamental tasks within machine learning, essential for predicting continuous outcomes and categorizing data into discrete classes, respectively. The choice of algorithm for these tasks significantly impacts the performance and efficiency of the solutions.

Despite the abundance of machine learning algorithms available, understanding their underlying mechanisms and comparative performance remains a challenge. This project aims to address this by implementing and critically evaluating various algorithms for regression and classification. By developing custom implementations of gradient descent algorithms and feed-forward neural networks (FFNNs), we seek to gain deeper insights into their functionality, advantages, and limitations.

We begin by replacing the traditional matrix inversion methods in linear regression models with our own gradient descent (GD) and stochastic gradient descent (SGD) algorithms. Enhancements such as momentum, Adagrad, RMSprop, and Adam optimizers are incorporated to improve convergence and performance. These methods are applied to regression tasks using synthetic data generated from quadratic functions.

Next, we develop a custom FFNN for regression, experimenting with different activation functions—Sigmoid, ReLU, and Leaky ReLU—and network architectures. The FFNN is then adapted for classification tasks, specifically applied to the Wisconsin Breast Cancer dataset, to assess its performance against logistic regression implemented with our SGD algorithm.

The report is organized as follows: In Section 2, we detail the methodologies employed, including data generation, algorithm implementations, and model evaluation metrics. Section 3 presents the results obtained from each algorithm and discusses their performance. In Section 4, we provide a critical evaluation of the various algorithms, summarizing their pros and cons. Finally, in Section 5, we conclude with our main findings and discuss potential avenues for future work, including possible improvements and applications.

2 Methods

2.1 Data Generation and Design Matrix

A synthetic dataset was generated based on a quadratic function with added Gaussian noise, represented by:

$$y = 2 + 3x + 4x^2 + 0.1 \cdot \text{noise}$$

where x values were randomly generated within a specified range, and noise was added to introduce variability. The data generation process was implemented as follows:

```python
def generate_data(n_samples=100):
    np.random.seed(0)
    x = np.random.rand(n_samples)
    y = 2 + 3*x + 4*x**2 + 0.1 * np.random.randn(n_samples)
    return x, y
```

Figure 1: Code for generating a synthetic dataset based on a quadratic function with added Gaussian noise

The generated dataset was used as the basis for testing various optimization techniques. To facilitate polynomial regression, a design matrix X was constructed with polynomial terms up to x^2, as shown in the following code snippet:

```
X = np.c_[np.ones(x.shape), x, x**2]  # For polynomial terms (1, x, x^2)
y = y.reshape(-1, 1)
```

Figure 2: Construction of the design matrix X with polynomial terms up to x^2 for quadratic regression.

A design matrix X was constructed to include polynomial terms up to x^2, structured as:

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{bmatrix}$$

where n is the number of samples. This setup allowed the model to capture the quadratic relationship in the data.

2.2 Gradient Decent Algorithms

Gradient Descent (GD) and Stochastic Gradient Descent (SGD) were employed as primary optimization techniques to minimize the Mean Squared Error (MSE) by iteratively updating parameters.

The classic gradient descent approach minimizes the cost function by taking steps in the direction of the steepest descent, defined by the negative gradient of the MSE. Starting with initial parameter values, each iteration adjusts the parameters θ according to:

$$\theta \leftarrow \theta - \eta \nabla_\theta \text{MSE}$$

where η is the learning rate, determining the step size. In Figure 3, the gradient descent trajectory is visualized, showing how parameters iteratively approach the optimal solution.

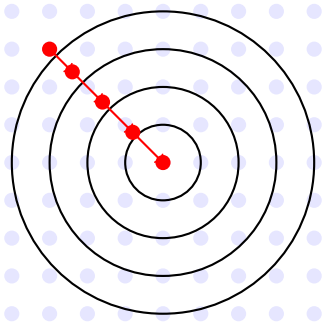


Figure 3: Visualization of Gradient Descent optimization path on a convex cost surface

To improve computational efficiency, especially with large datasets, Stochastic Gradient Descent (SGD) was employed. Instead of calculating the gradient across the entire dataset, SGD uses a random subset (mini-batch) at each iteration. The parameter update for SGD is given by:

$$\theta \leftarrow \theta - \eta \nabla_\theta \text{MSE}_{\text{batch}}$$

where $\text{MSE}_{\text{batch}}$ represents the MSE computed over the mini-batch. This stochastic approach introduces some noise in the updates, which can help the algorithm escape shallow local minima and improve generalization.

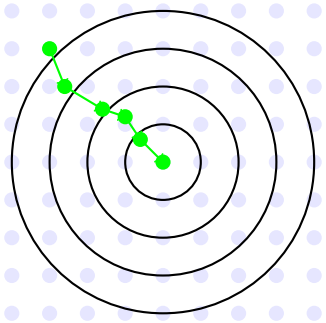


Figure 4: Visualization of Stochastic Gradient Descent path with noise due to mini-batch updates

These visualizations highlight the different optimization trajectories of GD and SGD. While GD steadily progresses towards the minimum, SGD's path is more erratic due to the mini-batch updates, potentially allowing faster convergence and escape from local minima.

Several optimization techniques were applied to estimate the model parameters. Gradient Descent (GD) was initially implemented to minimize the Mean

Squared Error (MSE) by updating parameters iteratively in the direction of the negative gradient. The code for the basic gradient descent algorithm is provided below:

```python
def gradient_descent(X, y, learning_rate=0.01, n_iterations=1000):
    n_samples, n_features = X.shape
    theta = np.random.randn(n_features, 1)
    for i in range(n_iterations):
        gradients = (2 / n_samples) * X.T @ (X @ theta - y)
        theta -= learning_rate * gradients
    return theta
```

Figure 5: Implementation of basic gradient descent to optimize model parameters for ordinary least squares regression

$$\theta \leftarrow \theta - \eta \nabla_\theta \text{MSE}$$

where η is the learning rate. To enhance convergence speed and stability, momentum was introduced into GD, modifying the update to incorporate past gradients:

$$v \leftarrow \gamma v + \eta \nabla_\theta \text{MSE}, \quad \theta \leftarrow \theta - v$$

where γ is the momentum parameter. Stochastic Gradient Descent (SGD) with mini-batches was also applied, dividing the data into small random batches to improve computational efficiency, especially for larger datasets.

Adaptive learning rate methods, including Adagrad, RMSprop, and Adam, were also tested. Adagrad adjusted the learning rate based on the cumulative sum of squared gradients for each parameter:

$$\theta \leftarrow \theta - \frac{\eta}{\sqrt{G + \epsilon}} \nabla_\theta \text{MSE}$$

where G accumulates squared gradients and ϵ prevents division by zero. RMSprop further improved this by using a decaying average of squared gradients:

$$G \leftarrow \beta G + (1 - \beta) \nabla_\theta \text{MSE}^2$$

where β controls the decay rate. Adam combined both momentum and RMSprop, maintaining decaying averages of gradients and squared gradients for more stable updates, with parameter updates given by:

$$\theta \leftarrow \theta - \frac{\eta}{\sqrt{\hat{v} + \epsilon}} \hat{m}$$

where $\hat{m}$ and $\hat{v}$ are bias-corrected estimates of the gradient and squared gradient, respectively.

2.2.1 Model Evaluation

Each optimization method was evaluated based on the Mean Squared Error (MSE) and R^2 score, where MSE is calculated as:

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

and the R^2 score is defined as:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

with $\bar{y}$ as the mean of actual values. Higher R^2 scores and lower MSE values indicated a better fit to the data. Finally, model predictions from each method were plotted against actual data to visually compare the performance of the optimizers.

2.3 Neural Networks

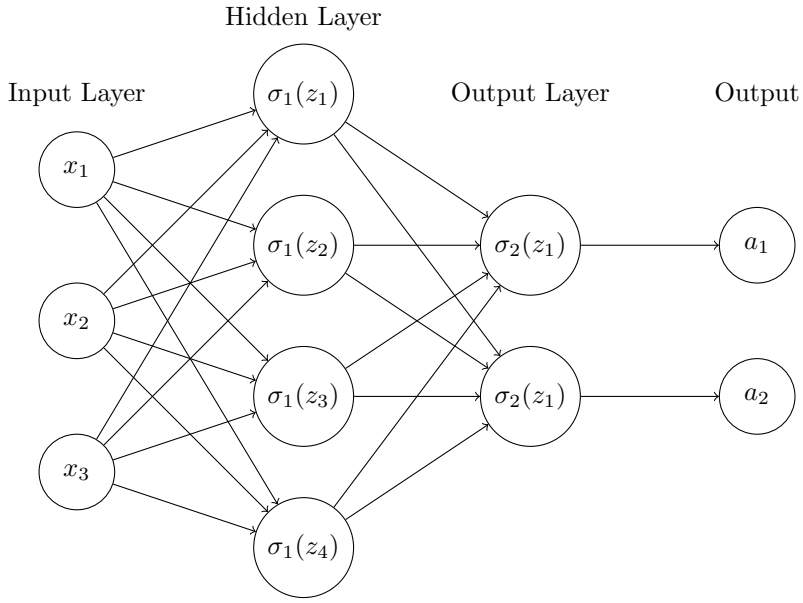
Artificial Neural Networks are computational models inspired by the structure and function of biological neural networks in the human brain. They consist of interconnected groups of artificial neurons that process information using a connectionist approach to computation. They have the ability to learn from data, making them powerful tools for modeling complex patterns and predicting outcomes in various domains.

Over the past few decades, neural networks have evolved significantly, leading to the development of deep learning architectures. Deep Neural Networks, which contain multiple hidden layers, have achieved remarkable success in fields such as image and speech recognition, natural language processing, and autonomous systems. This success is largely attributed to advances in computational power, the availability of large datasets, and improved training algorithms.

For this project we created a fully manual neural network code including initialization of layers, forward propagation, backward propagation and training of the network.

