Ben-Gurion University of the Negev

Mini Project

202.1.4721

Avraham Ron

Deep Learning

Publish date: November 6th, 2024
Submission date: March 16th, 2025

Contents

0 Introduction

This project focuses on the design and implementation of neural network models, specifically fully connected neural networks and Residual Networks (ResNet).

The project's core revolves around several key classes:

NeuralNetwork: Implements a multi-layer neural network for classification, managing forward and backward propagation across multiple NNLayer instances. It supports training using gradient-based optimization and evaluation on test data.

ResNetLayer (Residual Layer): Implements a residual block $(x + W_2 * activation(W_1 x + b))$. These skip connections help mitigate vanishing gradient issues, improving gradient flow in deeper networks.

ResNet: A deep Residual Network (ResNet) utilizing ResNetLayer and NNLayer to build a more stable and trainable architecture. Supports forward propagation, backpropagation, and parameter updates during training.

Activation: Defines activation functions (ReLU, Tanh, Softmax) and their derivatives for backpropagation. Applies activations to neuron outputs and computes their gradients during training.

Loss: Implements cross-entropy loss for classification and least squares loss for regression. Computes gradients of the loss function, guiding the optimizer in minimizing prediction errors.

Optimizer: Implements Stochastic Gradient Descent (SGD) for updating model weights. Supports mini-batch training, tracks training/validation performance, and provides convergence analysis.

Data: Loads and preprocesses datasets from '.mat' files, handling training-test splits, one-hot encoding of labels, and initialization of model weights and biases.

1 Part I: The Classifier and Optimizer

1.1 The Classifier

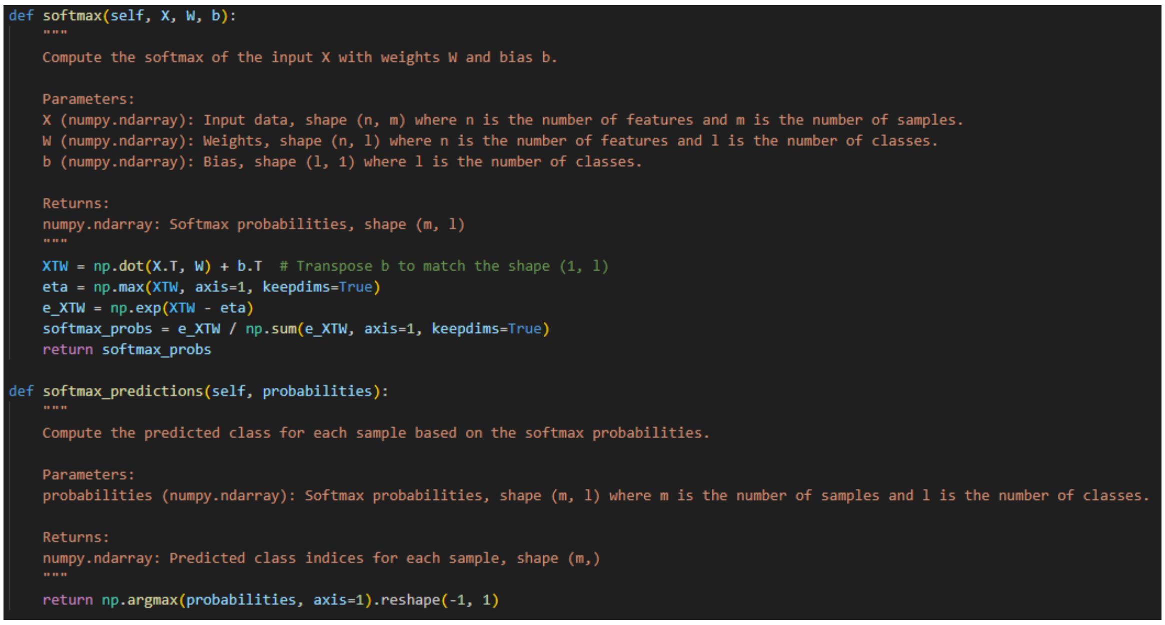
Overview of the Classifier

The classifier implemented in this project utilizes the softmax function, which is widely used for multi-class classification. This function transforms the network's raw output values (logits) into probabilities, enabling the assignment of input samples to specific categories. Each output neuron corresponds to a class, and the softmax function ensures that the sum of probabilities for all classes equals one.

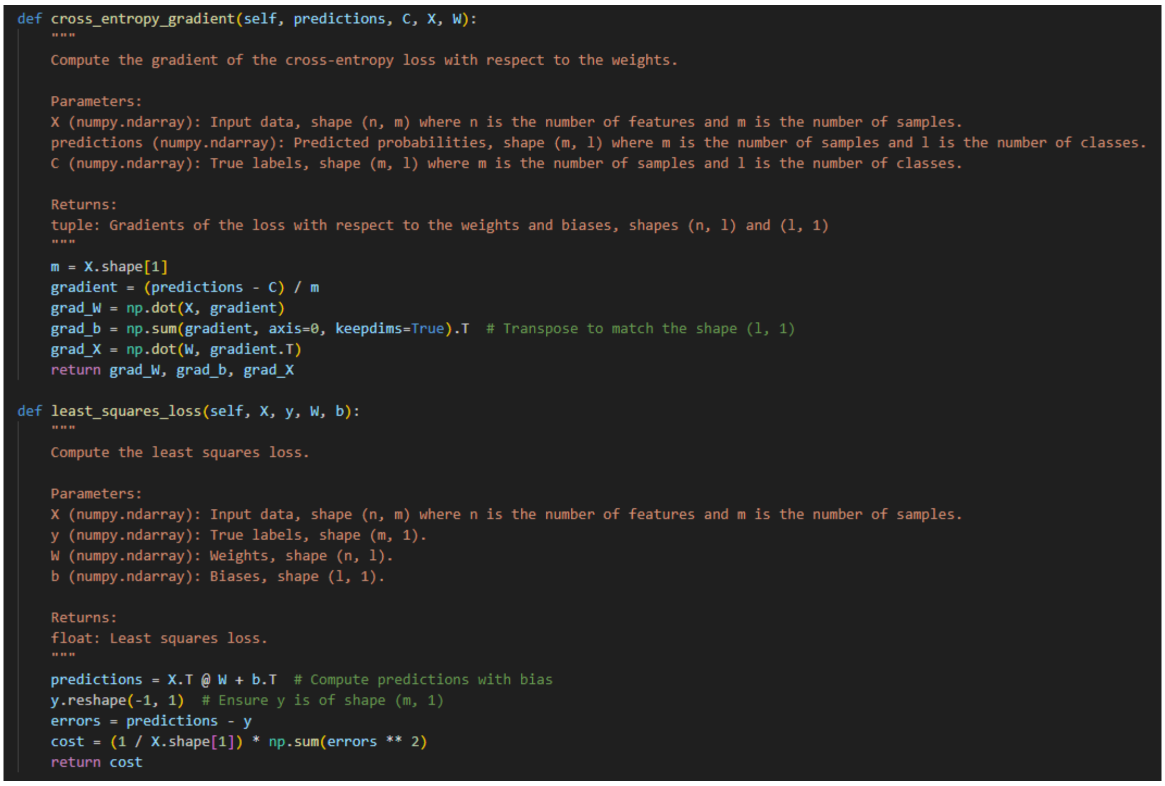
Loss Function and Gradient Computation

To train the classifier, the model employs the cross-entropy loss function, which measures how well the predicted class probabilities align with the actual labels. Cross-entropy is particularly effective for classification because it heavily penalizes incorrect predictions, encouraging the model to improve. The loss is computed based on the predicted probability distribution and the actual one-hot encoded labels.

During backpropagation, gradients of the loss with respect to weights, biases, and input activations are calculated to update the parameters of the model. The derivative of the loss function helps in adjusting the network's weights, guiding it towards making more accurate predictions. These gradients are essential for performing efficient updates using optimization techniques like Stochastic Gradient Descent (SGD).



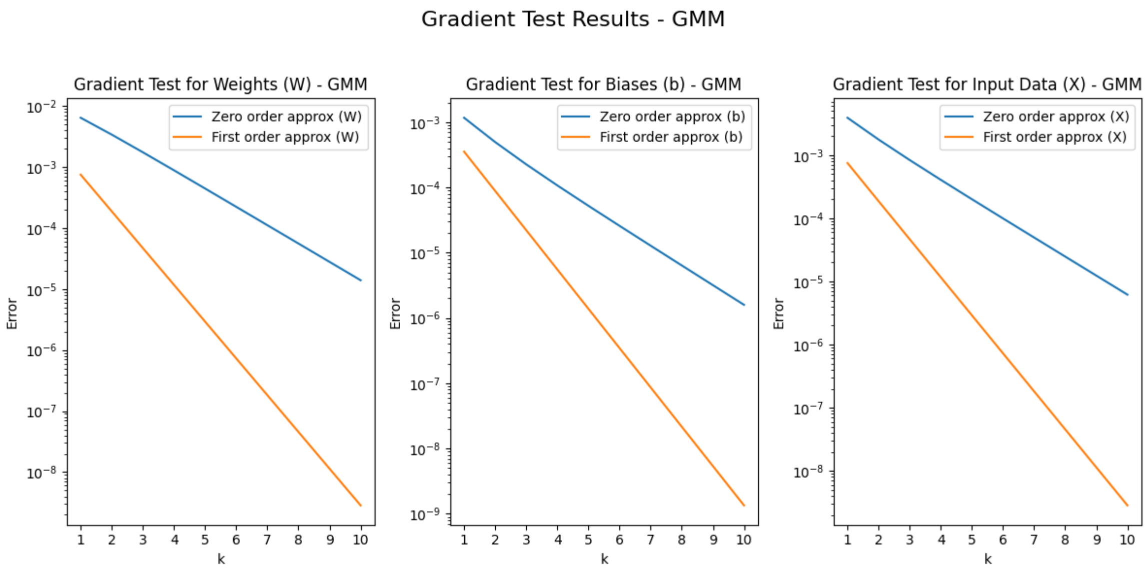
```python
def softmax(self, X, W, b):
    """
    Compute the softmax of the input X with weights W and bias b.

    Parameters:
    X (numpy.ndarray): Input data, shape (n, m) where n is the number of features and m is the number of samples.
    W (numpy.ndarray): Weights, shape (n, l) where n is the number of features and l is the number of classes.
    b (numpy.ndarray): Bias, shape (l, 1) where l is the number of classes.

    Returns:
    numpy.ndarray: Softmax probabilities, shape (m, l)
    """
    XTW = np.dot(X.T, W) + b.T  # Transpose b to match the shape (1, l)
    eta = np.max(XTW, axis=1, keepdims=True)
    e_XTW = np.exp(XTW - eta)
    softmax_probs = e_XTW / np.sum(e_XTW, axis=1, keepdims=True)
    return softmax_probs

def softmax_predictions(self, probabilities):
    """
    Compute the predicted class for each sample based on the softmax probabilities.

    Parameters:
    probabilities (numpy.ndarray): Softmax probabilities, shape (m, l) where m is the number of samples and l is the number of classes.

    Returns:
    numpy.ndarray: Predicted class indices for each sample, shape (m,)
    """
    return np.argmax(probabilities, axis=1).reshape(-1, 1)
```



```
def cross_entropy_gradient(self, predictions, C, X, W):
    """
    Compute the gradient of the cross-entropy loss with respect to the weights.

    Parameters:
    X (numpy.ndarray): Input data, shape (n, m) where n is the number of features and m is the number of samples.
    predictions (numpy.ndarray): Predicted probabilities, shape (m, l) where m is the number of samples and l is the number of classes.
    C (numpy.ndarray): True labels, shape (m, l) where m is the number of samples and l is the number of classes.

    Returns:
    tuple: Gradients of the loss with respect to the weights and biases, shapes (n, l) and (l, 1)
    """
    m = X.shape[1]
    gradient = (predictions - C) / m
    grad_W = np.dot(X, gradient)
    grad_b = np.sum(gradient, axis=0, keepdims=True).T  # Transpose to match the shape (l, 1)
    grad_X = np.dot(W, gradient.T)
    return grad_W, grad_b, grad_X

def least_squares_loss(self, X, y, W, b):
    """
    Compute the least squares loss.

    Parameters:
    X (numpy.ndarray): Input data, shape (n, m) where n is the number of features and m is the number of samples.
    y (numpy.ndarray): True labels, shape (m, 1).
    W (numpy.ndarray): Weights, shape (n, 1).
    b (numpy.ndarray): Biases, shape (1, 1).

    Returns:
    float: Least squares loss.
    """
    predictions = X.T @ W + b.T  # Compute predictions with bias
    y.reshape(-1, 1)  # Ensure y is of shape (m, 1)
    errors = predictions - y
    cost = (1 / X.shape[1]) * np.sum(errors ** 2)
    return cost
```

Gradient Test

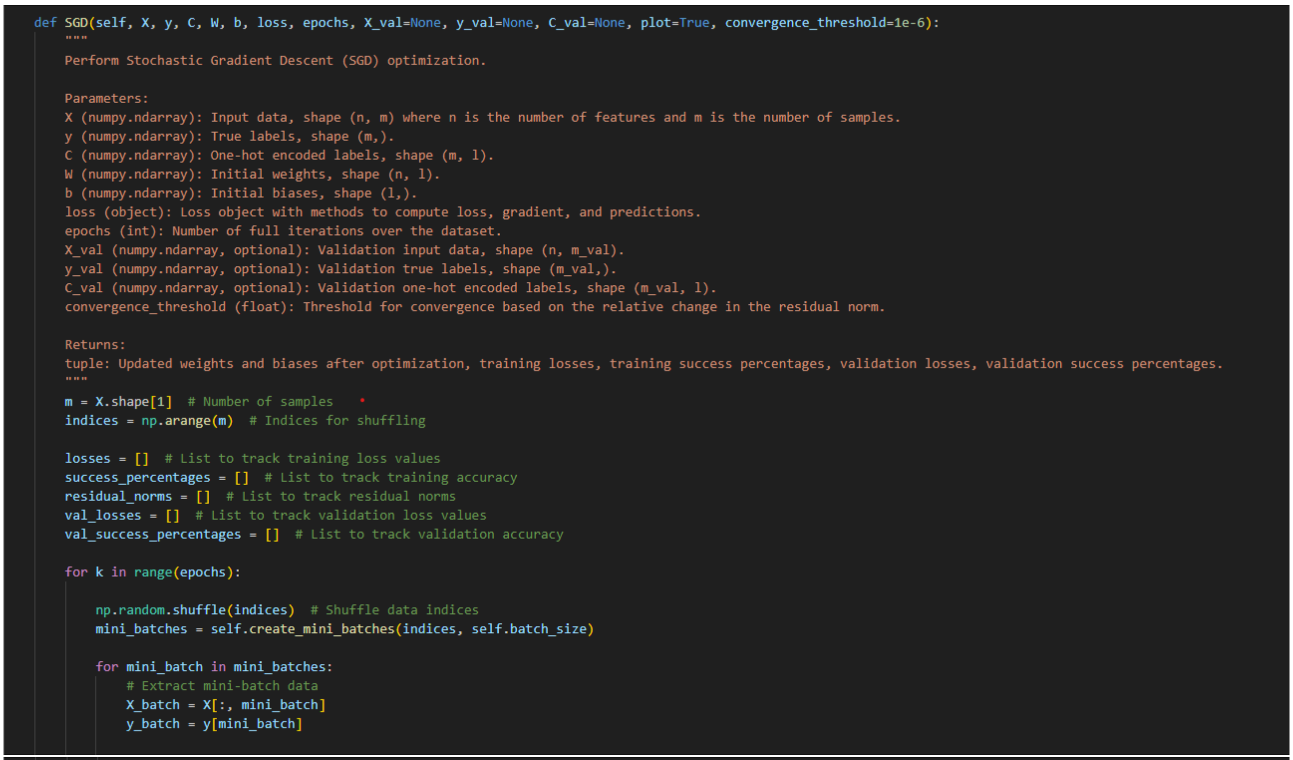
An important step in ensuring the correctness of the model is verifying the computed gradients. The gradient test is conducted by comparing analytically derived gradients with numerical approximations obtained through small perturbations in the network's parameters. If the two sets of gradients are closely aligned, it confirms that the computed gradients are correct.