2.3.1 Neural network architecture

An artificial neural network is essentially a structure that emerges from specific utilization's of linear algebra, but we can describe them visually like the following



With linear algebra we define our data for this case as the input vector x.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

The lines in the visual representation is mathematically represented as a weight matrix

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix}$$

The amount of rows represent the amount of nodes we want and the columns represent the amount of elements in our input vector.

z is the linear combination with the mathematical expression

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

b is the bias and is the vector with the same dimensionality as the number of nodes for a given hidden layer

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

For all linear combination for our nodes we put them through an activation function, which in this case is the sigmoid function. The activation function introduces non-linearity to our network which is essential to differentiate our hidden layers and capture non-linear relationships.

2.3.2 Forward propagation

Forward propagation is the process by which input data is passed through a neural network to produce an output. In each layer, the input vector is transformed by a series of weights and biases, and then an activation function is applied to introduce non-linearity. Mathematically, for a single layer, the forward propagation can be represented as:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

where: - $\mathbf{x}$ is the input vector, - $\mathbf{W}$ is the weight matrix, - $\mathbf{b}$ is the bias vector, - $\mathbf{z}$ is the resulting output before activation.

The output $\mathbf{z}$ is then passed through an activation function σ to produce the layer's final output:

$$\mathbf{a} = \sigma(\mathbf{z})$$

Forward propagation continues layer by layer until the final output layer is reached, providing a prediction for the given input. The final output-layer will vary depending on what sort task you are dealing with. We will se later that for the regression task we remove activations for the final layer completely, but for this example the output will be a two dimensional vector

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \sigma_2(z_1) \\ \sigma_2(z_2) \end{bmatrix}$$

2.3.3 Backpropogation

Backpropagation is a key algorithm in training neural networks, allowing the model to adjust its weights to minimize prediction errors. It involves two main steps: forward propagation to compute the network's predictions, followed by backward propagation to calculate the error gradients with respect to each weight.

In the backward pass, the loss L is differentiated with respect to each weight in the network using the chain rule. The gradient can be expressed as:

$$\frac{\partial L}{\partial W_{ij}^{(l)}} = \frac{\partial L}{\partial z_i^{(l)}} \cdot \frac{\partial z_i^{(l)}}{\partial W_{ij}^{(l)}}$$

These gradients are then used to update the weights to reduce the error:

$$W \leftarrow W - \eta \frac{\partial L}{\partial W}$$

where η is the learning rate, controlling the size of the updates. This iterative process continues until the model achieves acceptable accuracy.

2.3.4 Training our neural network

For this task we made a manual code for a neural network including flexible number of hidden layers, amount of nodes in each layer and activation functions. The task is a regression task of the second-order polynomial using our neural network, so the final output layer will have no activation function as we want no boundary ranges for the output data.

First we initialized our network with normally distributed weights. We then initialized the biases as zero. Since biases serve as additional adjustable parameters that shift the activation functions, starting them at zero ensures that all neurons in the layer initially have an equal contribution, thereby standardizing the starting point for optimization.

We implemented the initialization in code as the following

```python
# Initialize network
def initialize_network(n_inputs, n_hidden_layers, nodes_per_layer, n_outputs, activations):
    np.random.seed(0)
    network = {}
    layer_sizes = [n_inputs] + nodes_per_layer + [n_outputs]

    for layer in range(n_hidden_layers + 1):
        network[f"W{layer}"] = np.random.randn(layer_sizes[layer], layer_sizes[layer + 1]) * 0.1
        network[f"b{layer}"] = np.zeros((1, layer_sizes[layer + 1]))
        network[f"activation{layer}"] = activations[layer]

    return network
```

Figure 6: Implementation of initialization of network

We implemented the forward propegation algorithm like the following

```python
# Forward propagation
def forward_propagation(X, network):
    activations = {"A0": X}
    for layer in range(len(network) // 3):
        W = network[f"W{layer}"]
        b = network[f"b{layer}"]
        act_fn, _ = activation_functions[network[f"activation{layer}"]]

        Z = activations[f"A{layer}"] @ W + b
        activations[f"A{layer + 1}"] = act_fn(Z) if layer < len(network) // 3 - 1 else Z

    return activations
```

Figure 7: Implementation of forward propagation

And finally, here is the implementation for the backward propagation, the parametric updates and training of the network. We used Stochastic Gradient Decent as optimization method.

```python
# Backward propagation
def back_propagation(y, activations, network):
    gradients = {}
    n_layers = len(network) // 3
    y_pred = activations[f"A{n_layers}"]

    dA = y_pred - y
    for layer in reversed(range(n_layers)):
        _, act_derivative = activation_functions[network[f"activation{layer}"]]
        dZ = dA if layer == n_layers - 1 else dA * act_derivative(activations[f"A{layer + 1}"])
        dW = activations[f"A{layer}"].T @ dZ / y.shape[0]
        db = np.sum(dZ, axis=0, keepdims=True) / y.shape[0]

        gradients[f"dW{layer}"] = dW
        gradients[f"db{layer}"] = db
        if layer > 0:
            dA = dZ @ network[f"W{layer}"].T

    return gradients

# Update parameters
def update_parameters(network, gradients, learning_rate):
    for layer in range(len(network) // 3):
        network[f"W{layer}"] -= learning_rate * gradients[f"dW{layer}"]
        network[f"b{layer}"] -= learning_rate * gradients[f"db{layer}"]
    return network

# Training function with mini-batch SGD
def train_neural_network_sgd(X, y, network, n_epochs=1000, learning_rate=0.01, batch_size=16):
    n_samples = X.shape[0]
    losses = []

    for epoch in range(n_epochs):
        # Shuffle data at the start of each epoch
        indices = np.random.permutation(n_samples)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        epoch_loss = 0
        for start in range(0, n_samples, batch_size):
            end = start + batch_size
            X_batch = X_shuffled[start:end]
            y_batch = y_shuffled[start:end]

            # Forward pass
            activations = forward_propagation(X_batch, network)
            y_pred = activations[f"A{len(network) // 3}"]

            # Compute loss for the batch
            batch_loss = mse_loss(y_batch, y_pred)
            epoch_loss += batch_loss * len(y_batch)  # Accumulate loss over all batches

            # Backward pass
            gradients = back_propagation(y_batch, activations, network)

            # Update parameters
            network = update_parameters(network, gradients, learning_rate)

        # Record average loss for the epoch
        epoch_loss /= n_samples
        losses.append(epoch_loss)

        if epoch % 100 == 0:
            print(f"Epoch {epoch}, Loss: {epoch_loss:.4f}")

    return network, losses
```

Figure 8: Implementation of backward propagation, parametric updates and training of network

As the cost function for this task we decided to use the regular Mean Squared Error. This decision was made by evaluating what cost function would fit best for the regression task using neural networks. Like in project 1, we used MSE as it was the best cost function to evaluate how well our regression code fitted to the underlying data. Since our final output for the network is linear, we get the same type of output as a linear regression algorithm and that's why we believe MSE is the best cost function for this task.

```python
# Loss function
def mse_loss(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)
```

Figure 9: Implementation of Mean Square Error as the cost function

For comparison against well known machine learning frameworks, we chose to compare our manual neural network against PyTorch[1] for the exact same regression task. When testing our code against PyTorch, we got the following results

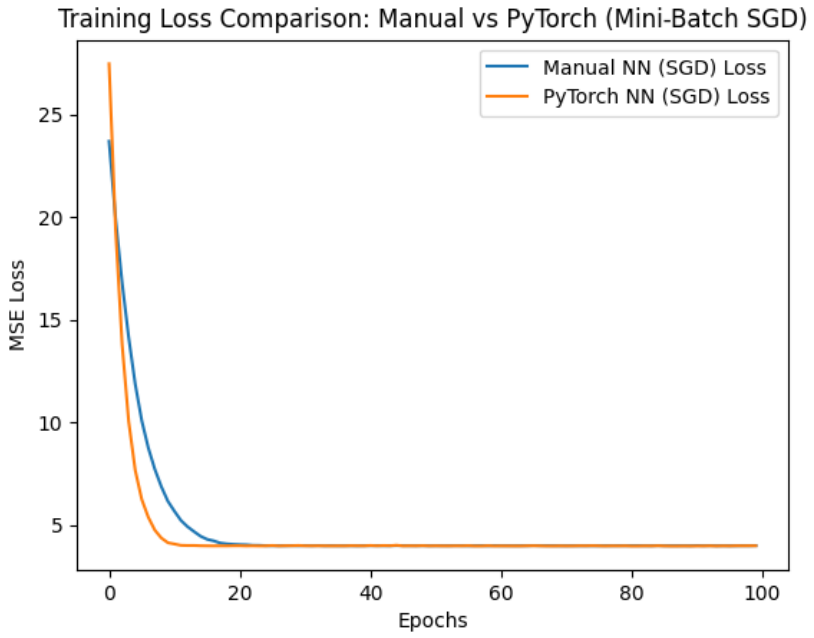


Figure 10: Manual and PyTorch comparison for the loss (MSE)

As expected, PyTorch preforms slightly better, but more importantly it has

the same pattern of development as the manual neural network. We can expect PyTorch to preform better as it is known to have better utilization of hardware to gain performance.

This solidifies our confidence that the network is correctly preforming as it should and gives us the ability to accurately discuss our results with confidence.

For testing different regularization parameters we slightly altered our backward propagation code implementation as the following

```python
# Backward propagation with L2 regularization
def back_propagation(y, activations, network, lambda_reg):
    gradients = {}
    n_layers = len(network) // 3
    y_pred = activations[f"A{n_layers}"]

    dA = y_pred - y
    for layer in reversed(range(n_layers)):
        _, act_derivative = activation_functions[network[f"activation{layer}"]]
        dZ = dA if layer == n_layers - 1 else dA * act_derivative(activations[f"A{layer + 1}"])
        dW = activations[f"A{layer}"].T @ dZ / y.shape[0] + (lambda_reg / y.shape[0]) * network[f"W{layer}"]
        db = np.sum(dZ, axis=0, keepdims=True) / y.shape[0]

        gradients[f"dW{layer}"] = dW
        gradients[f"db{layer}"] = db
        if layer > 0:
            dA = dZ @ network[f"W{layer}"].T

    return gradients
```

Figure 11: Backward propagation with L2 regularization

2.4 Testing activation functions

To test different activation functions for each hidden layer, various permutations of the activation functions were generated and iterated over. This was accomplished using a permutation iterator, which produced a list of lists containing every possible permutation of the activation functions. This approach allowed for a systematic comparison of activation functions, providing insights into the performance and behavior of each combination.

```
n_inputs = X.shape[1]
n_hidden_layers = 3
nodes_per_layer = [10, 10, 10]
n_outputs = 1
all_losses = []
optimal = [[], 100]
n_epochs = 100
learning_rate = 0.01

activations = ["sigmoid", "relu", "leaky_relu"]
combination_length = len(activations)
activation_permutations = list(product(activations, repeat=combination_length))
activation_permutations = [list(item) for item in activation_permutations]
for permutation in activation_permutations:
    permutation.append("linear")

for actv in activation_permutations:

    network = initialize_network(n_inputs, n_hidden_layers, nodes_per_layer, n_outputs, actv)
    network, sgd_losses = train_neural_network_sgd(X, y, network, n_epochs, learning_rate, batch_size=16)
    current = [actv, sgd_losses[n_epochs-1]]
    print(f"Configuration: {current[0]}, MSE: {current[1]}")

    if current[1] < optimal[1]:
        optimal = current
    all_losses.append(sgd_losses)
```

Figure 12: Implementation for creating a list of list for every permutation of the activation function list.

The code iterates over each set of activation functions from this list, builds and trains the network accordingly, and records the losses in the list 'all_losses'. The variable 'optimal' stores the best-performing activation function permutation. To validate observed patterns, the experiment was repeated with different learning rates and epoch counts.

For visualization, the code plots the loss for each permutation from 'all_losses'. To clearly distinguish results, models using the sigmoid activation function are separated (colored in blue), while those omitting it are displayed in orange. This classification highlights significant differences in performance across the activation functions.

```
for idx, ls in enumerate(all_losses):
    color = 'blue' if 'sigmoid' in activation_permutations[idx] else 'orange'
    plt.plot(ls, color=color, label=f"Config {idx}: {activation_permutations[idx]}")

plt.xlabel("Epochs")
plt.ylabel("MSE Loss")
plt.title(f"Training Loss Over {n_epochs} Epochs for Different Configurations. L-rate = {learning_rate}")

plt.show()

print(f"Best performance: {optimal}")
```

Figure 13: Code for visualization of the commparison results

2.5 Classification using Neural Networks

Classification analysis using neural networks is a powerful technique in supervised machine learning, where the goal is to categorize data into predefined classes based on learned patterns. In classification tasks, the network is trained to recognize specific patterns and assign labels by adjusting weights and biases through backpropagation. This process allows the model to minimize classi-

fication errors by iteratively updating its parameters. With the flexibility to handle complex, non-linear relationships in data, neural networks have become a popular choice for tasks like image recognition, natural language processing, and medical diagnosis, offering high accuracy and adaptability across a range of domains. For this task we used a binary classification algorithm using our manual coded neural network to try to accurately predict on the Wisconsin breast cancer dataset.

2.5.1 The Binary Cross-entropy cost function

The binary cross-entropy loss is well-suited for binary classification because it directly reflects the probability of correct classification. The function penalizes confident but incorrect predictions more heavily than less certain ones, effectively guiding the model to make predictions that reflect the underlying distribution of the classes. This approach optimizes the model's performance by focusing on reducing large errors, leading to faster convergence and improved accuracy.

The cross-entropy cost function can be mathematically expressed as the following

$$\text{Binary Cross-Entropy} = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \qquad (1)$$

We implemented the cost function in code as the following

```python
def binary_cross_entropy_loss(y_true, y_pred, network, lambda_reg):
    epsilon = 1e-15  # To prevent log(0)
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    cross_entropy = -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))

    # Compute L2 regularization term
    l2_term = 0
    num_layers = len(network) // 3
    for layer in range(num_layers):
        W = network[f"W{layer}"]
        l2_term += np.sum(np.square(W))
    l2_term = (lambda_reg / (2 * y_true.shape[0])) * l2_term

    return cross_entropy + l2_term
```

Figure 14: Implementation of Binary cross-entropy cost function

Here we can see that we also introduced an L2 regularization parameter that is calculated with the help of lambda as function input.

2.5.2 Measurement of performance

For performance measurement we will use the accuracy score defined as the following

$$\text{Accuracy} = \frac{1}{m}\sum_{i=1}^{m} 1(\hat{y}_i = y_i) \tag{2}$$

This will alow us to quantify how often we have the correct binary prediction on the Wisconsin breast cancer dataset. For implementation we simply imported the function from the scikitlearn.metrics framework. (referer her)

2.5.3 Code adjustments for classification tasks

As mentioned earlier, this is a binary classification task, meaning, we only receive output as either 1 or 0. To implement this in code we only need a slight adjustment to our manual neural network code. Since we managed to build a code with the ability to customize which activation function we want for which layer, we can now easily change the activation function of our output layer.

When dealing with regression tasks we wanted a linear last activation, meaning, the same type of output as our original data. This is still the case with the classification task. This means we need to get a single output for each input that is either 0 or 1. Or in other words, cancer or not. We can achieve this by feeding our final z vector into a logistic function like the sigmoid, make it 1 if its above 0.5 or 0 if its below 0.5. Below is the implementation of this slight change in our function for training the network.

```
# Compute loss and accuracy for the batch
batch_loss = binary_cross_entropy_loss(y_batch, y_pred, network, lambda_reg)
epoch_loss += batch_loss * len(y_batch)
y_pred_labels = (y_pred >= 0.5).astype(int)
batch_accuracy = accuracy_score(y_batch, y_pred_labels)
epoch_accuracy += batch_accuracy * len(y_batch)
```

Figure 15: Implementation of binary output

In the code above y-pred is the final output of the sigmoid activation function for the output layer. It is then converted into a binary output using boolean logic.

2.6 Logistic Regression

Logistic regression is a popular classification technique that models the probability of a binary outcome by applying a logistic function to a linear combination of the input features. This probabilistic approach enables logistic regression to provide interpretable predictions, mapping any input to a value between 0 and

1, which can be interpreted as the probability of belonging to a specific class.

To implement logistic regression for our classification task, we begin by defining a suitable cost function. The typical choice is the binary cross-entropy cost function, which effectively penalizes incorrect predictions based on their confidence levels. For an input feature set represented by a design matrix X, where each row corresponds to a sample and each column to a feature, our objective is to find the optimal set of weights $\mathbf{w}$ and a bias term b that minimize this cost. The design matrix structure facilitates efficient computation and gradient-based optimization of the parameters.

2.6.1 Implementation of code

We proceed by coding logistic regression using our stochastic gradient descent (SGD) algorithm, with learning rate as a hyperparameter, to iteratively update the weights and biases. Additionally, we incorporate an L2 regularization parameter to control for overfitting, especially in high-dimensional spaces. This regularization term helps balance the model's complexity by penalizing large weights, encouraging the algorithm to find a simpler, more generalizable solution. Here is the code implementation of our logistic regression and functions to compute gradient and code

```python
# Cost function with L2 regularization
def compute_cost(X, y, weights, lambda_reg):
    m = y.shape[0]
    h = sigmoid(X @ weights)
    epsilon = 1e-15  # To prevent log(0)
    h = np.clip(h, epsilon, 1 - epsilon)
    cost = (
        -1 / m
    ) * np.sum(y * np.log(h) + (1 - y) * np.log(1 - h))
    reg_term = (lambda_reg / (2 * m)) * np.sum(np.square(weights[1:]))
    return cost + reg_term

# Gradient computation with L2 regularization
def compute_gradient(X, y, weights, lambda_reg):
    m = y.shape[0]
    h = sigmoid(X @ weights)
    gradient = (1 / m) * (X.T @ (h - y))
    # Apply regularization to weights (exclude bias term)
    gradient[1:] += (lambda_reg / m) * weights[1:]
    return gradient

# Logistic Regression using SGD
def logistic_regression_sgd(X, y, learning_rate=0.01, n_epochs=1000, lambda_reg=0.01, batch_size=32):
    m, n = X.shape
    weights = np.zeros((n, 1))
    losses = []
    accuracies = []

    for epoch in range(n_epochs):
        # Shuffle data at the start of each epoch
        indices = np.random.permutation(m)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        for start in range(0, m, batch_size):
            end = start + batch_size
            X_batch = X_shuffled[start:end]
            y_batch = y_shuffled[start:end]

            # Compute gradient and update weights
            gradient = compute_gradient(X_batch, y_batch, weights, lambda_reg)
            weights -= learning_rate * gradient

        # Compute loss and accuracy for monitoring
        loss = compute_cost(X, y, weights, lambda_reg)
        losses.append(loss)

        # Predict on the training set
        y_pred_train = sigmoid(X @ weights) >= 0.5
        accuracy = accuracy_score(y, y_pred_train)
        accuracies.append(accuracy)

        if epoch % 100 == 0:
            print(
                f"Epoch {epoch}, Loss: {loss:.4f}, Training Accuracy: {accuracy:.4f}"
            )

    return weights, losses, accuracies
```

Figure 16: Implementation of Logistic regression

Here we end with applying our regression results to a sigmoid function, which introduces non-linearity and gives us the 0 to 1 range boundary that we want for the classification tasks.

2.6.2 Scikit-learn comparison

To validate that our logistic regression code is preforming as expected and to test its performance we compared it against scikit-learns [2] built in logistic regression framework. After training the parameters for both versions using stochastic gradient decent we received a final output in the command line terminal as the following

```
Final Test Accuracy (Manual Logistic Regression): 0.9825
Final Test Accuracy (Scikit-Learn Logistic Regression): 0.9386
```

Figure 17: Command line output of the final predictions for manual and scikit-learn logistic regression after training

As we can see, our model performed even better than the scikit-learn version, validating the correction of our code and further solidifying our confidence when analyzing results.