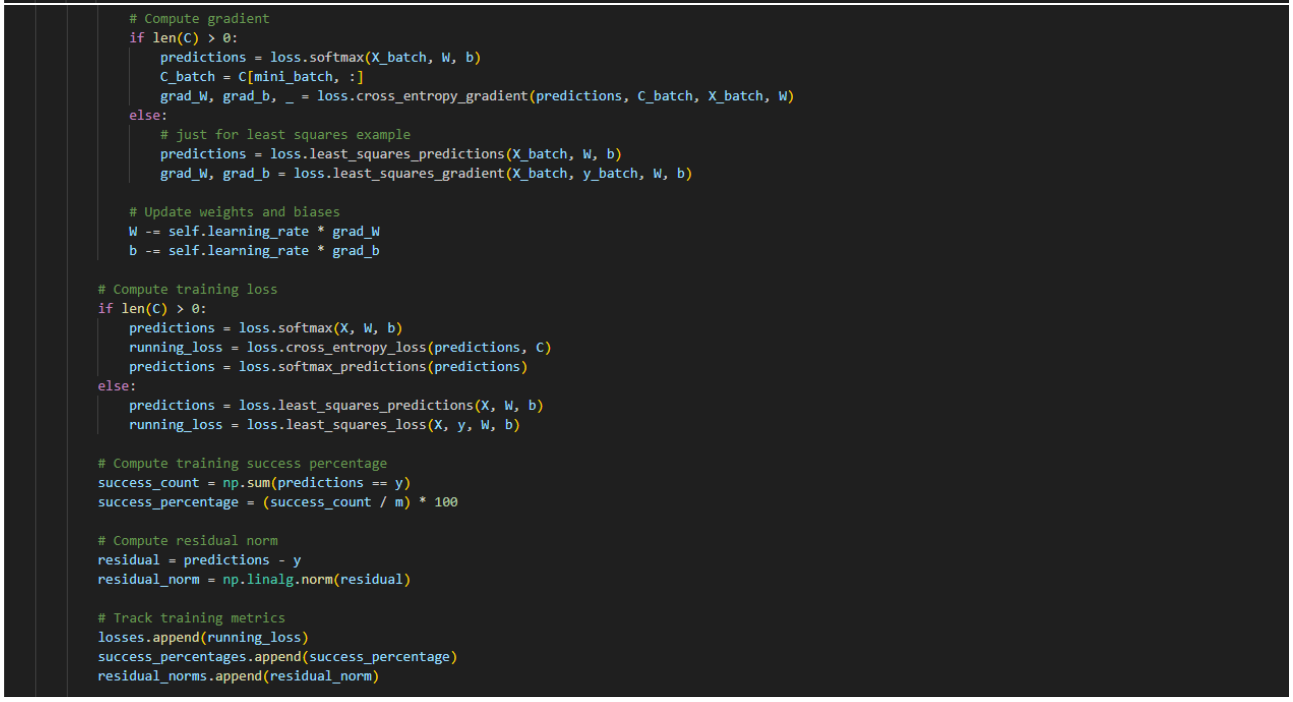
1.2 The Optimizer

Overview of the Optimizer

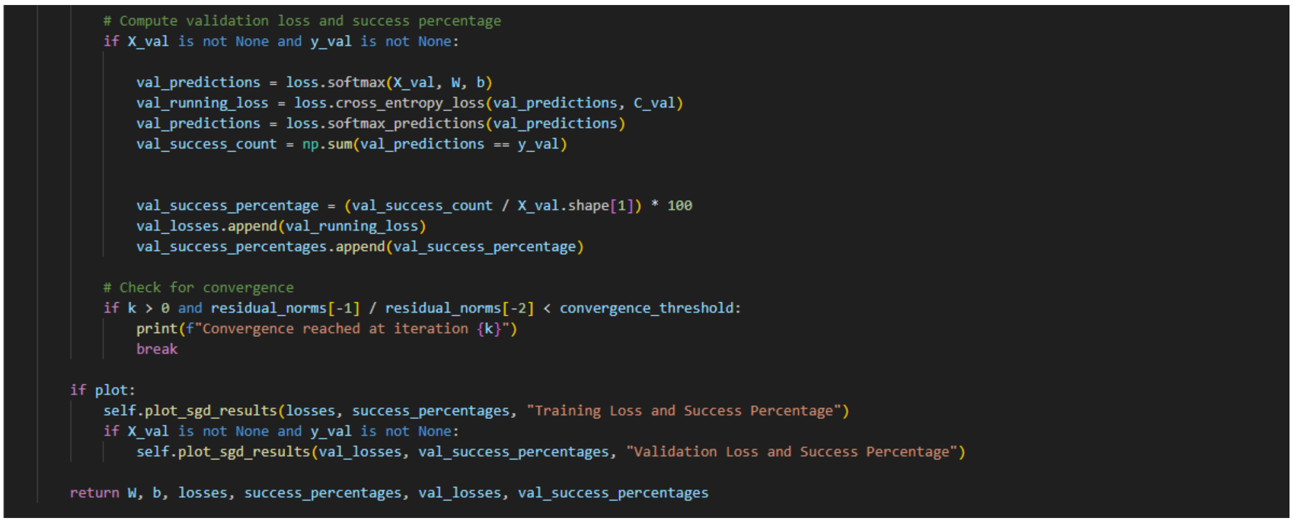
The optimizer used in this project is based on the Stochastic Gradient Descent (SGD) algorithm, which is a widely employed optimization method for neural network training. SGD updates the model parameters iteratively by computing the gradients of the loss function and adjusting the weights and biases in the direction that minimizes the error. This process is controlled by a learning rate, which determines the step size of the updates, and a batch size, which dictates how many samples are processed at each step.



```python
def SGD(self, X, y, C, W, b, loss, epochs, X_val=None, y_val=None, C_val=None, plot=True, convergence_threshold=1e-6):
    """
    Perform Stochastic Gradient Descent (SGD) optimization.

    Parameters:
    X (numpy.ndarray): Input data, shape (n, m) where n is the number of features and m is the number of samples.
    y (numpy.ndarray): True labels, shape (m,).
    C (numpy.ndarray): One-hot encoded labels, shape (m, l).
    W (numpy.ndarray): Initial weights, shape (n, l).
    b (numpy.ndarray): Initial biases, shape (l,).
    loss (object): Loss object with methods to compute loss, gradient, and predictions.
    epochs (int): Number of full iterations over the dataset.
    X_val (numpy.ndarray, optional): Validation input data, shape (n, m_val).
    y_val (numpy.ndarray, optional): Validation true labels, shape (m_val,).
    C_val (numpy.ndarray, optional): Validation one-hot encoded labels, shape (m_val, l).
    convergence_threshold (float): Threshold for convergence based on the relative change in the residual norm.

    Returns:
    tuple: Updated weights and biases after optimization, training losses, training success percentages, validation losses, validation success percentages.
    """
    m = X.shape[1]  # Number of samples
    indices = np.arange(m)  # Indices for shuffling

    losses = []  # List to track training loss values
    success_percentages = []  # List to track training accuracy
    residual_norms = []  # List to track residual norms
    val_losses = []  # List to track validation loss values
    val_success_percentages = []  # List to track validation accuracy

    for k in range(epochs):

        np.random.shuffle(indices)  # Shuffle data indices
        mini_batches = self.create_mini_batches(indices, self.batch_size)

        for mini_batch in mini_batches:
            # Extract mini-batch data
            X_batch = X[:, mini_batch]
            y_batch = y[mini_batch]
```



```python
            # Compute gradient
            if len(C) > 0:
                predictions = loss.softmax(X_batch, W, b)
                C_batch = C[mini_batch, :]
                grad_W, grad_b, _ = loss.cross_entropy_gradient(predictions, C_batch, X_batch, W)
            else:
                # just for least squares example
                predictions = loss.least_squares_predictions(X_batch, W, b)
                grad_W, grad_b = loss.least_squares_gradient(X_batch, y_batch, W, b)

            # Update weights and biases
            W -= self.learning_rate * grad_W
            b -= self.learning_rate * grad_b

        # Compute training loss
        if len(C) > 0:
            predictions = loss.softmax(X, W, b)
            running_loss = loss.cross_entropy_loss(predictions, C)
            predictions = loss.softmax_predictions(predictions)
        else:
            predictions = loss.least_squares_predictions(X, W, b)
            running_loss = loss.least_squares_loss(X, y, W, b)

        # Compute training success percentage
        success_count = np.sum(predictions == y)
        success_percentage = (success_count / m) * 100

        # Compute residual norm
        residual = predictions - y
        residual_norm = np.linalg.norm(residual)

        # Track training metrics
        losses.append(running_loss)
        success_percentages.append(success_percentage)
        residual_norms.append(residual_norm)
```

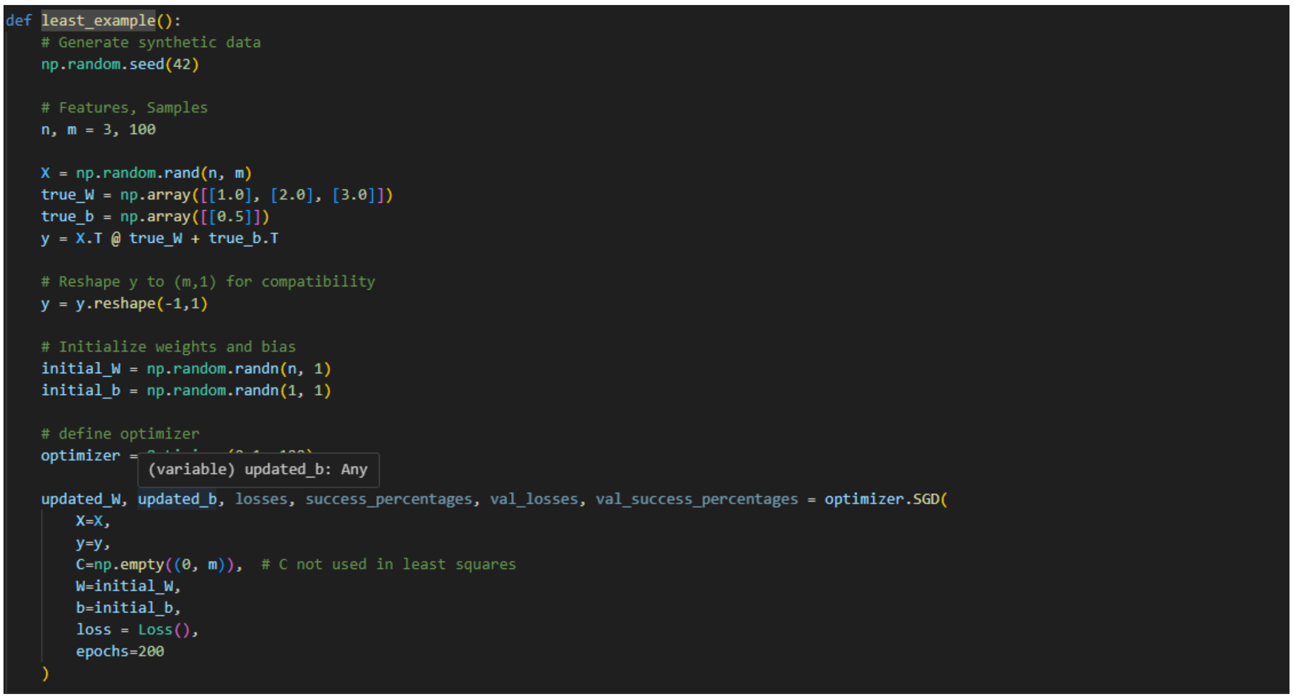


```python
        # Compute validation loss and success percentage
        if X_val is not None and y_val is not None:

            val_predictions = loss.softmax(X_val, W, b)
            val_running_loss = loss.cross_entropy_loss(val_predictions, C_val)
            val_predictions = loss.softmax_predictions(val_predictions)
            val_success_count = np.sum(val_predictions == y_val)


            val_success_percentage = (val_success_count / X_val.shape[1]) * 100
            val_losses.append(val_running_loss)
            val_success_percentages.append(val_success_percentage)

        # Check for convergence
        if k > 0 and residual_norms[-1] / residual_norms[-2] < convergence_threshold:
            print(f"Convergence reached at iteration {k}")
            break

    if plot:
        self.plot_sgd_results(losses, success_percentages, "Training Loss and Success Percentage")
        if X_val is not None and y_val is not None:
            self.plot_sgd_results(val_losses, val_success_percentages, "Validation Loss and Success Percentage")

    return W, b, losses, success_percentages, val_losses, val_success_percentages
```

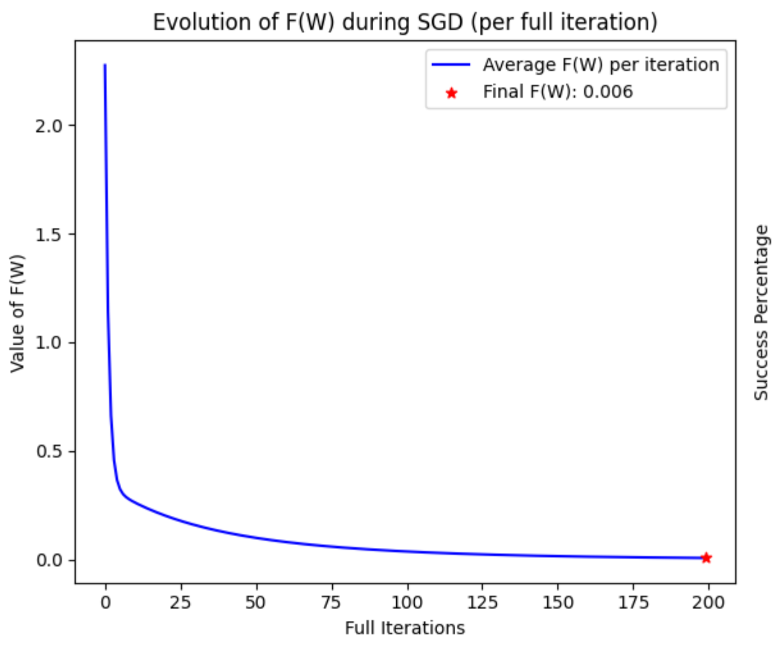
The implementation of SGD in this project not only performs parameter updates but also tracks performance metrics such as training and validation loss over time. This allows for effective monitoring of the optimization process and helps in assessing whether the model is converging properly.