3 Results

3.1 Gradient Decent Algorithms

The synthetic dataset, generated with a quadratic relationship and added Gaussian noise, served as a baseline to evaluate the performance of various optimization algorithms. Each method was assessed using Mean Squared Error (MSE) and R^2 scores. The results indicate that all optimization methods achieved high R^2 values, demonstrating a strong fit to the underlying quadratic pattern in the data.

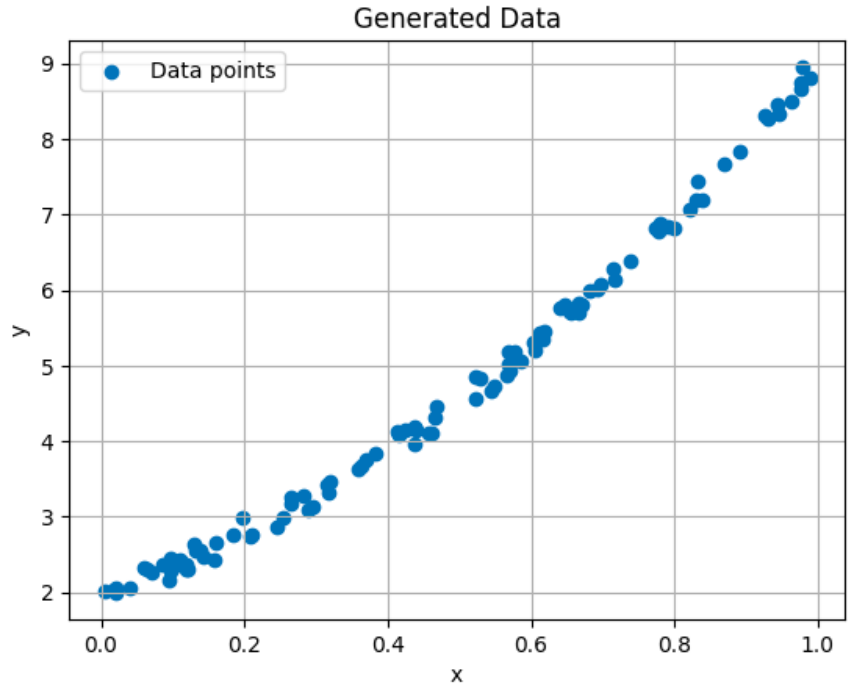


Figure 18: Generated Data

Basic Gradient Descent (GD) achieved an MSE of 0.0098 and an R^2 score of 0.9976, demonstrating effective parameter estimation. Adding momentum to GD slightly improved convergence speed and stability, yielding similar metrics (MSE of 0.0097 and R^2 of 0.9976), indicating that the inclusion of momentum can enhance optimization efficiency without drastically affecting accuracy for this dataset.

Stochastic Gradient Descent (SGD) with mini-batches produced an MSE of 0.0128 and an R^2 score of 0.9968. While SGD allows for more frequent parameter updates, which is advantageous in larger datasets, it resulted in a slightly higher MSE in this context. This demonstrates the trade-off between computational efficiency and precision that SGD offers, which can be particularly useful when working with large datasets or limited resources.

Adaptive learning rate methods, including Adagrad, RMSprop, and Adam, were also tested. Adagrad showed an MSE of 0.0568 and an R^2 of 0.9858, as its learning rate decays over time, potentially slowing convergence on simpler datasets. RMSprop improved on this by stabilizing the learning rate, achieving an MSE of 0.0098 and an R^2 of 0.9975, comparable to basic GD. Adam, which combines momentum with adaptive learning rates, achieved an MSE of 0.0743 and an R^2 of 0.9814. While Adam's performance was slightly lower here, its advantages

in handling more complex data structures make it highly valuable in scenarios involving diverse feature scales and data patterns.

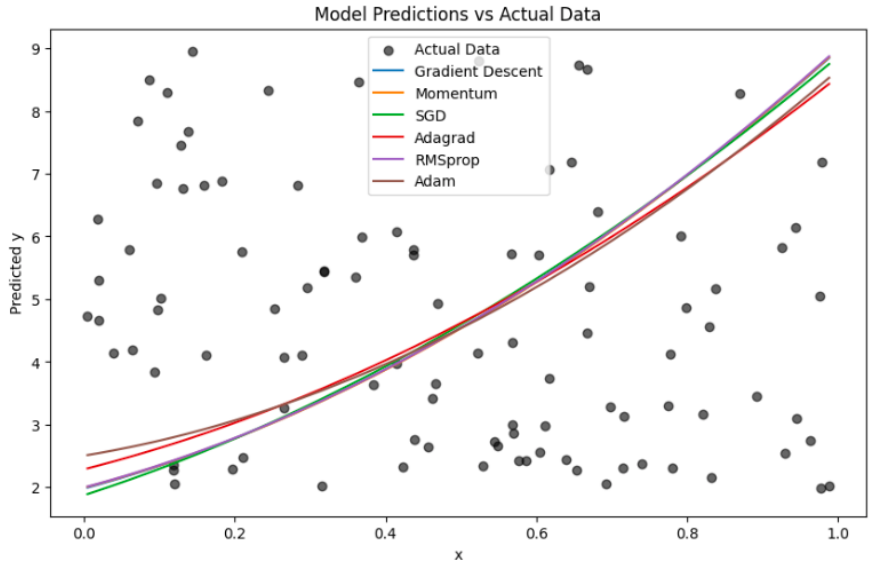


Figure 19: Model Predictions vs Actual Data

Figure illustrates the predicted values from each optimization method compared against the actual data points. Each method successfully captures the quadratic trend in the data, with Gradient Descent, Momentum, and RMSprop aligning most closely with the observed values.

3.2 Regression using neural networks

For the regression task using neural networks we did analysis of the relationship between MSE and R2 vs the learning rate, regularization term lambda, and the number of epochs.

3.2.1 MSE and R2 as a function of learning rate

We did an analysis by studying the MSE and R2 values for our regression as a function of learning rate. The following plots describe our findings

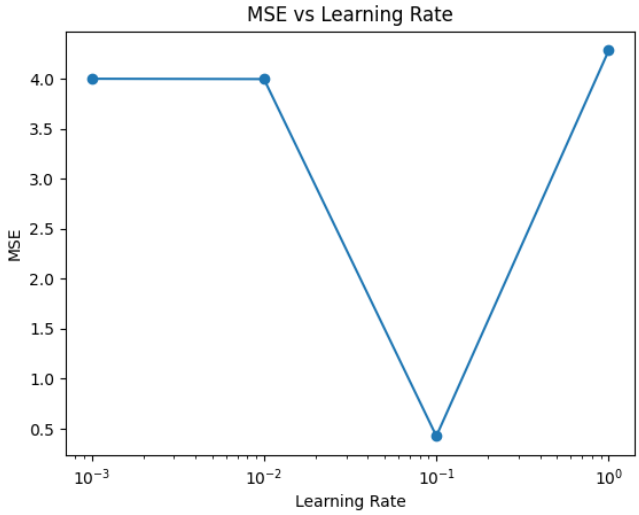


Figure 20: MSE as a function of learning rate

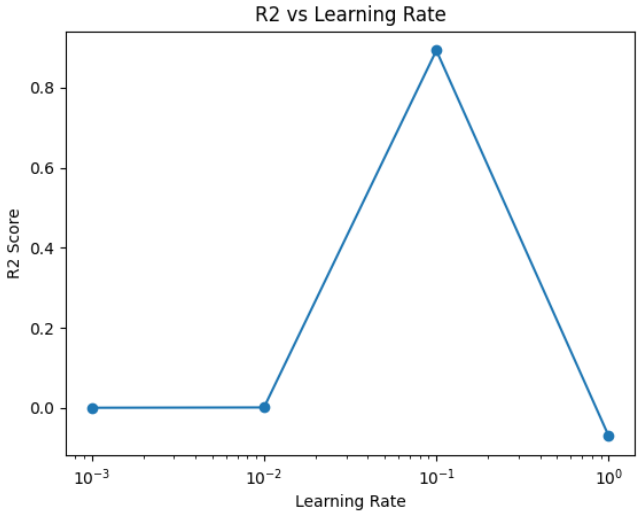


Figure 21: R2 score as a function of learning rate

The leanings we gather from these plots have to be carefully interpreted. As we can see, our model outperformed every other variant when choosing a learning rate of 0.1, but we also observe a rather odd consistency for every other learning rate value in our range. We believe this is an emergent property derived

from the fact that our data, a second-order polynomial, has low complexity and a specific magnitude for the gradients (learning rate) is superior in finding the optimal values.

3.2.2 MSE as a function of number of epochs

Since we are using stochastic gradient decent for training our parameters it is useful to analyze the relationship between the values we evaluate our model with and the number of epochs for SGD. The following plot describe this relationship.

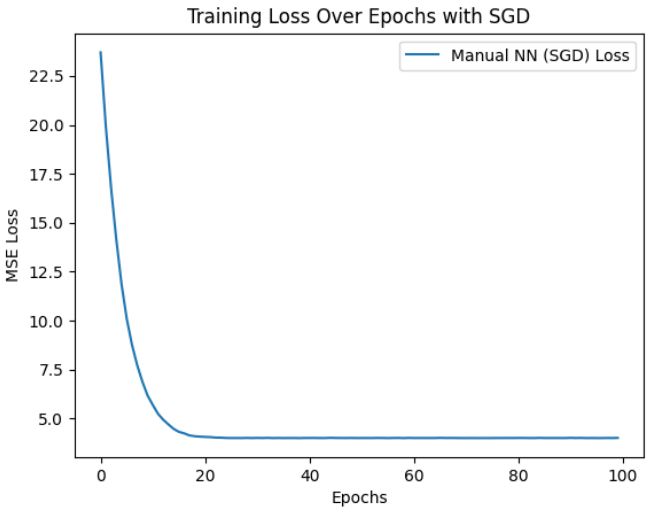


Figure 22: MSE as a function of number of epochs

This plot is describing a development as we expected. When the number of iterations/epochs increase, the more the parameters can improve based on the cost function, leading to lower overall loss scores. Another observation is that it converges quite fast only needing about 20 epochs to essentially find its assumptive global minimum.