Validation of the Optimizer

To verify the correctness of the optimizer, a simple synthetic dataset is generated for a least squares regression problem. The goal is to observe how the optimizer minimizes the loss over multiple training epochs and to ensure that the parameter updates align with theoretical expectations.



```python
def least_example():
    # Generate synthetic data
    np.random.seed(42)

    # Features, Samples
    n, m = 3, 100

    X = np.random.rand(n, m)
    true_W = np.array([[1.0], [2.0], [3.0]])
    true_b = np.array([[0.5]])
    y = X.T @ true_W + true_b.T

    # Reshape y to (m,1) for compatibility
    y = y.reshape(-1,1)

    # Initialize weights and bias
    initial_W = np.random.randn(n, 1)
    initial_b = np.random.randn(1, 1)

    # define optimizer
    optimizer = 
                (variable) updated_b: Any

    updated_W, updated_b, losses, success_percentages, val_losses, val_success_percentages = optimizer.SGD(
        X=X,
        y=y,
        C=np.empty((0, m)),  # C not used in least squares
        W=initial_W,
        b=initial_b,
        loss = Loss(),
        epochs=200
    )
```



The verification process starts by defining the least squares loss function and computing its gradients analytically. Then, using the generated dataset, the model is trained using the SGD optimizer, and its effectiveness is evaluated by tracking how the loss evolves over successive iterations. If the optimizer is correctly implemented, the loss should steadily decrease, indicating successful parameter optimization.

Discussion of Results

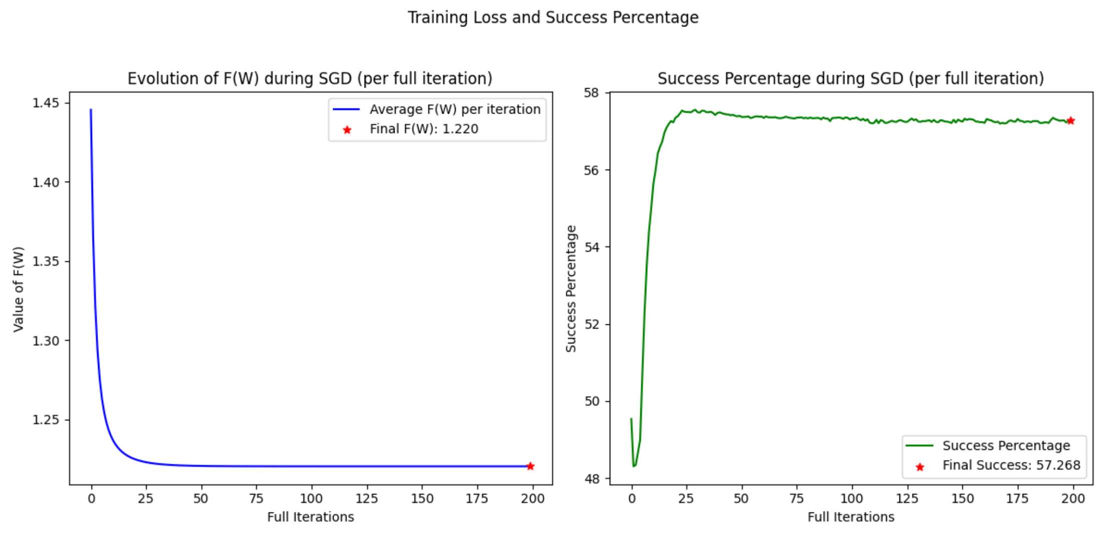
Observations from the least squares experiment show a rapid decline in the loss during the initial training epochs, signifying efficient learning. As training progresses, the loss stabilizes at a low value, confirming that the optimizer successfully converges to an optimal solution.

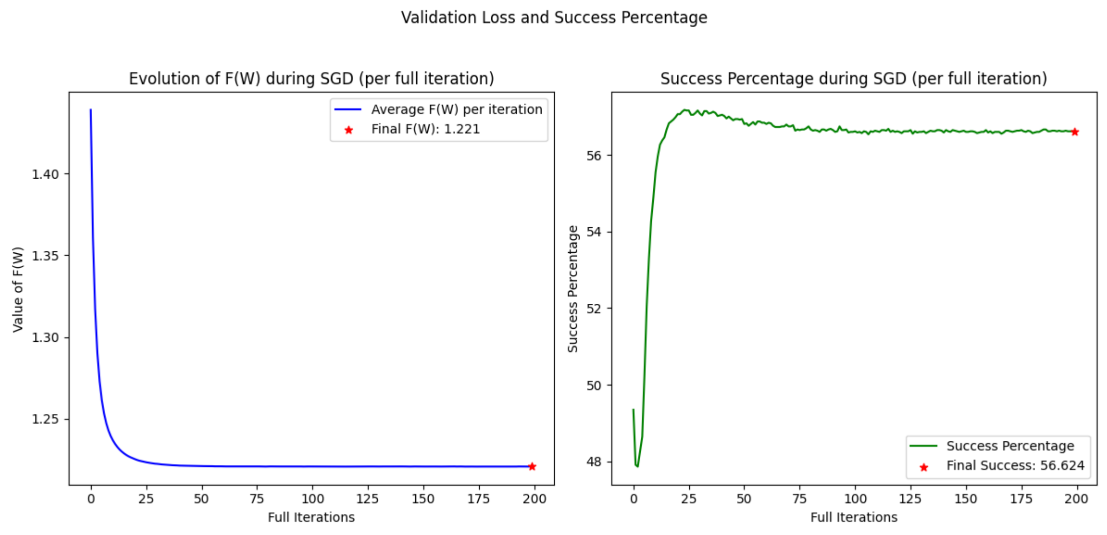
The effectiveness of the optimizer is demonstrated through its ability to consistently reduce the loss and reach a steady state. The convergence behavior suggests that the learning rate and batch size were appropriately chosen, balancing fast learning with stability. This experiment validates that the implemented SGD algorithm is functioning as expected and is capable of optimizing network parameters effectively.

1.3 Minimization of the Classifier using the Optimizer

To evaluate the effectiveness of our SGD implementation, we examine how it minimizes the softmax function when applied to different datasets. The following analysis explores the optimization process across various training configurations.

1. **"Peaks"**
   Hyperparamaters:
   Learning rate: 0.01 Batch-size: 200 over 200 epochs

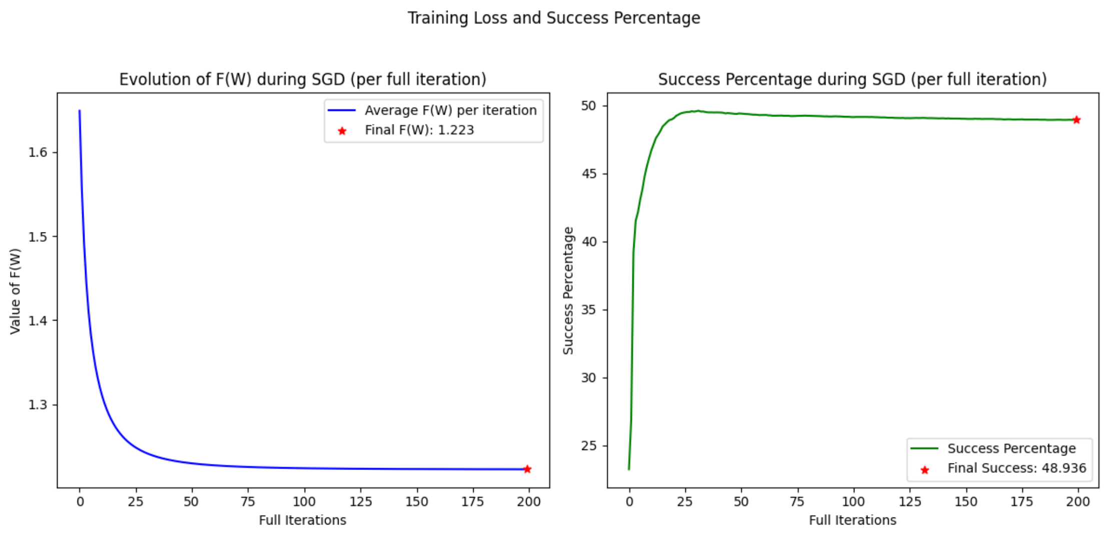


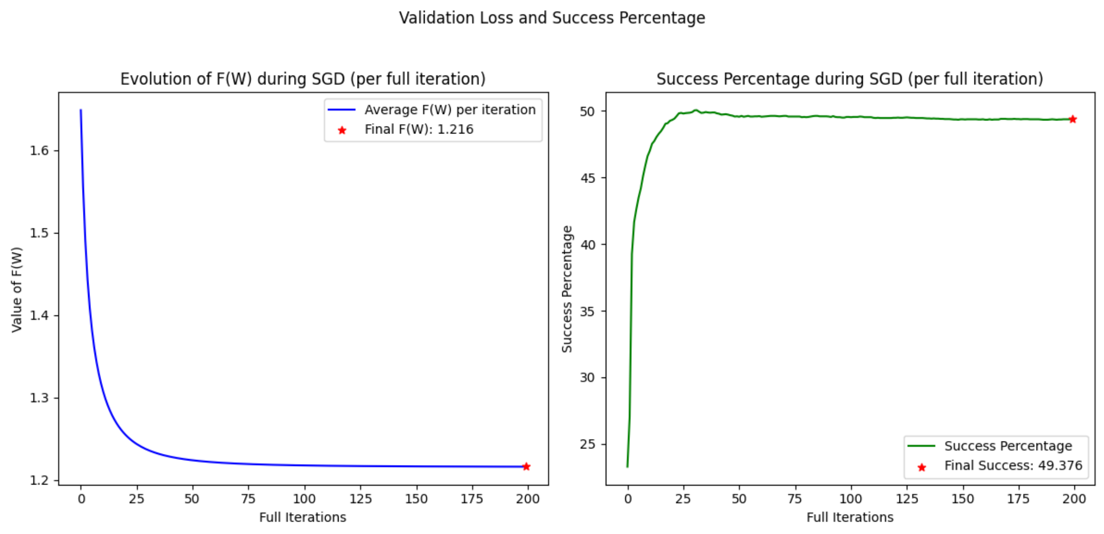


For the Peaks dataset, selecting an appropriate batch size was crucial to maintaining a steady improvement in classification accuracy throughout training. Previous trials with a decreasing learning rate led to sharp fluctuations in performance, making optimization unstable. To counteract this, we opted for a fixed learning rate of 0.01 and a batch size of 200, which provided a more controlled and consistent learning process over 200 epochs.

The final success rate stabilized at approximately 57%, which aligns with expectations given the dataset's complexity. Since the current model is based on a linear classifier, it struggles to capture the non-linear patterns present in the Peaks dataset. This suggests that enhancing the model with additional layers or alternative activation functions could lead to significant improvements in classification performance.

2. **"GMM"**
   Hyperparamaters:
   Learning rate: 0.01 Batch-size: 256 over 200 epochs





For the GMM dataset, careful selection of batch size and learning rate was necessary to ensure a smooth and stable optimization process. Initial experiments with a decreasing learning rate led to erratic fluctuations in accuracy, hindering convergence. To address this, a fixed learning rate of 0.01 and a batch size of 256 were chosen, which allowed for more consistent updates over 200 epochs, reducing instability in training.

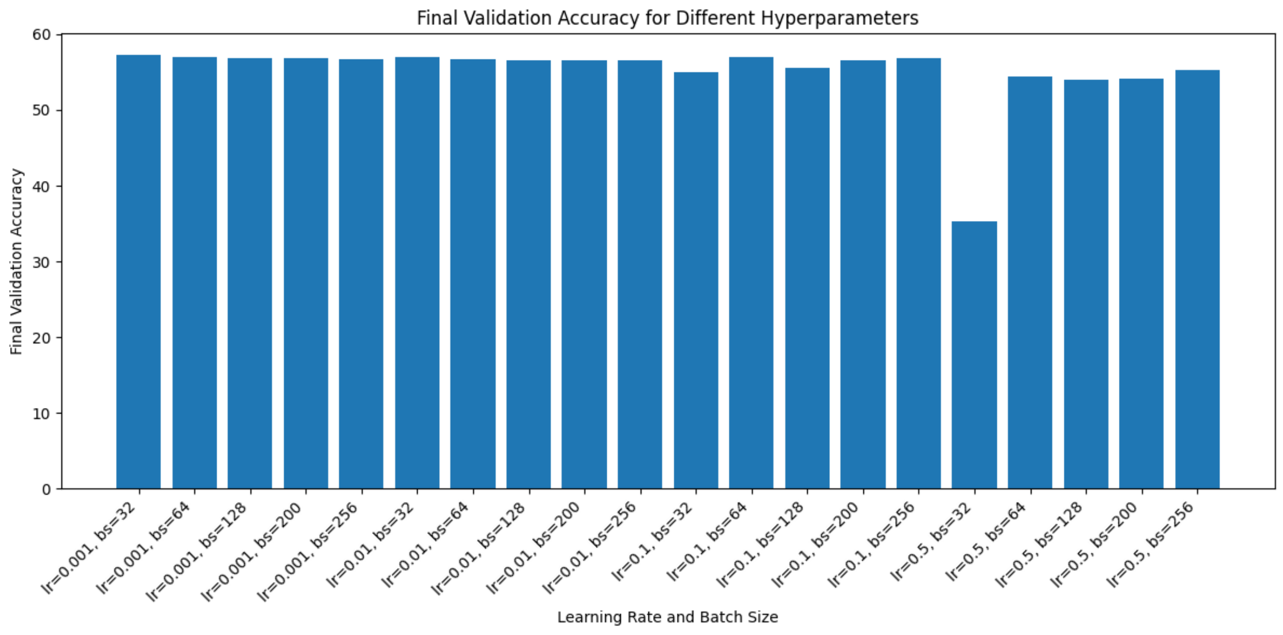
The final success rate settled at approximately 49%, which is expected given the dataset's complex, non-linearly separable nature. Since the current model is a linear classifier, it lacks the representational power needed to effectively distinguish between overlapping distributions in the GMM dataset. Future improvements, such as incorporating deeper architectures or non-linear activation functions, could lead to significantly better classification results.

The Effect of different learning rates and batch-size

In the experiments, the following hyperparameters were tested:

1. Learning Rates: [0.001, 0.01, 0.1, 0.5]
2. Batch Sizes: [32, 64, 128, 200, 256]

In the following bar graph, we can see the result of the experiment using the "Peaks" dataset over 50 epochs.



Best hyperparameters: Learning Rate = 0.1, Batch Size = 64

1. **Effect of Learning Rate:**
   A lower learning rate resulted in slower but more controlled convergence, requiring additional epochs to achieve a meaningful reduction in loss. While this provided stability in parameter updates, it also increased training time. Conversely, higher learning rates led to rapid initial progress but introduced instability, causing fluctuations in loss values and, in some cases, preventing the model from reaching an optimal solution. Finding the right balance was essential to avoid divergence while ensuring efficient learning.

2. **Effect of Batch Size:**
   Smaller batch sizes led to greater variability in loss trends due to higher variance in gradient estimates. This increased the model's ability to escape poor local minima but also introduced fluctuations that made training less predictable. On the other hand, larger batch sizes resulted in more stable updates with smoother loss curves, though they occasionally required additional epochs to achieve comparable reductions in loss. While reducing noise in gradient estimation, very large batch sizes sometimes hindered generalization, making it necessary to tune this parameter carefully.

2 Part II: The Neural Network

2.1 Intro

This project explores the implementation of both standard feedforward neural networks and Residual Networks (ResNet). These architectures are designed to process various data types and enhance classification performance by learning complex patterns effectively. The following sections provide an overview of each network type utilized in this project.

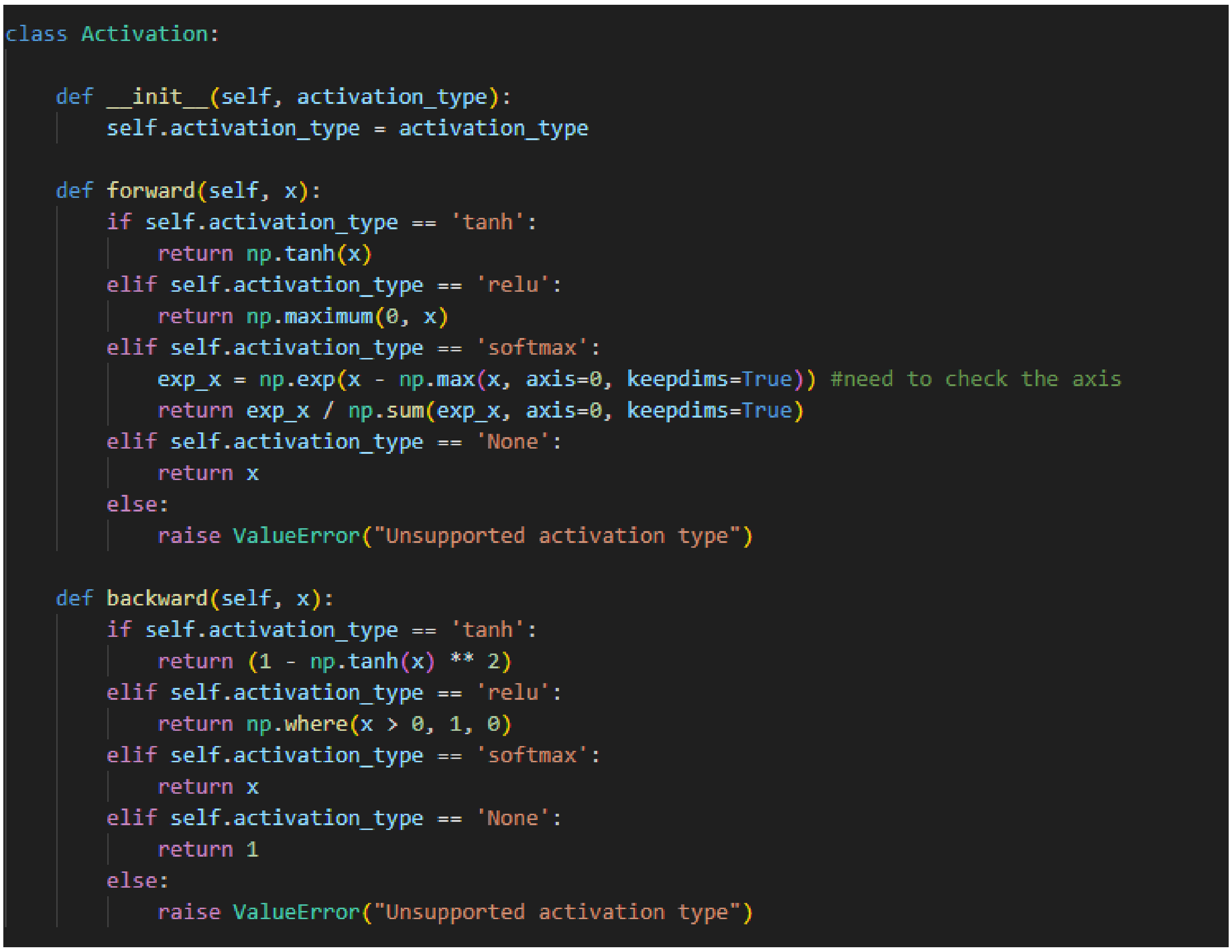
2.2 Activation

The Activation class in this project manages different activation functions essential for neural networks. These functions introduce non-linearity, allowing the network to complex patterns. The class includes a forward method to apply the activation function to inputs and a backward method to compute its derivative for gradient calculations.

Activation function that were implemented:

1. Tanh
2. ReLU
3. Softmax
4. None - no activation.

Code implementation:



```python
class Activation:

    def __init__(self, activation_type):
        self.activation_type = activation_type

    def forward(self, x):
        if self.activation_type == 'tanh':
            return np.tanh(x)
        elif self.activation_type == 'relu':
            return np.maximum(0, x)
        elif self.activation_type == 'softmax':
            exp_x = np.exp(x - np.max(x, axis=0, keepdims=True)) #need to check the axis
            return exp_x / np.sum(exp_x, axis=0, keepdims=True)
        elif self.activation_type == 'None':
            return x
        else:
            raise ValueError("Unsupported activation type")

    def backward(self, x):
        if self.activation_type == 'tanh':
            return (1 - np.tanh(x) ** 2)
        elif self.activation_type == 'relu':
            return np.where(x > 0, 1, 0)
        elif self.activation_type == 'softmax':
            return x
        elif self.activation_type == 'None':
            return 1
        else:
            raise ValueError("Unsupported activation type")
```

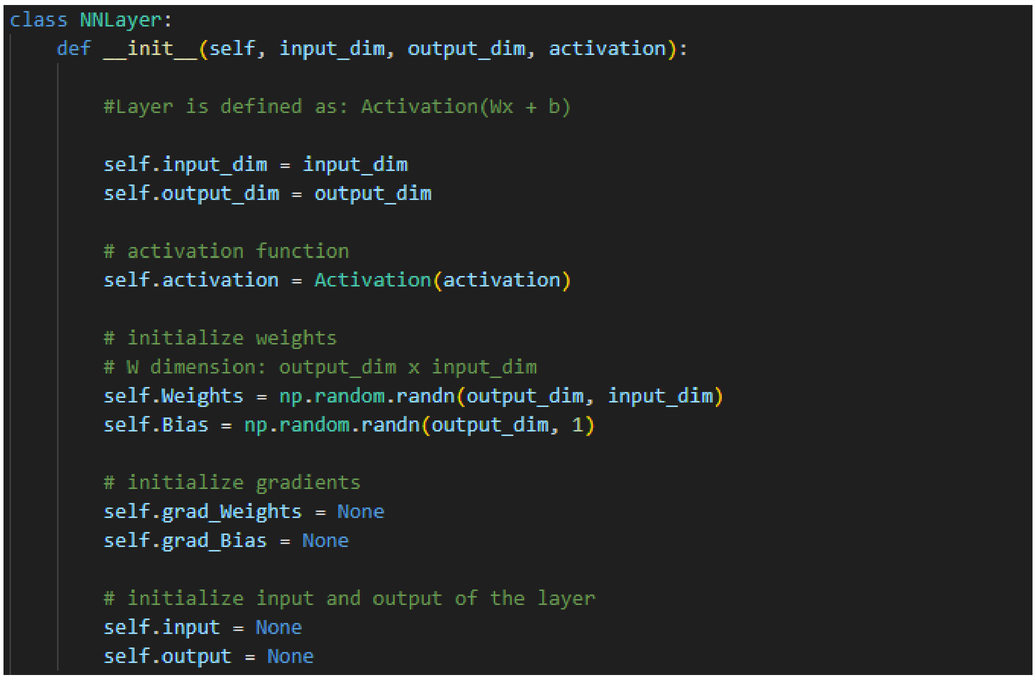
2.3 Neural Network

2.3.1 Neural Layer

Layer Overview:

The NNLayer class defines a single computational layer within a neural network, handling both forward and backward propagation—key processes for model training.
Each layer follows the transformation $\sigma(Wx + b)$ and is initialized with the following attributes:

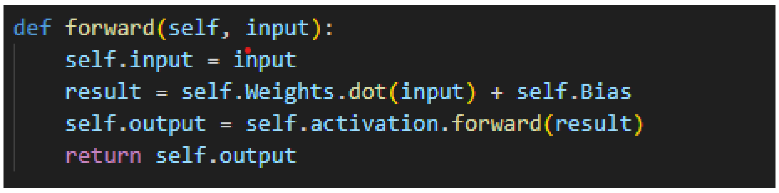
1. $input_dim$: Number of input features.
2. $output_dim$: Number of output features.
3. $activation$: Specifies the activation function (e.g., ReLU, Tanh, Softmax).
4. $Weights$: Randomly initialized weight matrix.
5. $Bias$: Randomly initialized bias vector.
6. $grad_Weights$: Stores the gradient of the loss with respect to the weights.
7. $grad_Bias$: Stores the gradient of the loss with respect to the bias.
8. $input$: Stores the input data fed into the layer.
9. $output$: Holds the output produced by the layer after applying the activation function.



```
class NNLayer:
    def __init__(self, input_dim, output_dim, activation):

        #Layer is defined as: Activation(Wx + b)

        self.input_dim = input_dim
        self.output_dim = output_dim

        # activation function
        self.activation = Activation(activation)

        # initialize weights
        # W dimension: output_dim x input_dim
        self.Weights = np.random.randn(output_dim, input_dim)
        self.Bias = np.random.randn(output_dim, 1)

        # initialize gradients
        self.grad_Weights = None
        self.grad_Bias = None

        # initialize input and output of the layer
        self.input = None
        self.output = None
```

Layer Forward pass:

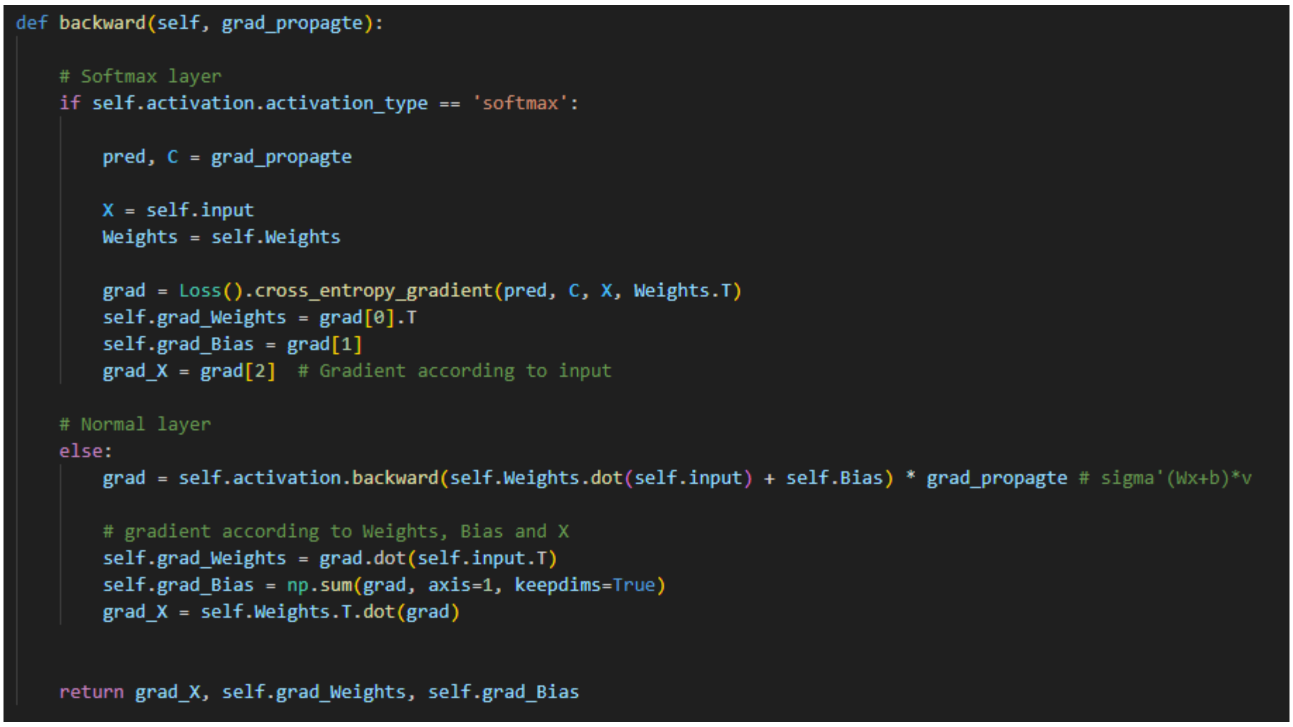
The forward method executes the following operations: It first computes the linear transformation $W_1x + b$, then applies the chosen activation function to the result. This transformation refines the input data, enabling it to be effectively processed by subsequent layers or the final output layer.



```python
def forward(self, input):
    self.input = input
    result = self.Weights.dot(input) + self.Bias
    self.output = self.activation.forward(result)
    return self.output
```

Layer Backward pass:

The backward pass in the NNLayer class plays a fundamental role in training the neural network. It calculates the gradients of the loss function with respect to the layer's parameters (weights and biases) and its input. These gradients are crucial for adjusting the model's parameters during optimization, enabling the network to improve its predictions over time. Each layer computes its respective gradients, which are then propagated backward to refine the updates in earlier layers.



```python
def backward(self, grad_propagte):

    # Softmax layer
    if self.activation.activation_type == 'softmax':

        pred, C = grad_propagte

        X = self.input
        Weights = self.Weights

        grad = Loss().cross_entropy_gradient(pred, C, X, Weights.T)
        self.grad_Weights = grad[0].T
        self.grad_Bias = grad[1]
        grad_X = grad[2]  # Gradient according to input

    # Normal layer
    else:
        grad = self.activation.backward(self.Weights.dot(self.input) + self.Bias) * grad_propagte # sigma'(Wx+b)*v

        # gradient according to Weights, Bias and X
        self.grad_Weights = grad.dot(self.input.T)
        self.grad_Bias = np.sum(grad, axis=1, keepdims=True)
        grad_X = self.Weights.T.dot(grad)


    return grad_X, self.grad_Weights, self.grad_Bias
```

2.3.2 Neural Network

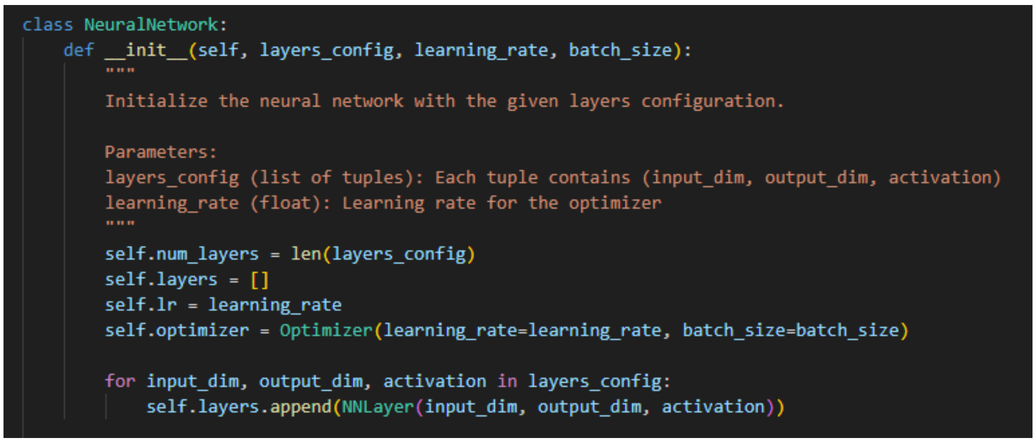
Network Description:

The NeuralNetwork class manages the construction, training, and assessment of a neural network. It facilitates multiple layers, each with specific configurations, and includes methods for executing forward propagation, backpropagation, and updating model parameters.