If we were to compare with our linear regression form project 1 we would see a similar pattern. First of all, we are not using the same data or similar ways of attaining optimal parameters so a direct comparison is not possible, but we can interpret the number of epochs in our neural network as essentially the number of polynomial degree for our linear regression using a design matrix. The same pattern occurs and they both have an increasing level of computational strain.

3.2.3 MSE and R2 as a function of regularization term λ

Finally we want to analyses how our scores change when we introduce regularization and what changes occur when applying different values for lambda. The following plots shows our results.

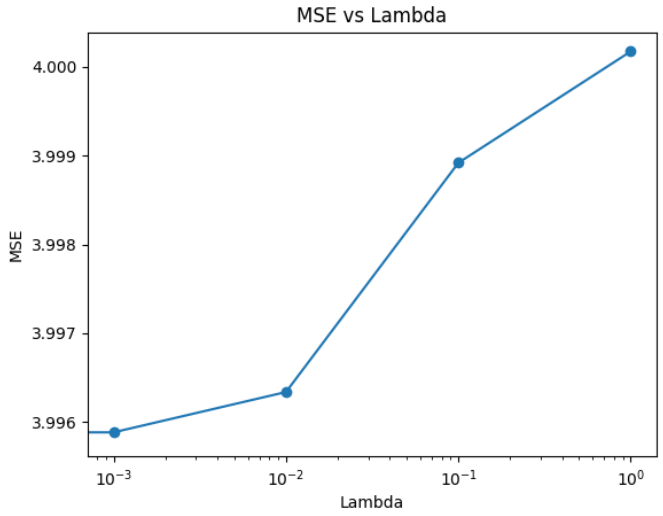


Figure 23: MSE as a function of lambda

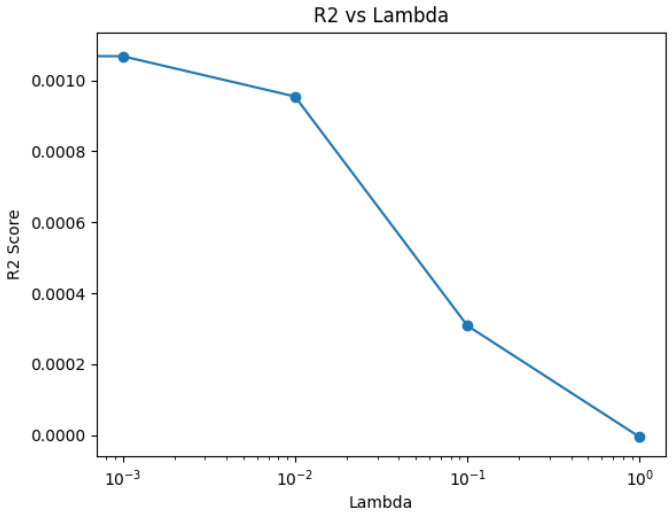
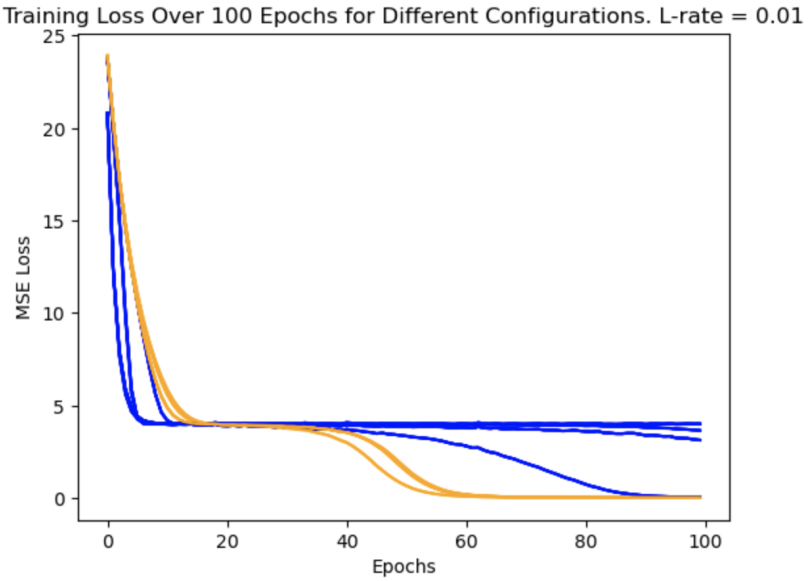


Figure 24: R2 as a function of lambda

As we can tell from these plots, the model performance is negatively correlated with increasing lambda values. This again, can be a property that emerges from the fact that our data is quite simple and there is a low probability of over-fitting. We can easily conclude for this part that regularization does not provide any additional performance gains for the regression task using our neural network.

3.3 Testing activation functions

Testing different activation functions (ReLU, Leaky ReLU, and Sigmoid) on the hidden layers of the neural network highlighted notable limitations when using the Sigmoid activation function. Models that included Sigmoid consistently performed worse than those using only ReLU or Leaky ReLU, a trend that becomes particularly evident when iterating over 600 to 1000 iterations.




```
Best performance: [['relu', 'relu', 'leaky_relu', 'linear'], 0.014125649355756185]
```

Figure 25: MSE for each permutation of activation functions relu, leaky relu and sigmoid

Figure 23 illustrates that models incorporating only ReLU and Leaky ReLU (orange) achieve significantly better performance, indicated by lower Mean Squared Error (MSE). ReLU and Leaky ReLU help maintain stronger gradient flow, allowing the network to learn more effectively. Conversely, Sigmoid activations tend to saturate quickly with larger inputs, resulting in diminishing gradients that hinder learning. The same pattern can be found in runs where you reduce or increase complexity

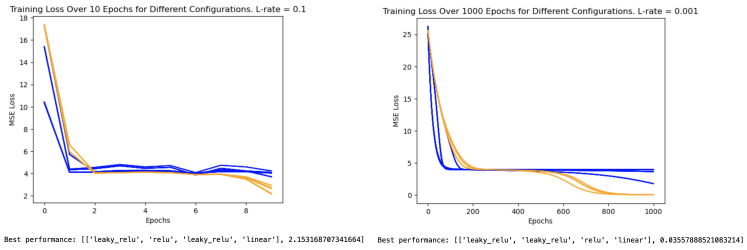
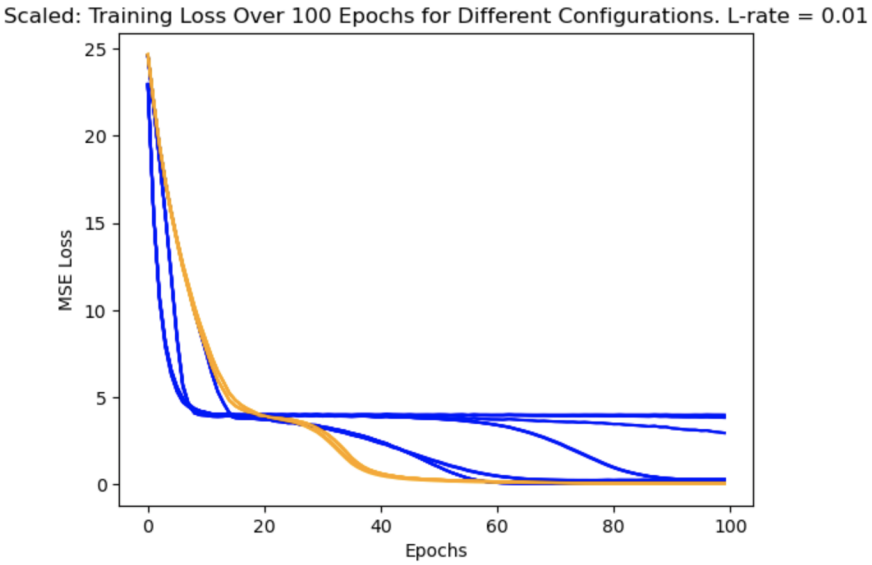


Figure 26: Low and high complexity runs of the model

One factor contributing to the poor performance of Sigmoid may be the unscaled nature of the data. Scaling the data using Scikit-Learn's StandardScaler [2] yields different results when rerunning the model.




```
Best performance: [['sigmoid', 'leaky_relu', 'relu', 'linear'], 0.03726851269069132]
```

Figure 27: Comparison between permmutations of activation functions using scaled data.

After scaling, models with Sigmoid functions show some improvement in MSE scores. However, models without Sigmoid still achieve better MSE scores more consistently. This indicates that for this regression case, omitting the Sigmoid function remains beneficial even after scaling.

In conclusion, ReLU and Leaky ReLU outperform Sigmoid in handling unscaled and scaled data in regression cases, due to their unbounded positive range, which preserves gradient flow and avoids the saturation issues inherent

to Sigmoid. This difference underscores the advantages of ReLU-based activations, which effectively handle unscaled data by nullifying or allowing a slight gradient for negative values (in the case of Leaky ReLU), thus supporting better learning outcomes in this model.

3.4 Classification analysis using neural networks

We now want to analyze how well we can perform binary classification on the Wisconsin breast cancer dataset using our neural network and what the optimal hyper-parameters are that produces the highest accuracy score. We do this step-by-step studying the relationship between the binary cross-entropy loss function and the hyper-parameters we wish to gain an insight about.

3.4.1 Accuracy as a function of learning rate

The learning rate is one of the most critical hyper-parameters of gradient decent and we can get quite different performances based on what rate we chose to use. It is then critical to be careful and study which learning rate gives us the best accuracy score.