Initialization:

The NeuralNetwork class is initialized with the following parameters:

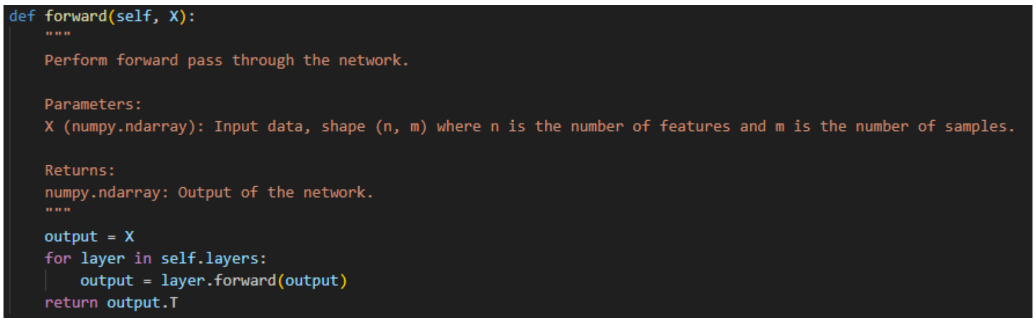
1. *num_layers*: number of layers.
2. *layers*: are initialized from a list of tuples, where each tuple contains the input dimension, output dimension, and activation function for a layer.
3. *lr*: The learning rate for the optimizer.
4. *optimizer*: initialized with the learning rate and batch size



```python
class NeuralNetwork:
    def __init__(self, layers_config, learning_rate, batch_size):
        """
        Initialize the neural network with the given layers configuration.

        Parameters:
        layers_config (list of tuples): Each tuple contains (input_dim, output_dim, activation)
        learning_rate (float): Learning rate for the optimizer
        """
        self.num_layers = len(layers_config)
        self.layers = []
        self.lr = learning_rate
        self.optimizer = Optimizer(learning_rate=learning_rate, batch_size=batch_size)

        for input_dim, output_dim, activation in layers_config:
            self.layers.append(NNLayer(input_dim, output_dim, activation))
```

Network Forward pass:

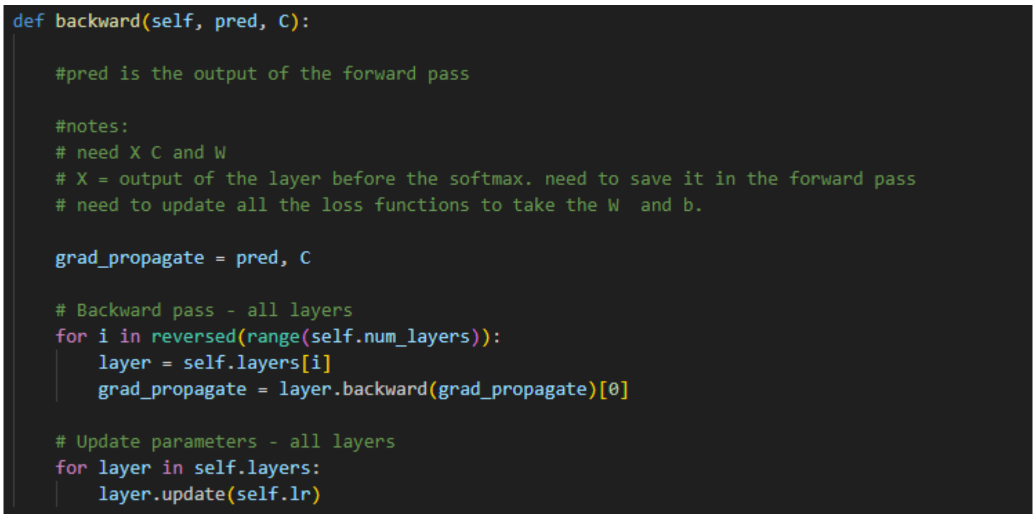
The forward method follows a two-step process: first, it calculates the linear transformation $W_1x + b$. Next, it applies the designated activation function to introduce non-linearity. This transformation ensures that the output is appropriately formatted for the subsequent layer or final classification.



```python
def forward(self, X):
    """
    Perform forward pass through the network.

    Parameters:
    X (numpy.ndarray): Input data, shape (n, m) where n is the number of features and m is the number of samples.

    Returns:
    numpy.ndarray: Output of the network.
    """
    output = X
    for layer in self.layers:
        output = layer.forward(output)
    return output.T
```

Network Backward pass:

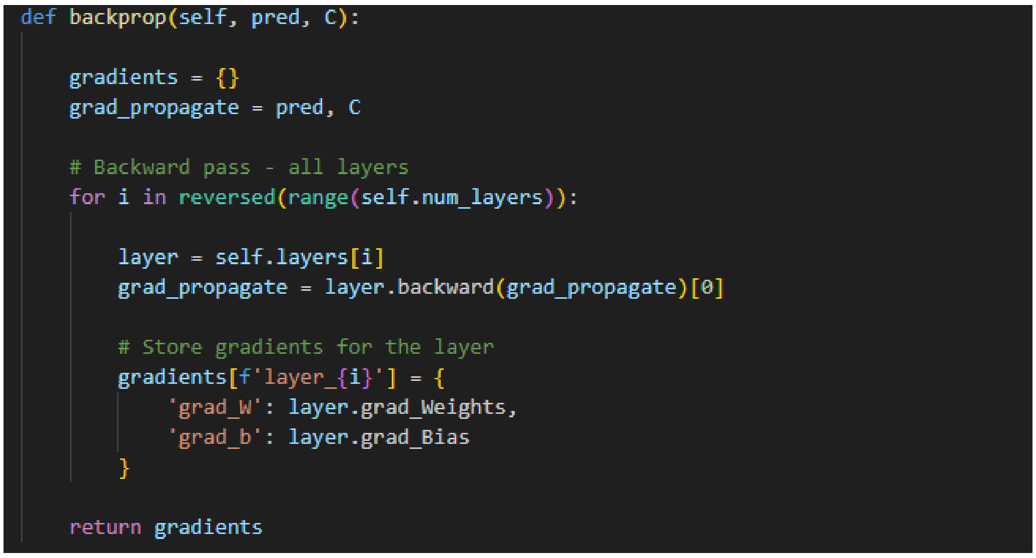
The backward method executes the backpropagation process across the network. It receives the predicted outputs and corresponding true labels, calculates the gradients of the loss function with respect to the weights, biases, and input activations of each layer, and applies these gradients to update the parameters via the optimizer. This step is critical for adjusting the network's parameters to minimize the loss and improve performance over successive iterations.



```python
def backward(self, pred, C):

    #pred is the output of the forward pass

    #notes:
    # need X C and W
    # X = output of the layer before the softmax. need to save it in the forward pass
    # need to update all the loss functions to take the W  and b.

    grad_propagate = pred, C

    # Backward pass - all layers
    for i in reversed(range(self.num_layers)):
        layer = self.layers[i]
        grad_propagate = layer.backward(grad_propagate)[0]

    # Update parameters - all layers
    for layer in self.layers:
        layer.update(self.lr)
```

Backprop (for Testing):

The backprop method functions similarly to the backward method but is intended solely for verification purposes. It performs the backward pass without modifying the network's parameters, instead returning the computed gradients for each layer. This method is particularly useful for validating the accuracy of gradient computations, such as in gradient tests that ensure the correctness of backpropagation throughout the entire network.



```python
def backprop(self, pred, C):

    gradients = {}
    grad_propagate = pred, C

    # Backward pass - all layers
    for i in reversed(range(self.num_layers)):

        layer = self.layers[i]
        grad_propagate = layer.backward(grad_propagate)[0]

        # Store gradients for the layer
        gradients[f'layer_{i}'] = {
            'grad_W': layer.grad_Weights,
            'grad_b': layer.grad_Bias
        }

    return gradients
```

Training Process:

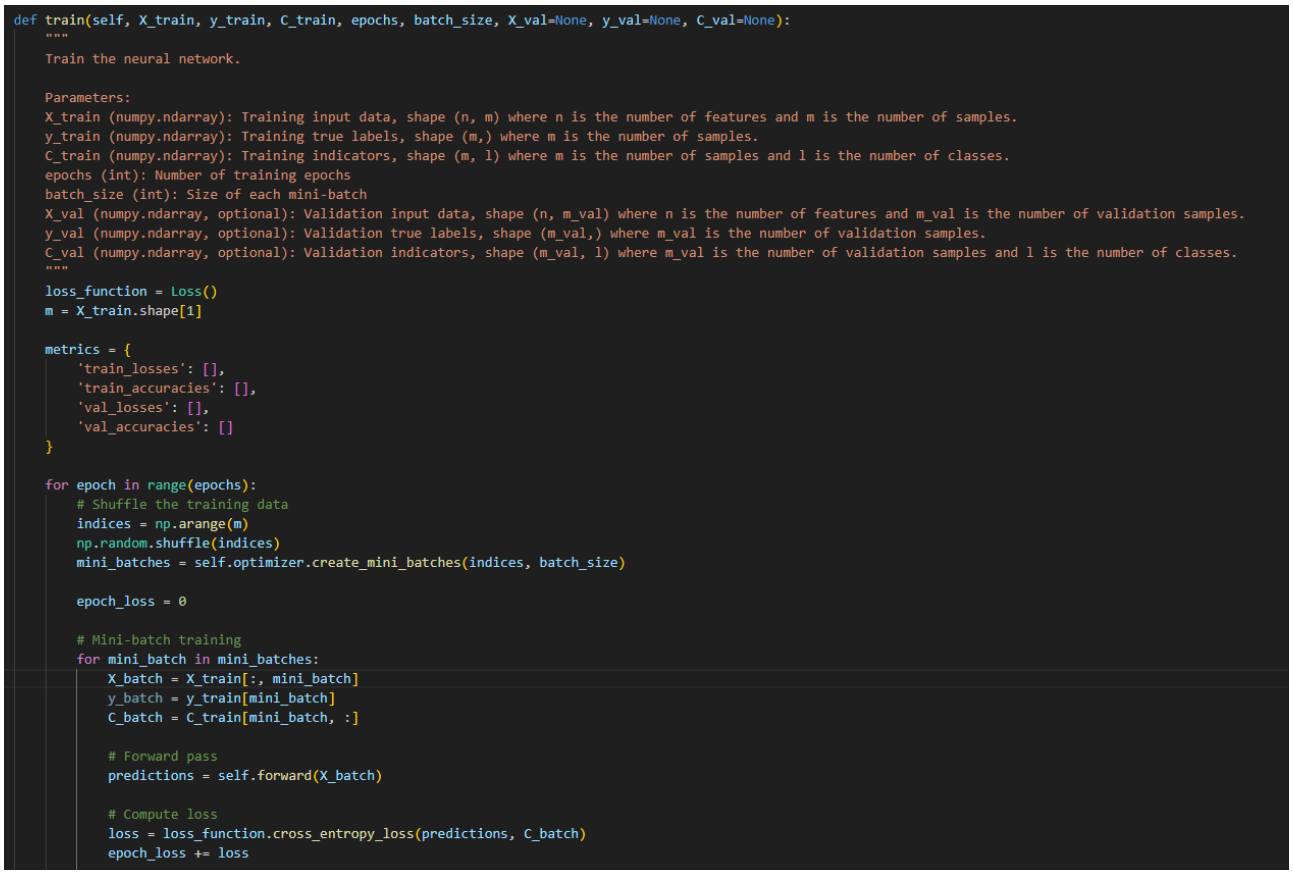
The `train` function is responsible for optimizing the neural network by iteratively updating its parameters using mini-batch gradient descent. The function trains the network over multiple epochs, computing the loss and updating the model's parameters based on the gradients obtained during backpropagation. It also evaluates performance using accuracy metrics on both training and validation data.

Training Procedure:

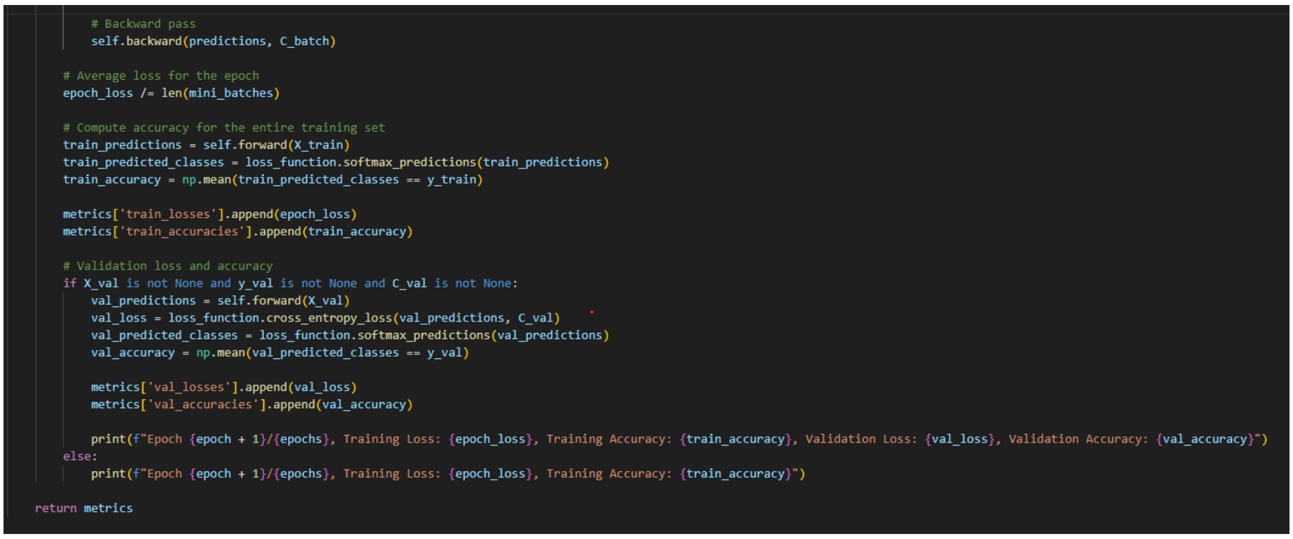
1. **Initialization:** The function starts by setting up the loss function and initializing tracking metrics for training and validation performance.
2. **Shuffling and Mini-batching:** The training data is randomly shuffled at the beginning of each epoch, and mini-batches are created for stochastic gradient descent.

3. **Mini-batch Training:**
   - Each mini-batch is passed through the network during the forward pass to compute predictions.
   - The cross-entropy loss is calculated based on the predictions and true labels.
   - The backward pass computes gradients, updating network parameters using the optimizer.
4. **Performance Evaluation:** After processing all mini-batches, the function computes the average training loss and evaluates accuracy on the entire training set.
5. **Validation (Optional):** If validation data is provided, the function also evaluates loss and accuracy on the validation set.
6. **Logging and Output:** Training progress is printed at each epoch, displaying loss and accuracy for both training and validation sets.

This training loop ensures that the model effectively learns patterns from the data by continuously refining its parameters while tracking improvements over time.