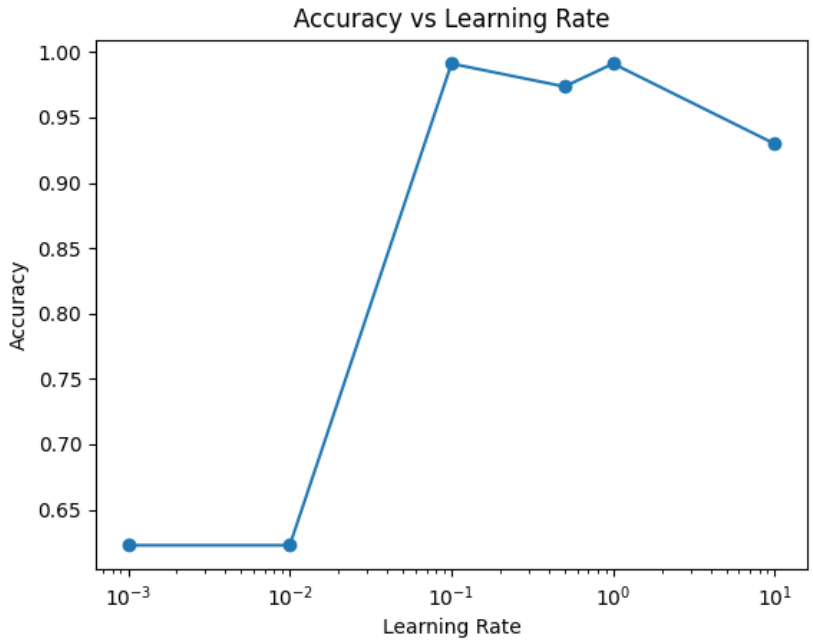


Figure 28: Accuracy as a function of learning rate

The plot illustrates the impact of learning rate on the accuracy of a neural network trained on the Wisconsin Breast Cancer dataset. At very low learning

rates (10^{-3} to 10^{-2}), the model achieves only about 65% accuracy, indicating that the training is too slow to effectively adjust the weights. This results in underfitting, as the model is unable to capture the patterns in the data.

At a moderate learning rate around 10^{-1}, there is a sharp increase in accuracy, reaching nearly 100%. This suggests that this learning rate allows the model to converge effectively, finding an optimal set of weights that generalizes well to the dataset. This rate appears to be close to ideal, balancing learning speed with stability.

However, as the learning rate increases further (10^{0} to 10^{1}), the accuracy begins to drop slightly, indicating some instability. High learning rates can cause the model to overshoot the optimal solution, leading to reduced accuracy.

In summary, a learning rate around 0.1 appears optimal for this task, as it balances convergence speed and stability. Lower learning rates lead to underfitting, while excessively high rates cause instability, reducing model performance.

3.4.2 Loss and Accuracy as a Function of Number of Epochs

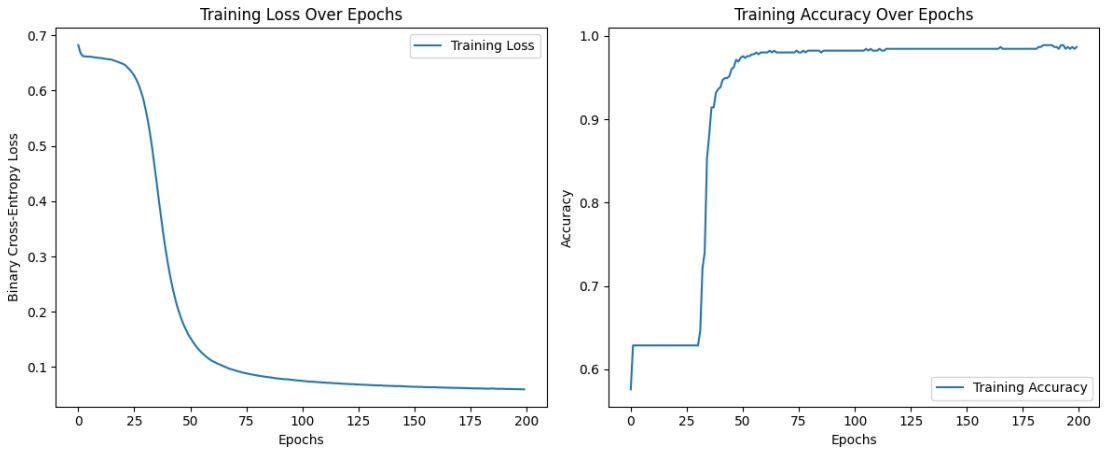


Figure 29: Loss and accuracy as a function of number of epochs

The plots in Figure illustrate the training loss and accuracy of the neural network as functions of the number of epochs. The left plot shows the binary cross-entropy loss over epochs, while the right plot shows the training accuracy over the same period. Both metrics provide insights into the model's learning progression and convergence behavior.

In the loss plot, we observe a sharp decrease in binary cross-entropy loss during

the initial epochs, especially within the first 50 epochs. This rapid reduction indicates that the model quickly learns and adjusts its weights to reduce classification errors. After around 50 epochs, the loss reduction becomes more gradual, and the curve flattens as it approaches a minimum, suggesting that the model is converging to an optimal solution with minimal further improvement.

The accuracy plot complements this observation, showing a sharp increase in accuracy during the early epochs, with a significant rise occurring around the 25-50 epoch range. After reaching close to 100% accuracy, the curve levels off, indicating that the model has achieved near-perfect accuracy on the training set. This plateau suggests that the model is well-fitted to the training data, with limited additional improvement as training continues.

Overall, these plots indicate that the model reaches a state of high performance relatively quickly, stabilizing around 50 epochs. Beyond this point, additional epochs contribute little to performance improvement, implying that 50 epochs might be sufficient for training in this case. The high final accuracy suggests that the model has learned the patterns in the training data effectively.

3.4.3 Accuracy vs Number of Hidden Layers

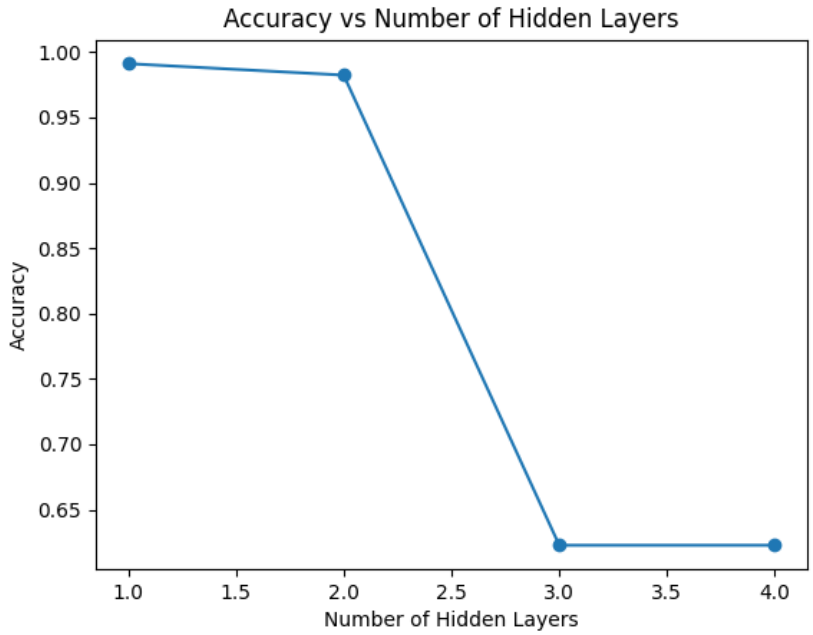


Figure 30: Accuracy as a function of number of hidden layers

This plot shows the impact of the number of hidden layers on model accuracy. With one or two hidden layers, the model achieves near-perfect accuracy, indicating effective learning and convergence. However, as the number of hidden layers increases to three or more, there is a sharp decline in accuracy, dropping to around 65%. This decrease likely reflects overfitting or instability in the model, where excessive depth without sufficient regularization can lead to poor generalization on this dataset. This suggests that one or two hidden layers are optimal for this task.

3.4.4 Accuracy vs Nodes per Layer

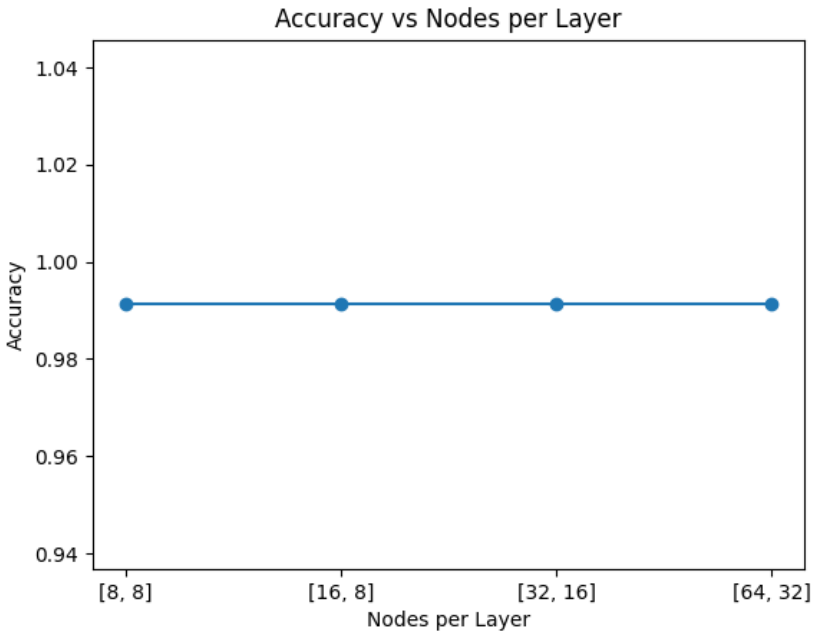


Figure 31: Accuracy as a function of nodes

This illustrates the model's accuracy as a function of the number of nodes per layer. Across different configurations—ranging from small networks (e.g., [8, 8] nodes) to larger ones (e.g., [64, 32] nodes)—the accuracy remains consistently high, close to 100%. This stability implies that the model's performance is not highly sensitive to the specific number of nodes in each layer for this dataset, as long as there is sufficient capacity to capture the patterns in the data.

3.4.5 Accuracy vs Activation Function

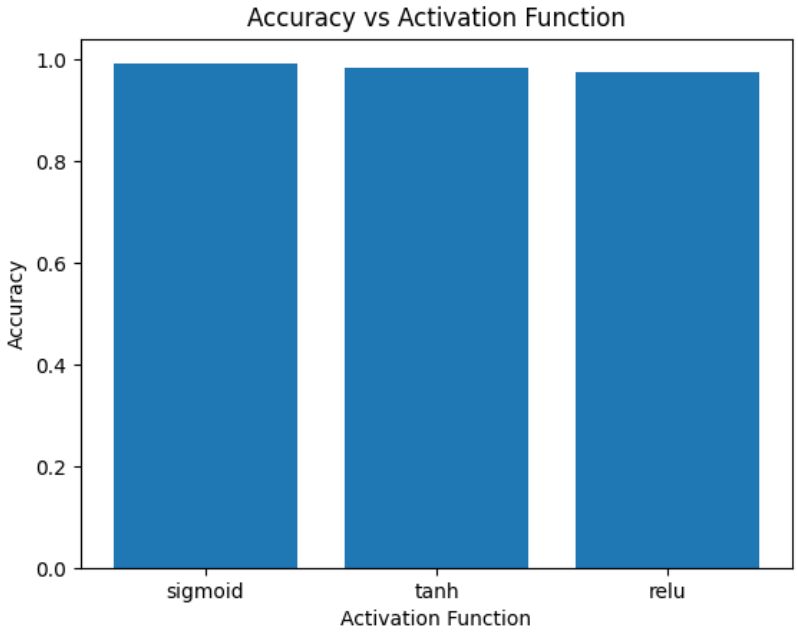


Figure 32: Accuracy scores from each activation function configuration

The plot in Figure compares the accuracy across different activation functions: sigmoid, tanh, and ReLU. The results show that all three activation functions yield similar, high accuracy, close to 100%. This indicates that the choice of activation function has minimal impact on accuracy for this task. However, given ReLU's computational efficiency and its tendency to perform well in deep networks, it could still be a preferred choice despite similar accuracy levels.

3.5 Logistic Regression Analysis

In this section, we analyze the performance of logistic regression, trained using stochastic gradient descent (SGD) with various learning rates, and compare it to the feedforward neural network (FFNN) we developed. The logistic regression model's accuracy and loss are evaluated across different learning rates and as a function of training epochs. Additionally, we include a regularization parameter to control overfitting and enhance generalization. The results obtained are then compared with the FFNN and Scikit-Learn's logistic regression functionality.

3.5.1 Test Accuracy vs Learning Rate

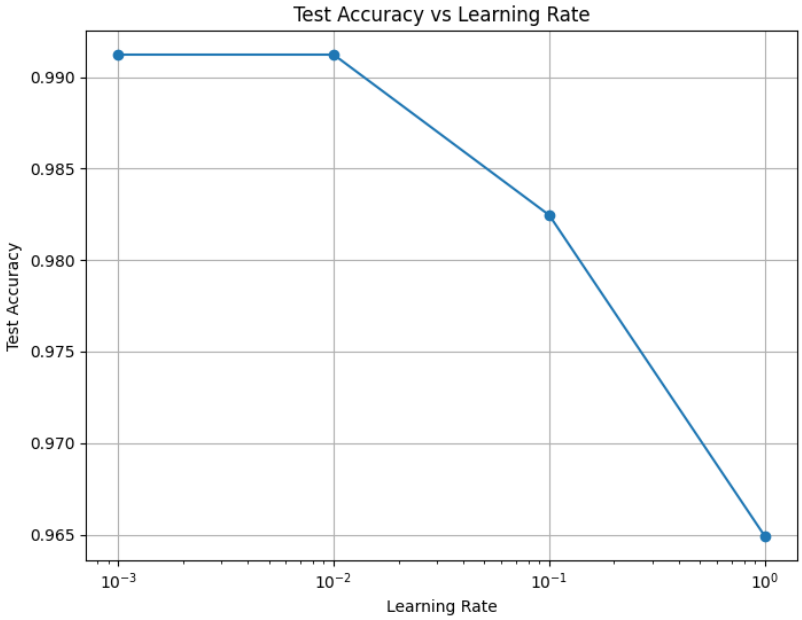


Figure 33: Accuracy as a function of learning rate

This plot illustrates the impact of learning rate on the test accuracy of logistic regression. At low learning rates (10^{-3} and 10^{-2}), the model achieves high accuracy (around 99%), suggesting effective training. However, as the learning rate increases to 10^{-1} and above, the test accuracy declines steadily, with a significant drop at a learning rate of 1. This suggests that high learning rates cause the model to overshoot optimal solutions, leading to instability and poor generalization. A learning rate of 10^{-3} or 10^{-2} appears ideal for this dataset, balancing learning speed and stability.

3.5.2 Training Loss and Accuracy over Epochs (Learning Rate = 1)

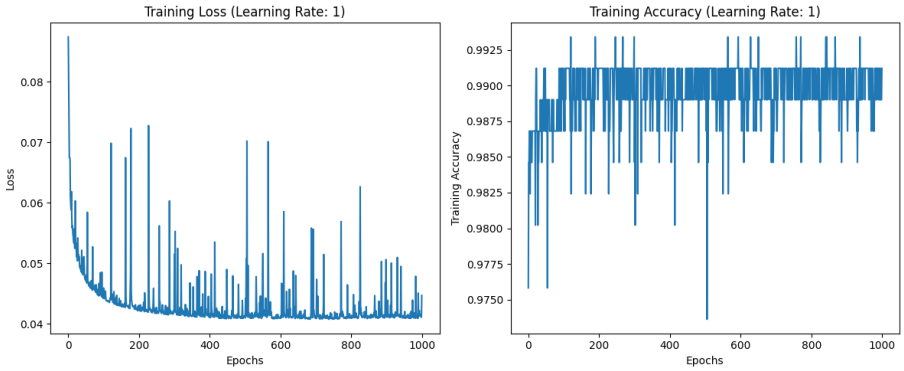


Figure 34: Accuracy vs epoch (learning rate = 1

Here we examine the training loss and accuracy over epochs with a learning rate of 1. The high volatility in both loss and accuracy indicates that the model is struggling to converge, likely due to the large learning rate. The loss fluctuates significantly without stabilizing, and the accuracy shows erratic behavior, suggesting that the model fails to settle on an optimal solution. This behavior highlights that a learning rate of 1 is too high, leading to unstable training and poor performance.

3.5.3 Training Loss and Accuracy over Epochs (Learning Rate = 0.1)

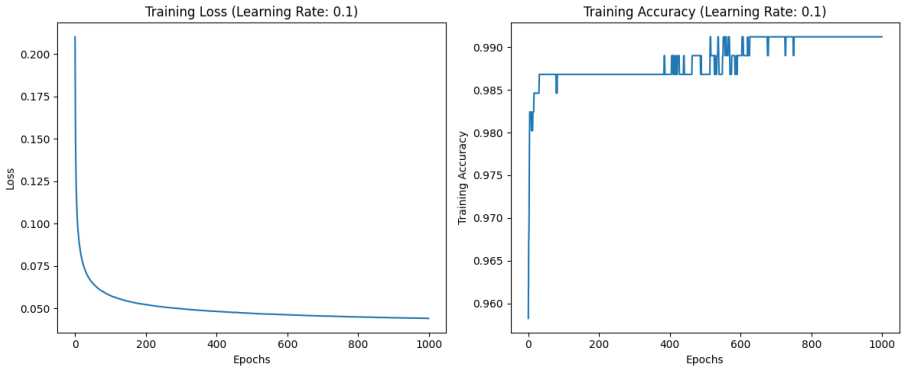


Figure 35: Accuracy vs epoch (learning rate = 0.1

This plot shows the training loss and accuracy for a learning rate of 0.1. The loss decreases steadily and stabilizes after around 200 epochs, indicating successful convergence. The accuracy also improves and remains stable after an initial rise, reaching nearly 99%. This learning rate offers a good balance between convergence speed and stability, resulting in high performance without the fluctuations observed at higher rates.

3.5.4 Training Loss and Accuracy over Epochs (Learning Rate = 0.01)

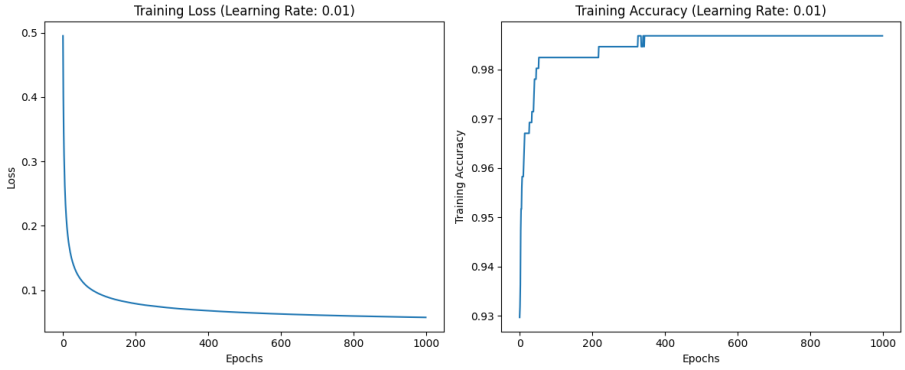


Figure 36: Accuracy vs epoch (learning rate = 0.01

Here we observe the model's performance with a learning rate of 0.01. The loss declines smoothly and stabilizes, although convergence is slower than with a rate of 0.1. The accuracy rises gradually and stabilizes around 98%, reflecting consistent and steady learning. This lower learning rate ensures stable training, but the convergence is slower, making it less efficient for rapid training compared to a learning rate of 0.1.