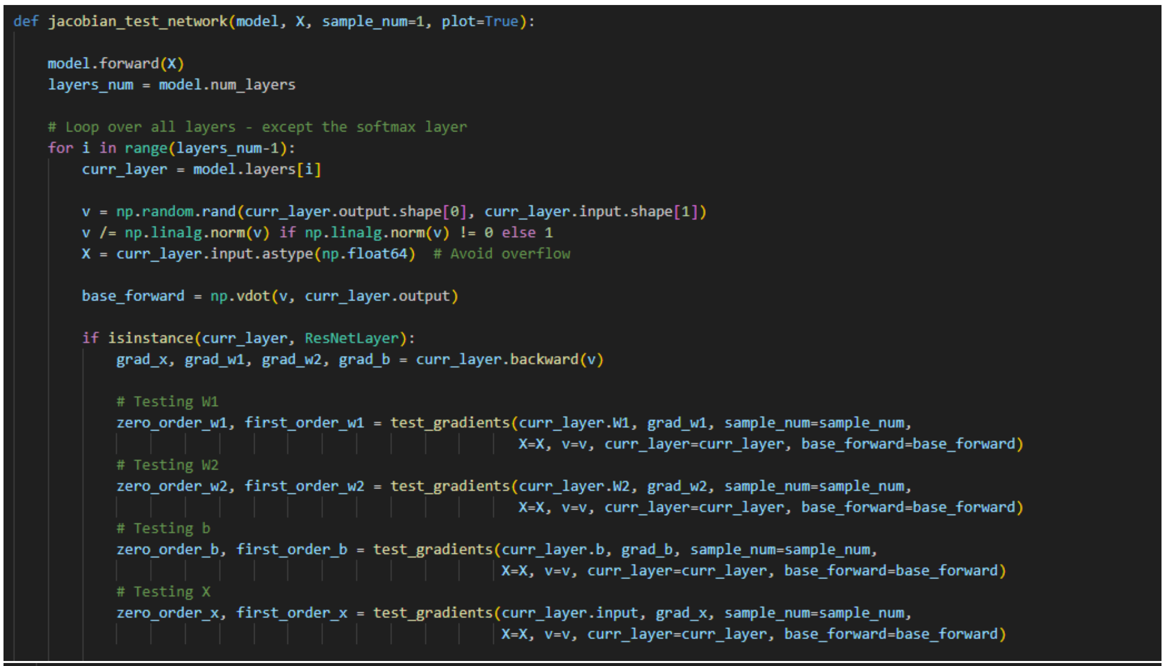


```python
def train(self, X_train, y_train, C_train, epochs, batch_size, X_val=None, y_val=None, C_val=None):
    """
    Train the neural network.

    Parameters:
    X_train (numpy.ndarray): Training input data, shape (n, m) where n is the number of features and m is the number of samples.
    y_train (numpy.ndarray): Training true labels, shape (m,) where m is the number of samples.
    C_train (numpy.ndarray): Training indicators, shape (m, l) where m is the number of samples and l is the number of classes.
    epochs (int): Number of training epochs
    batch_size (int): Size of each mini-batch
    X_val (numpy.ndarray, optional): Validation input data, shape (n, m_val) where n is the number of features and m_val is the number of validation samples.
    y_val (numpy.ndarray, optional): Validation true labels, shape (m_val,) where m_val is the number of validation samples.
    C_val (numpy.ndarray, optional): Validation indicators, shape (m_val, l) where m_val is the number of validation samples and l is the number of classes.
    """
    loss_function = Loss()
    m = X_train.shape[1]

    metrics = {
        'train_losses': [],
        'train_accuracies': [],
        'val_losses': [],
        'val_accuracies': []
    }

    for epoch in range(epochs):
        # Shuffle the training data
        indices = np.arange(m)
        np.random.shuffle(indices)
        mini_batches = self.optimizer.create_mini_batches(indices, batch_size)

        epoch_loss = 0

        # Mini-batch training
        for mini_batch in mini_batches:
            X_batch = X_train[:, mini_batch]
            y_batch = y_train[mini_batch]
            C_batch = C_train[mini_batch, :]

            # Forward pass
            predictions = self.forward(X_batch)

            # Compute loss
            loss = loss_function.cross_entropy_loss(predictions, C_batch)
            epoch_loss += loss
```

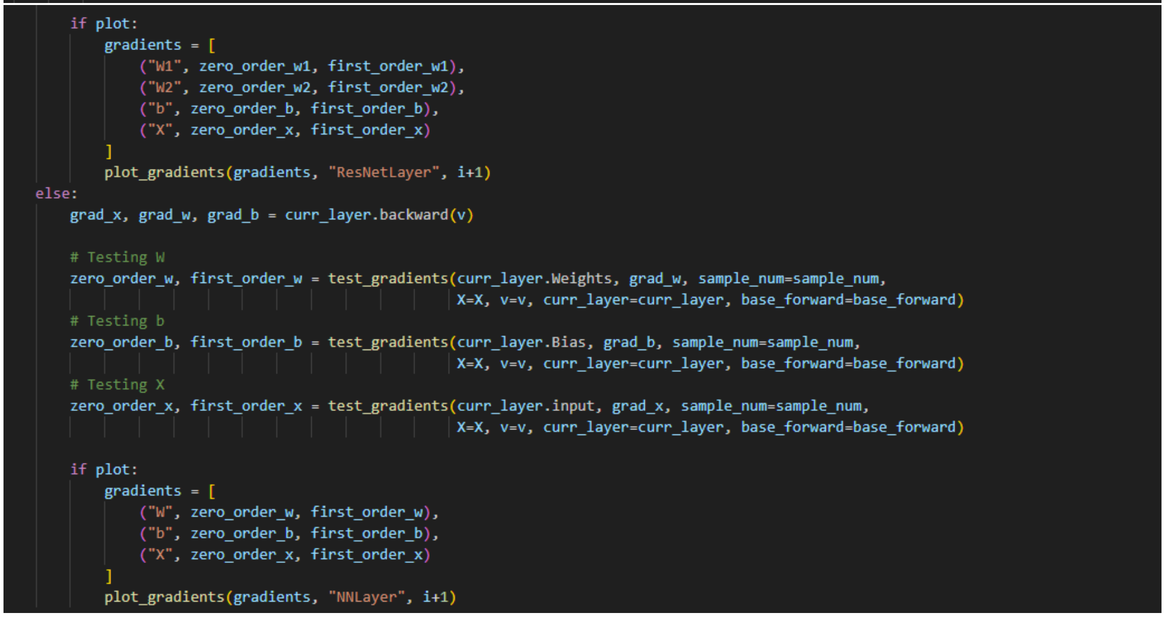

```
                # Backward pass
                self.backward(predictions, C_batch)

            # Average loss for the epoch
            epoch_loss /= len(mini_batches)

            # Compute accuracy for the entire training set
            train_predictions = self.forward(X_train)
            train_predicted_classes = loss_function.softmax_predictions(train_predictions)
            train_accuracy = np.mean(train_predicted_classes == y_train)

            metrics['train_losses'].append(epoch_loss)
            metrics['train_accuracies'].append(train_accuracy)

            # Validation loss and accuracy
            if X_val is not None and y_val is not None and C_val is not None:
                val_predictions = self.forward(X_val)
                val_loss = loss_function.cross_entropy_loss(val_predictions, C_val)
                val_predicted_classes = loss_function.softmax_predictions(val_predictions)
                val_accuracy = np.mean(val_predicted_classes == y_val)

                metrics['val_losses'].append(val_loss)
                metrics['val_accuracies'].append(val_accuracy)

                print(f"Epoch {epoch + 1}/{epochs}, Training Loss: {epoch_loss}, Training Accuracy: {train_accuracy}, Validation Loss: {val_loss}, Validation Accuracy: {val_accuracy}")
            else:
                print(f"Epoch {epoch + 1}/{epochs}, Training Loss: {epoch_loss}, Training Accuracy: {train_accuracy}")

        return metrics
```

2.3.3 Jacobian test - Layers

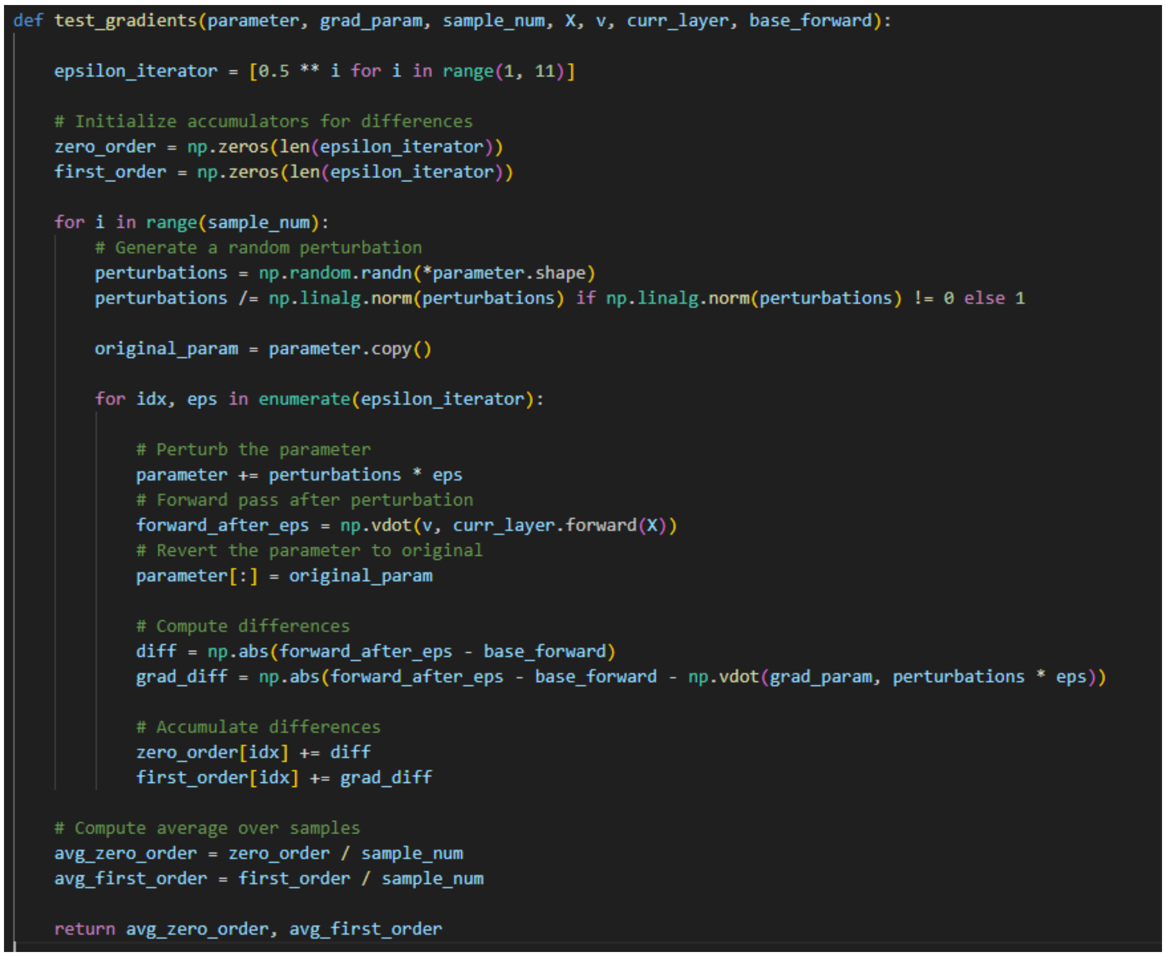
The Jacobian test serves as a validation technique to assess the accuracy of backpropagation by comparing analytically derived gradients with their numerical approximations. This is achieved by applying small perturbations to each parameter (weights, biases, and inputs) and observing the corresponding output changes. The method systematically evaluates all layers of the network, excluding the softmax layer, by leveraging the backprop function to compute gradients and contrasting them with perturbed forward outputs. A consistent decrease in these differences confirms the reliability of the computed gradients, ensuring the correctness of the backpropagation implementation.



```python
def jacobian_test_network(model, X, sample_num=1, plot=True):

    model.forward(X)
    layers_num = model.num_layers

    # Loop over all layers - except the softmax layer
    for i in range(layers_num-1):
        curr_layer = model.layers[i]

        v = np.random.rand(curr_layer.output.shape[0], curr_layer.input.shape[1])
        v /= np.linalg.norm(v) if np.linalg.norm(v) != 0 else 1
        X = curr_layer.input.astype(np.float64)  # Avoid overflow

        base_forward = np.vdot(v, curr_layer.output)

        if isinstance(curr_layer, ResNetLayer):
            grad_x, grad_w1, grad_w2, grad_b = curr_layer.backward(v)

            # Testing W1
            zero_order_w1, first_order_w1 = test_gradients(curr_layer.W1, grad_w1, sample_num=sample_num,
                                                          X=X, v=v, curr_layer=curr_layer, base_forward=base_forward)
            # Testing W2
            zero_order_w2, first_order_w2 = test_gradients(curr_layer.W2, grad_w2, sample_num=sample_num,
                                                          X=X, v=v, curr_layer=curr_layer, base_forward=base_forward)
            # Testing b
            zero_order_b, first_order_b = test_gradients(curr_layer.b, grad_b, sample_num=sample_num,
                                                        X=X, v=v, curr_layer=curr_layer, base_forward=base_forward)
            # Testing X
            zero_order_x, first_order_x = test_gradients(curr_layer.input, grad_x, sample_num=sample_num,
                                                        X=X, v=v, curr_layer=curr_layer, base_forward=base_forward)
```



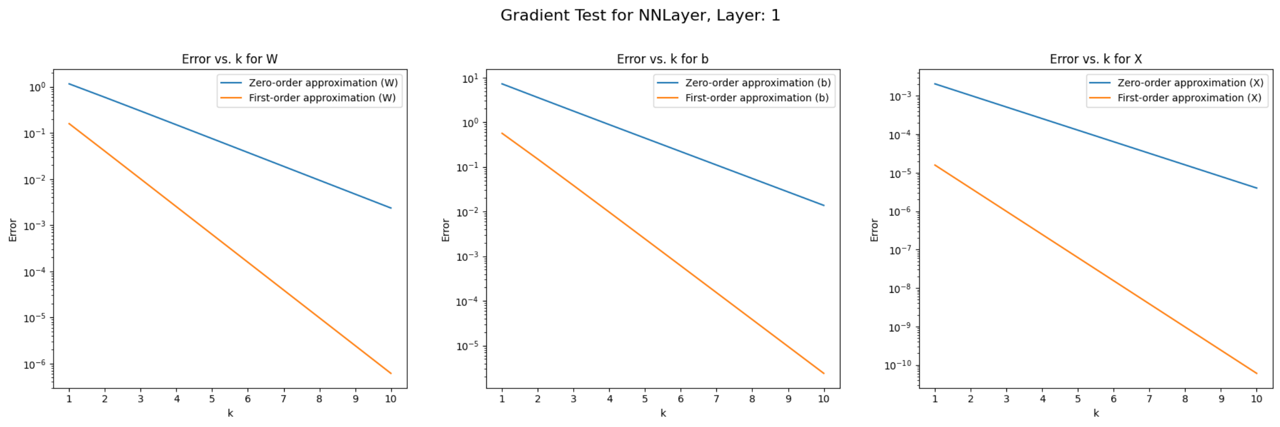
```python
            if plot:
                gradients = [
                    ("W1", zero_order_w1, first_order_w1),
                    ("W2", zero_order_w2, first_order_w2),
                    ("b", zero_order_b, first_order_b),
                    ("X", zero_order_x, first_order_x)
                ]
                plot_gradients(gradients, "ResNetLayer", i+1)
        else:
            grad_x, grad_w, grad_b = curr_layer.backward(v)

            # Testing W
            zero_order_w, first_order_w = test_gradients(curr_layer.Weights, grad_w, sample_num=sample_num,
                                                        X=X, v=v, curr_layer=curr_layer, base_forward=base_forward)
            # Testing b
            zero_order_b, first_order_b = test_gradients(curr_layer.Bias, grad_b, sample_num=sample_num,
                                                        X=X, v=v, curr_layer=curr_layer, base_forward=base_forward)
            # Testing X
            zero_order_x, first_order_x = test_gradients(curr_layer.input, grad_x, sample_num=sample_num,
                                                        X=X, v=v, curr_layer=curr_layer, base_forward=base_forward)

            if plot:
                gradients = [
                    ("W", zero_order_w, first_order_w),
                    ("b", zero_order_b, first_order_b),
                    ("X", zero_order_x, first_order_x)
                ]
                plot_gradients(gradients, "NNLayer", i+1)
```

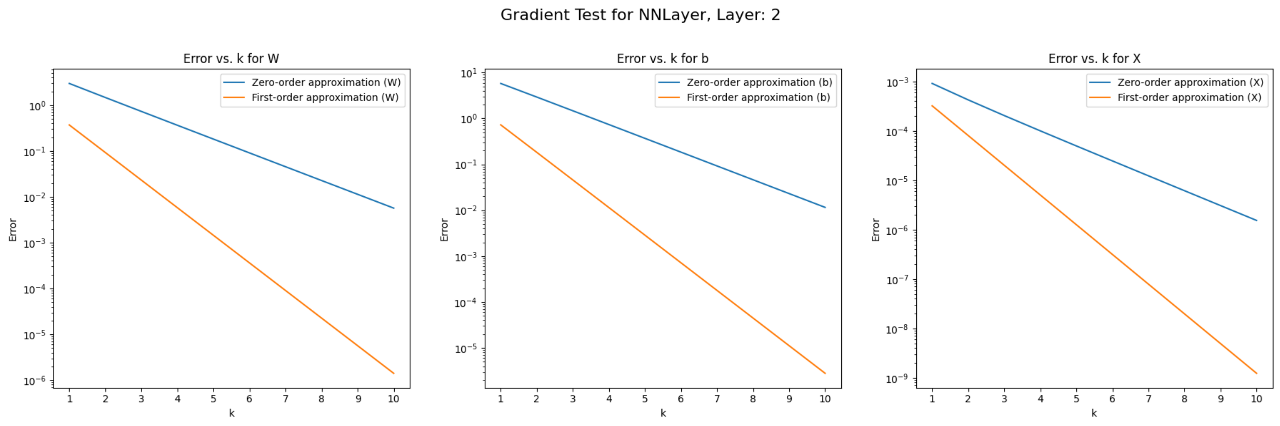


```python
def test_gradients(parameter, grad_param, sample_num, X, v, curr_layer, base_forward):

    epsilon_iterator = [0.5 ** i for i in range(1, 11)]

    # Initialize accumulators for differences
    zero_order = np.zeros(len(epsilon_iterator))
    first_order = np.zeros(len(epsilon_iterator))

    for i in range(sample_num):
        # Generate a random perturbation
        perturbations = np.random.randn(*parameter.shape)
        perturbations /= np.linalg.norm(perturbations) if np.linalg.norm(perturbations) != 0 else 1

        original_param = parameter.copy()

        for idx, eps in enumerate(epsilon_iterator):

            # Perturb the parameter
            parameter += perturbations * eps
            # Forward pass after perturbation
            forward_after_eps = np.vdot(v, curr_layer.forward(X))
            # Revert the parameter to original
            parameter[:] = original_param

            # Compute differences
            diff = np.abs(forward_after_eps - base_forward)
            grad_diff = np.abs(forward_after_eps - base_forward - np.vdot(grad_param, perturbations * eps))

            # Accumulate differences
            zero_order[idx] += diff
            first_order[idx] += grad_diff

    # Compute average over samples
    avg_zero_order = zero_order / sample_num
    avg_first_order = first_order / sample_num

    return avg_zero_order, avg_first_order
```

Here is a Jacobian test result for "Swiss Roll" dataset with the following network:

$$model = NeuralNetwork([(2, 10,' tanh'), (10, 10,' tanh'), (10, 2,' softmax')], 0.01, 32)$$

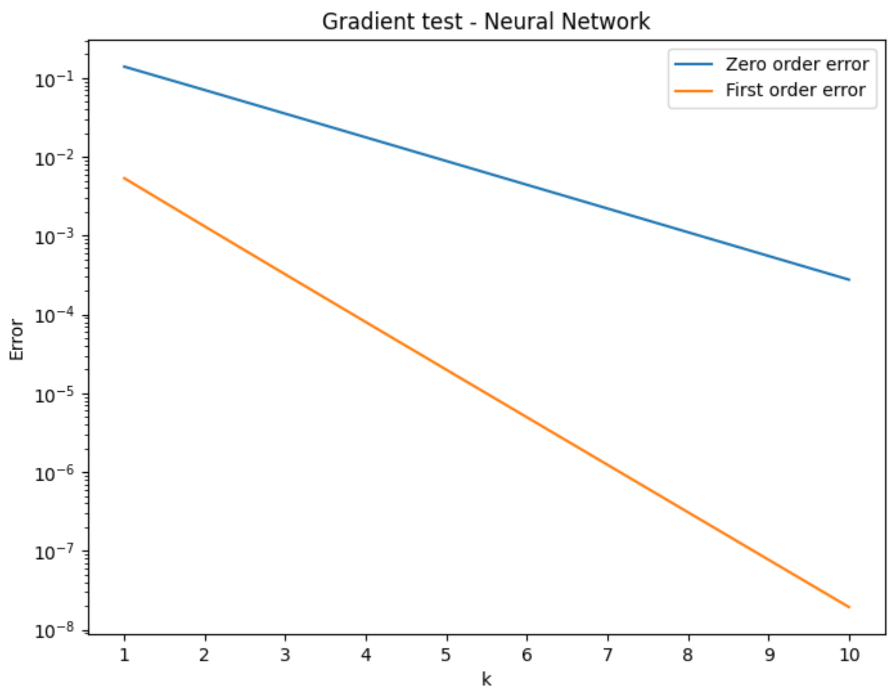




The graphs from the Jacobian test shows three decreasing curves for W, b, and X, each representing the zero-order and first-order errors. Both the zero-order and first-order differences decrease as the perturbation scale ϵ becomes smaller, which indicates that the numerical approximations are converging to the analytical gradients. The first-order errors decrease at a faster rate compared to the zero-order errors, demonstrating that the computed gradients are a good approximation of the true gradient values. The consistent decrease in both error types for all three parameters indicates that the network's gradient computations are accurate, confirming that the backpropagation implementation is functioning properly across layers.

2.3.4 Gradient test - Layers + Classifier

The *gradient_test_network* function verifies gradient computation for the entire network, including all layers and softmax, by comparing backpropagation results with numerical approximations. It perturbs all parameters, measures loss changes, and evaluates zero-order and first-order errors to ensure correct gradient calculations across the full model.
Building upon the previous example, here is the gradient test of the entire network:



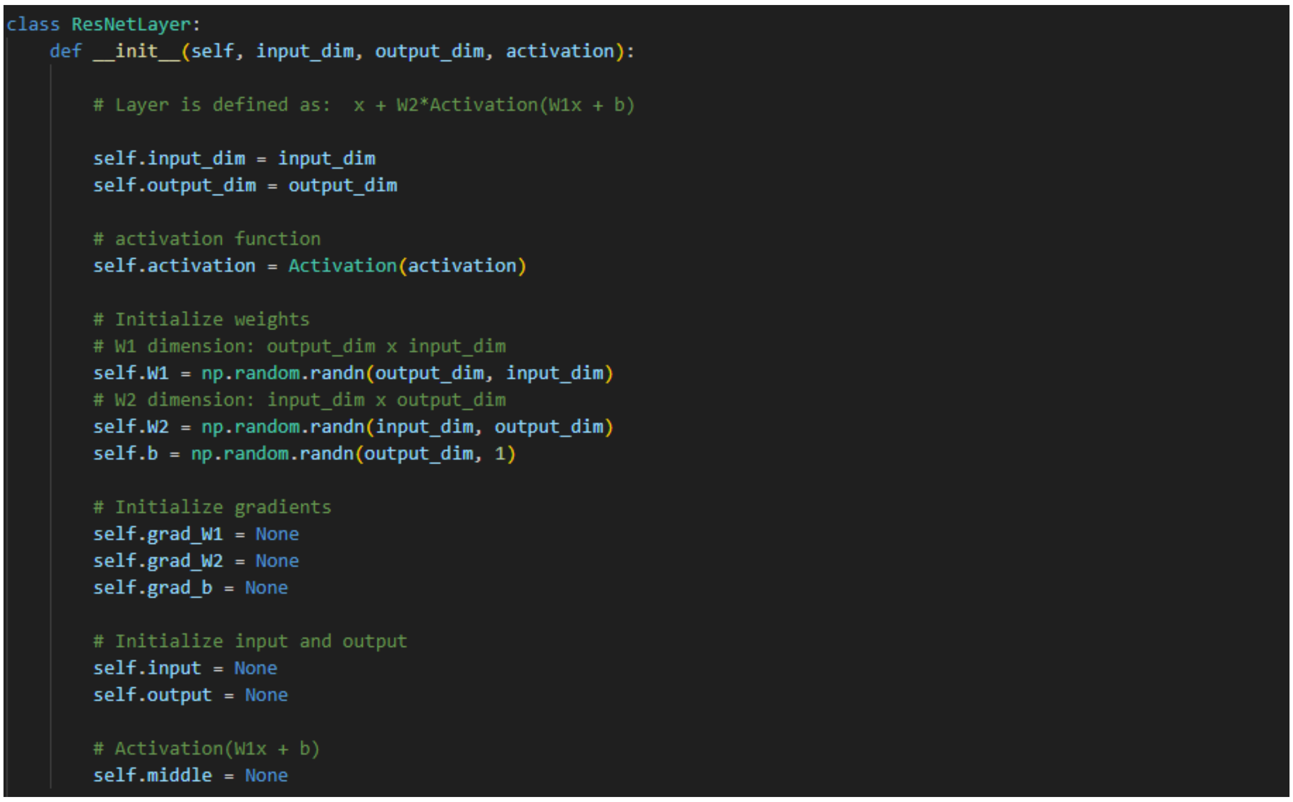
The gradient test graph for the entire network illustrates a consistent decline in both zero-order and first-order errors as the perturbation scale ϵ decreases. Notably, the first-order error diminishes at a faster rate, indicating that the computed gradients align closely with their numerical approximations. This trend serves as strong evidence that backpropagation is correctly implemented, ensuring that the model's gradient updates are accurate. Consequently, this validation confirms the reliability of the backpropagation mechanism, facilitating stable and efficient optimization throughout training.

2.4 Residual Neural Network

2.4.1 Resnet Layer

Layer Description:

The ResnetLayer class defines a single layer within a residual network.
This layer follows the structure $x + W_2\sigma(W_1x + b)$, where the residual connection helps mitigate issues like vanishing gradients in deeper networks. It is initialized with the following parameters:



```python
class ResNetLayer:
    def __init__(self, input_dim, output_dim, activation):

        # Layer is defined as:  x + W2*Activation(W1x + b)

        self.input_dim = input_dim
        self.output_dim = output_dim

        # activation function
        self.activation = Activation(activation)

        # Initialize weights
        # W1 dimension: output_dim x input_dim
        self.W1 = np.random.randn(output_dim, input_dim)
        # W2 dimension: input_dim x output_dim
        self.W2 = np.random.randn(input_dim, output_dim)
        self.b = np.random.randn(output_dim, 1)

        # Initialize gradients
        self.grad_W1 = None
        self.grad_W2 = None
        self.grad_b = None

        # Initialize input and output
        self.input = None
        self.output = None

        # Activation(W1x + b)
        self.middle = None
```

Layer Forward pass:

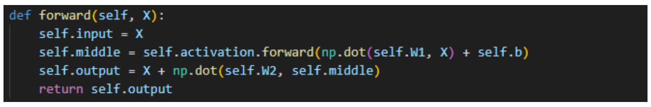
The forward method executes the following operations: It first calculates

$$\text{self.middle} = \sigma(W_1x + b)$$

where the activation function is applied to the linear transformation. Storing this intermediate result optimizes computational efficiency during the backward pass.
Next, the final output of the layer is computed as

$$\text{self.output} = x + W_2 \cdot \text{self.middle}$$

incorporating the residual connection. This formulation allows information to flow more effectively through the network, improving gradient propagation and overall training stability. The resulting transformed data is then passed to the subsequent layer or the output layer.

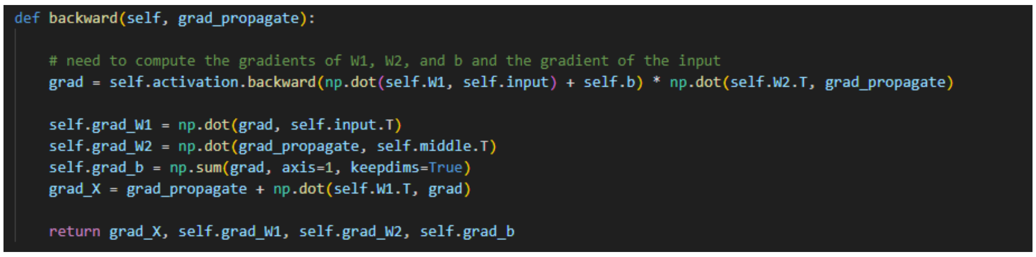


```python
def forward(self, X):
    self.input = X
    self.middle = self.activation.forward(np.dot(self.W1, X) + self.b)
    self.output = X + np.dot(self.W2, self.middle)
    return self.output
```

Layer Backward pass:

The backward pass of the `ResnetLayer` class calculates the gradients of the loss function with respect to the layer's parameters W_1, W_2, b and the input data. These gradients are used for updating the model's parameters through optimization, ensuring the network learns from the data effectively.

Once the gradients are computed, they are propagated backward through the network, allowing preceding layers to update their respective parameters. This backpropagation mechanism is essential for optimizing the entire network and improving its performance during training.



```python
def backward(self, grad_propagate):

    # need to compute the gradients of W1, W2, and b and the gradient of the input
    grad = self.activation.backward(np.dot(self.W1, self.input) + self.b) * np.dot(self.W2.T, grad_propagate)

    self.grad_W1 = np.dot(grad, self.input.T)
    self.grad_W2 = np.dot(grad_propagate, self.middle.T)
    self.grad_b = np.sum(grad, axis=1, keepdims=True)
    grad_X = grad_propagate + np.dot(self.W1.T, grad)

    return grad_X, self.grad_W1, self.grad_W2, self.grad_b
```

2.4.2 Resnet

Network Description:

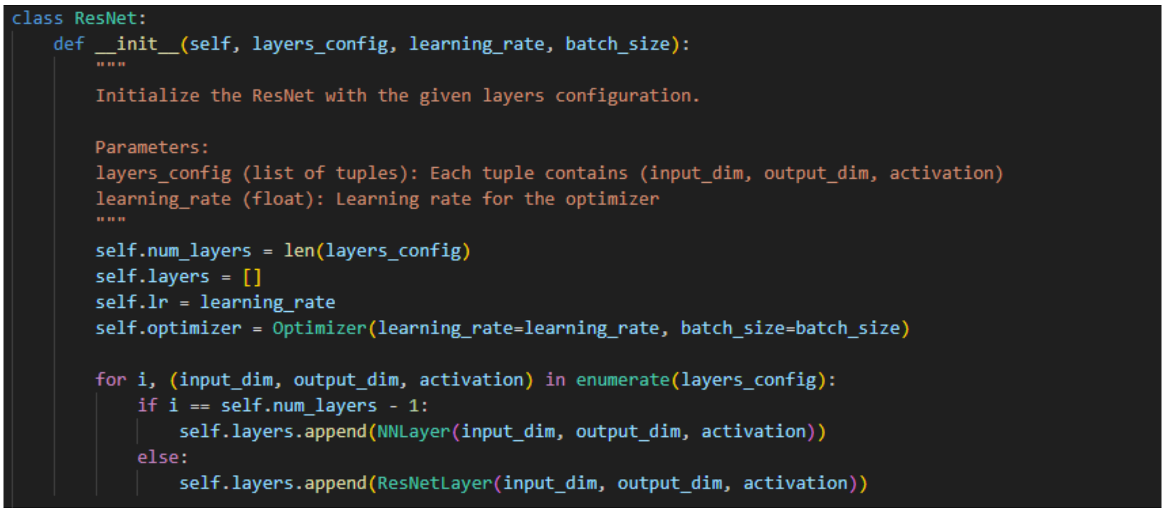
The `ResNet` class is responsible for constructing, training, and evaluating a neural network with residual connections. It supports multiple layers, each configurable with different parameters, and provides functionality for forward and backward propagation, as well as parameter updates.