3.5.5 Training Loss and Accuracy over Epochs (Learning Rate = 0.001)

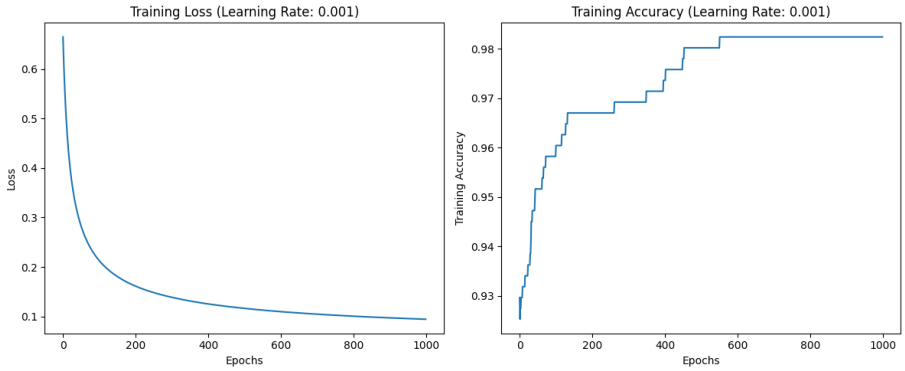


Figure 37: Accuracy vs epoch (learning rate = 1

This illustrates the results with a learning rate of 0.001. The loss decreases very gradually, reflecting a slow convergence. The accuracy also increases slowly, eventually stabilizing around 97%. While this learning rate provides the most stable convergence, it is too slow for practical purposes, as it requires significantly more epochs to reach optimal performance. This emphasizes the trade-off between stability and training speed at very low learning rates.

3.6 Critical evaluation of the various algorithms

- Relu and leaky Relu perform better in regression cases - validated by different runs with different epoch and learning rates
- Scaling improves sigmoid performance, however it is not as consistens as Relu and leaky relu
- Relu and leaky relu maintains stronger gradient flow, leading to more efficient learning
- Sigmoid activations saturates quickly with large inputs, resulting in diminishing gradients that hinder learning.

4 Critical Evaluation of the Various Algorithms

In this project, we implemented and analyzed several machine learning algorithms for both regression and classification tasks. The algorithms included various optimization techniques for linear regression models, a custom feed-forward neural network (FFNN) for regression and classification, and logistic regression using stochastic gradient descent (SGD). Below, we summarize and critically evaluate these algorithms, discussing their pros and cons, and determine which algorithms performed best for each task.

4.0.1 Regression Algorithms

Gradient Descent and Its Variants We began by implementing basic Gradient Descent (GD) and enhanced it with momentum, mini-batch Stochastic Gradient Descent (SGD), and adaptive learning rate methods such as Adagrad, RMSprop, and Adam. The primary goal was to optimize the parameters of linear regression models (OLS, Ridge, and Lasso regression) without relying on matrix inversion.

- **Basic Gradient Descent (GD):** Provided a solid baseline with effective convergence to the optimal solution. It was simple to implement but required careful tuning of the learning rate.
- **GD with Momentum:** Improved convergence speed and stability by incorporating past gradients, reducing the risk of getting stuck in shallow minima. However, it added an extra hyperparameter (momentum term) that needed tuning.
- **Stochastic Gradient Descent (SGD):** Enhanced computational efficiency, especially beneficial for larger datasets. The randomness introduced helped in escaping local minima but led to more erratic convergence paths, potentially requiring more epochs to converge.
- **Adaptive Learning Rate Methods (Adagrad, RMSprop, Adam):**
  - **Adagrad:** Adapted the learning rate based on the accumulation of past squared gradients, which was advantageous for sparse data. However, its learning rate could become excessively small over time, slowing convergence on non-sparse data.
  - **RMSprop:** Addressed Adagrad's diminishing learning rate by introducing a decay factor, leading to more stable and faster convergence.
  - **Adam:** Combined momentum and RMSprop, providing the benefits of both. It worked well out-of-the-box for many problems but sometimes led to worse generalization compared to SGD with momentum.

Feed-Forward Neural Network for Regression The custom FFNN was implemented with flexibility in the number of hidden layers and nodes, and different activation functions were tested (Sigmoid, ReLU, Leaky ReLU).

- **Pros:**
  - **Flexibility:** Capable of modeling complex, non-linear relationships in data.
  - **Performance:** Achieved high accuracy when appropriately configured.
  - **Activation Functions:** ReLU and Leaky ReLU performed better than Sigmoid in hidden layers, preventing vanishing gradient problems and promoting faster convergence.

- **Cons:**
  - **Computational Cost:** More resource-intensive than linear models due to increased complexity and number of parameters.
  - **Hyperparameter Tuning:** Requires careful tuning of learning rates, regularization parameters, number of layers, and nodes to prevent overfitting or underfitting.
  - **Training Time:** Longer training times compared to linear models, especially with larger architectures.

Comparison and Best Algorithm for Regression For the regression task, while the FFNN demonstrated strong performance and flexibility, the traditional linear regression models optimized with GD variants (especially GD with momentum and RMSprop) provided comparable accuracy with significantly less computational overhead. Given the simplicity of the dataset (a quadratic function with noise), the linear models were sufficient and more efficient.

4.0.2 Classification Algorithms

Feed-Forward Neural Network for Classification The FFNN was adapted for classification by changing the output layer's activation function to Sigmoid and using the binary cross-entropy loss function.

- **Pros:**
  - **High Accuracy:** Achieved near-perfect accuracy on the Wisconsin Breast Cancer dataset.
  - **Flexibility:** Capable of capturing complex patterns and interactions between features.
  - **Robustness:** Performed well across various configurations of learning rates, epochs, and activation functions.
- **Cons:**
  - **Overfitting Risk:** With increased depth or too many nodes, the network could overfit, especially evident when using more than two hidden layers.
  - **Computational Cost:** More computationally intensive than logistic regression.
  - **Hyperparameter Sensitivity:** Performance could degrade with suboptimal hyperparameters (too high learning rates or inappropriate activation functions).

Logistic Regression with SGD Implemented logistic regression using SGD and compared it with Scikit-Learn's implementation.

- **Pros:**
  - **Simplicity:** Easy to implement and interpret.
  - **Efficiency:** Faster training times due to fewer parameters.
  - **Performance:** Achieved high accuracy with proper learning rate selection.
- **Cons:**
  - **Limited Flexibility:** Assumes a linear decision boundary, which may not capture complex patterns.
  - **Hyperparameter Sensitivity:** Performance heavily dependent on learning rate and regularization parameter.
  - **Potential Underfitting:** May underfit if the true relationship between features and the target is non-linear.

Comparison and Best Algorithm for Classification While logistic regression performed well and offered simplicity and efficiency, the FFNN consistently achieved higher accuracy and demonstrated robustness to different configurations. The neural network's ability to model non-linear relationships made it more suitable for capturing the complexities in the dataset.

4.0.3 General Observations

- **Activation Functions:** ReLU and Leaky ReLU outperformed Sigmoid in hidden layers for both regression and classification, primarily due to their ability to mitigate the vanishing gradient problem and maintain stronger gradient flow.
- **Learning Rates:** The choice of learning rate significantly impacted convergence and performance across all algorithms. Learning rates that were too low led to slow convergence, while rates that were too high caused instability.
- **Regularization:** Regularization techniques like L2 regularization did not significantly improve performance in the regression task, likely due to the simplicity of the dataset and the low risk of overfitting.
- **Model Complexity:** Increasing the number of hidden layers beyond two in the neural network did not enhance performance and often led to decreased accuracy, highlighting the importance of model complexity matching the dataset's complexity.

4.0.4 Final Recommendations

- **Regression Tasks:** For datasets with simple, linear or low-degree polynomial relationships, traditional linear regression models optimized with efficient gradient-based methods are preferred due to their simplicity and computational efficiency.
- **Classification Tasks:** For datasets where complex, non-linear patterns are present, neural networks offer superior performance despite higher computational costs. Careful tuning of hyperparameters and network architecture is essential to maximize accuracy and prevent overfitting.
- **Algorithm Selection:** The choice of algorithm should consider the complexity of the data, the computational resources available, and the importance of model interpretability versus predictive performance.

Overall, this project demonstrated that while advanced algorithms like neural networks have the potential for higher performance, especially in classification tasks with complex data, simpler models can be more appropriate and efficient for certain regression problems.

5 Conclusion

In this project, we implemented and evaluated several machine learning algorithms for regression and classification tasks, focusing on understanding their strengths and limitations through practical application and critical analysis. For regression problems involving simple, low-degree polynomial relationships, traditional linear regression models optimized with gradient descent variants, particularly GD with momentum and RMSprop proved to be efficient and accurate. These methods required less computational power and fewer hyperparameter adjustments compared to neural networks, making them suitable for problems with linear or mildly non-linear patterns.

In contrast, for classification tasks involving complex or non-linear patterns, the custom feed-forward neural network demonstrated superior performance over logistic regression. The FFNN's ability to model intricate relationships and interactions between features allowed it to achieve higher accuracy on the Wisconsin Breast Cancer dataset. However, this came with increased computational cost and the necessity for careful hyperparameter tuning to prevent overfitting, highlighting a trade-off between performance and resource utilization.

Future work could involve applying these algorithms to more complex datasets with higher dimensionality and non-linear relationships to further assess their scalability and robustness. Exploring advanced neural network architectures, such as convolutional neural networks (CNNs) for image data or recurrent neural networks (RNNs) for sequential data, could provide deeper insights into

their capabilities and performance improvements. Additionally, implementing techniques like batch normalization, dropout, or learning rate schedulers may enhance the training efficiency and generalization ability of neural networks.

Overall, this project underscores the importance of selecting appropriate algorithms based on the specific characteristics of the data and the problem domain. Understanding the pros and cons of each method enables practitioners to make informed decisions, balancing factors such as accuracy, computational efficiency, and model complexity. By critically evaluating these algorithms, we contribute to the broader effort of optimizing machine learning techniques for varied applications.

Github: https://github.com/TheodorJaarvik/Fys_stk3155_Project2

References

[1] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[2] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Vincent Dubourg, Jake Vanderplas, Ana Passos, Rudolf Weiss, and Tom Keepers. Scikit-learn: Machine learning in python, 2011.