This class integrates the entire network architecture, ensuring seamless interaction between different layers while leveraging residual connections to enhance learning stability and mitigate gradient vanishing issues in deep networks.

Initialization:

The ResNet class is initialized with the following parameters:

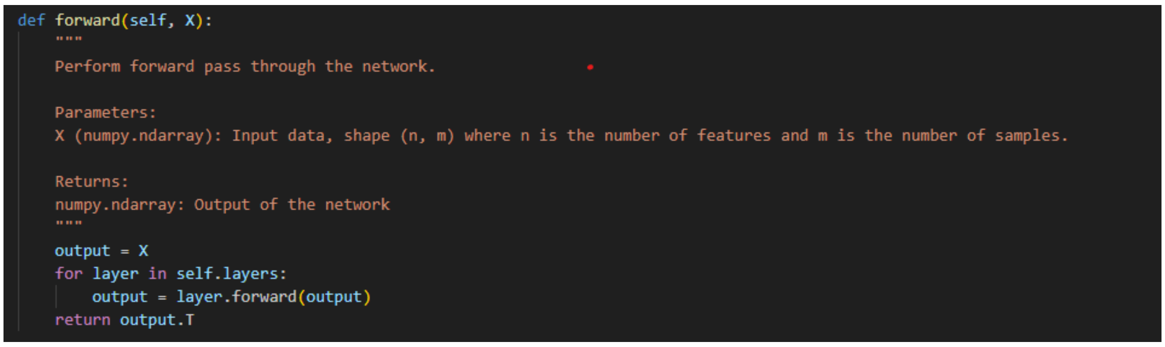
1. *num_layers*: number of layers.
2. *layers*: are initialized from a list of tuples, where each tuple contains the input dimension, output dimension, and activation function for a layer. The final layer is the classifier layer, which is initializes as NNLayer.
3. *lr*: The learning rate for the optimizer.
4. *optimizer*: initialized with the learning rate and batch size



```
class ResNet:
    def __init__(self, layers_config, learning_rate, batch_size):
        """
        Initialize the ResNet with the given layers configuration.

        Parameters:
        layers_config (list of tuples): Each tuple contains (input_dim, output_dim, activation)
        learning_rate (float): Learning rate for the optimizer
        """
        self.num_layers = len(layers_config)
        self.layers = []
        self.lr = learning_rate
        self.optimizer = Optimizer(learning_rate=learning_rate, batch_size=batch_size)

        for i, (input_dim, output_dim, activation) in enumerate(layers_config):
            if i == self.num_layers - 1:
                self.layers.append(NNLayer(input_dim, output_dim, activation))
            else:
                self.layers.append(ResNetLayer(input_dim, output_dim, activation))
```

Network Forward pass:

The forward method executes the forward propagation through the network. It processes the input data sequentially, passing it through each layer while applying the respective transformations. The final layer produces the network's output, which serves as the model's prediction.



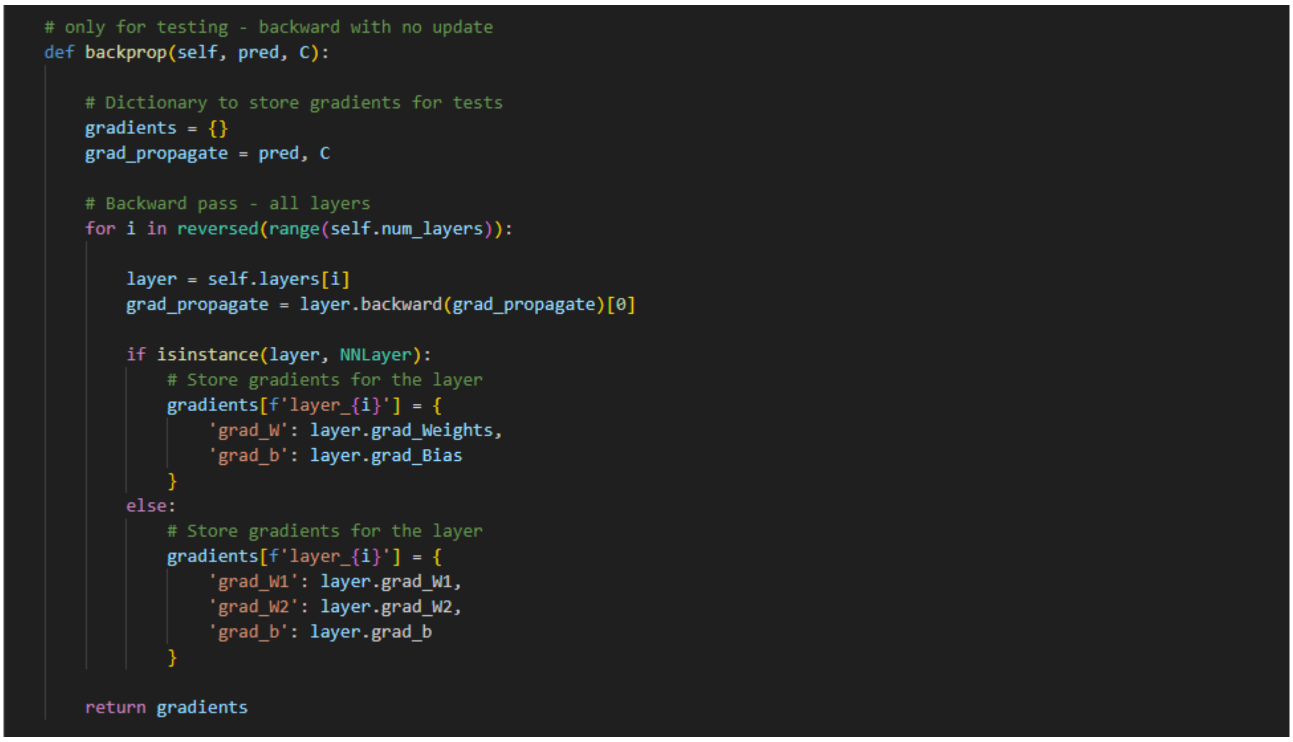
```
def forward(self, X):
    """
    Perform forward pass through the network.

    Parameters:
    X (numpy.ndarray): Input data, shape (n, m) where n is the number of features and m is the number of samples.

    Returns:
    numpy.ndarray: Output of the network
    """
    output = X
    for layer in self.layers:
        output = layer.forward(output)
    return output.T
```

Network Backward pass:

This process and implementation is identical to the backward method in the NeuralNetwork class.

Backprop (for Testing)

The backprop method is similar to the backward method but is used for testing purposes only. It performs the backward pass without updating the parameters and returns the computed gradients for each layer. This method is useful for validating the correctness of the gradient calculations (e.g. gradient test for the whole network).



```python
# only for testing - backward with no update
def backprop(self, pred, C):

    # Dictionary to store gradients for tests
    gradients = {}
    grad_propagate = pred, C

    # Backward pass - all layers
    for i in reversed(range(self.num_layers)):

        layer = self.layers[i]
        grad_propagate = layer.backward(grad_propagate)[0]

        if isinstance(layer, NNLayer):
            # Store gradients for the layer
            gradients[f'layer_{i}'] = {
                'grad_W': layer.grad_Weights,
                'grad_b': layer.grad_Bias
            }
        else:
            # Store gradients for the layer
            gradients[f'layer_{i}'] = {
                'grad_W1': layer.grad_W1,
                'grad_W2': layer.grad_W2,
                'grad_b': layer.grad_b
            }

    return gradients
```

Training Process

The `train` function in the ResNet class is responsible for optimizing the residual network by adjusting its parameters through mini-batch gradient descent. This function trains the network over multiple epochs, computing and updating the model parameters based on the gradients obtained during backpropagation. Additionally, it evaluates performance using accuracy metrics on both the training and validation datasets.

Training Procedure:

1. **Initialization:**
   - The function initializes the loss function and metric tracking dictionaries.
   - The number of training samples is determined.
2. **Shuffling and Mini-batching:**
   - The training data is randomly shuffled at the beginning of each epoch.
   - Mini-batches are created for batch-based stochastic gradient descent.
3. **Mini-batch Training:**
   - For each mini-batch:
     (a) A forward pass is performed through the ResNet model to compute predictions.

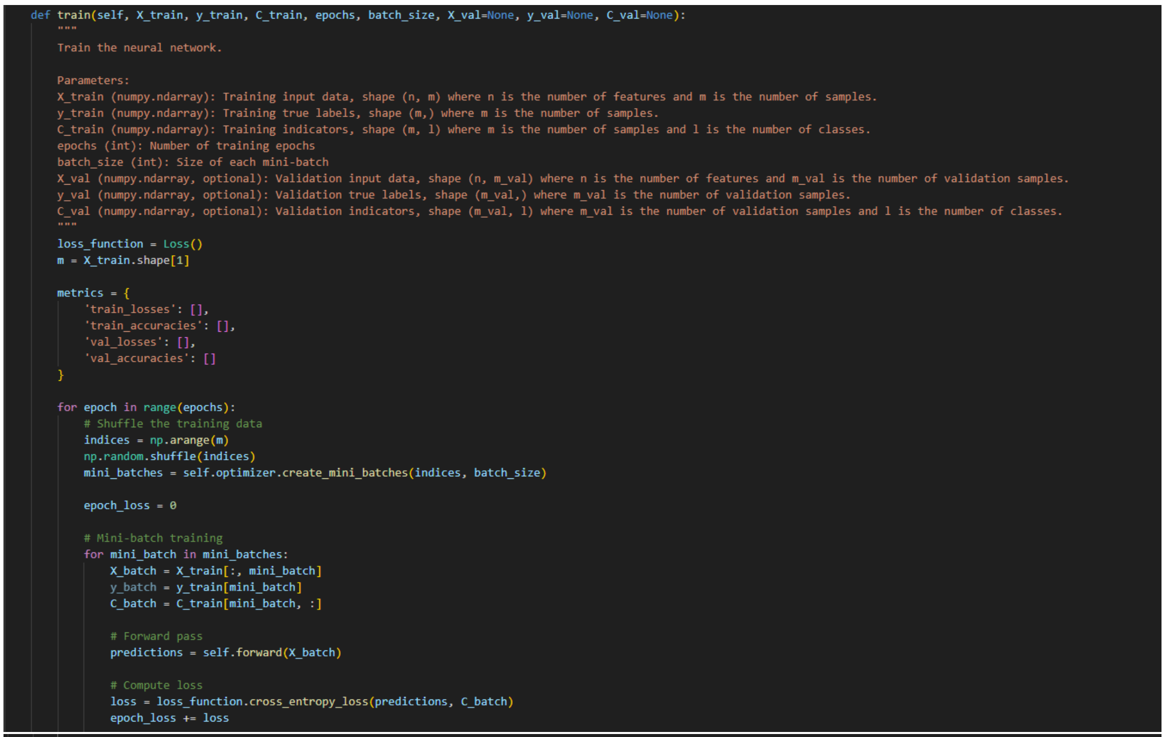
(b) The cross-entropy loss is computed based on predictions and actual labels.
(c) The backward pass propagates errors through the network to compute gradients.
(d) The optimizer updates the model's parameters using the computed gradients.

4. **Performance Evaluation:**
   - The average loss is computed for the epoch.
   - The model's accuracy is evaluated on the entire training set.

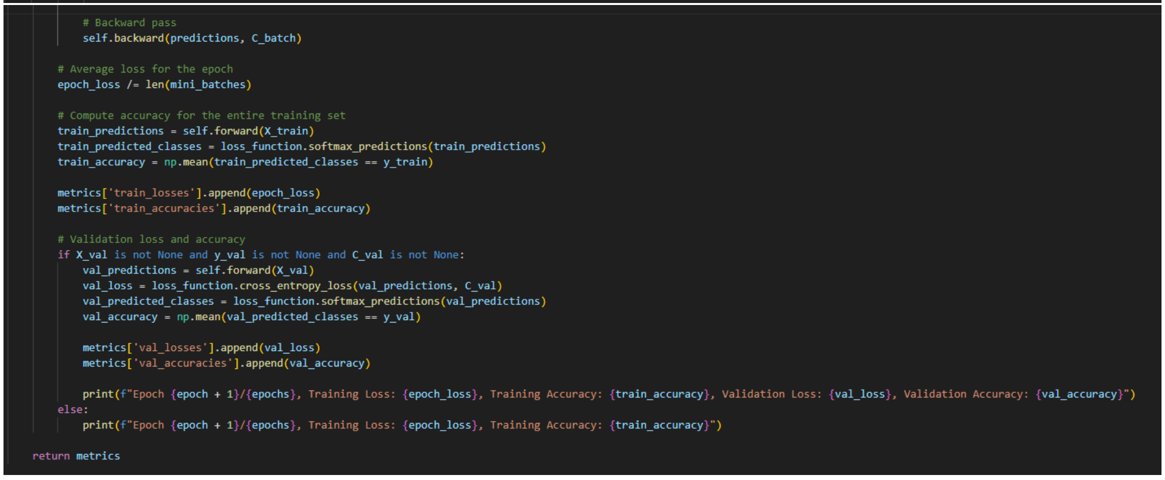
5. **Validation (Optional):**
   - If validation data is provided, the function computes the loss and accuracy on the validation set.

6. **Logging and Output:**
   - The function prints the training loss and accuracy for each epoch.
   - If validation data is available, validation loss and accuracy are also displayed.

By incorporating residual connections, this training loop ensures that deep networks can efficiently propagate gradients, mitigating the vanishing gradient problem. The function systematically updates the model's parameters while monitoring its performance over time.



```python
def train(self, X_train, y_train, C_train, epochs, batch_size, X_val=None, y_val=None, C_val=None):
    """
    Train the neural network.

    Parameters:
    X_train (numpy.ndarray): Training input data, shape (n, m) where n is the number of features and m is the number of samples.
    y_train (numpy.ndarray): Training true labels, shape (m,) where m is the number of samples.
    C_train (numpy.ndarray): Training indicators, shape (m, l) where m is the number of samples and l is the number of classes.
    epochs (int): Number of training epochs
    batch_size (int): Size of each mini-batch
    X_val (numpy.ndarray, optional): Validation input data, shape (n, m_val) where n is the number of features and m_val is the number of validation samples.
    y_val (numpy.ndarray, optional): Validation true labels, shape (m_val,) where m_val is the number of validation samples.
    C_val (numpy.ndarray, optional): Validation indicators, shape (m_val, l) where m_val is the number of validation samples and l is the number of classes.
    """
    loss_function = Loss()
    m = X_train.shape[1]

    metrics = {
        'train_losses': [],
        'train_accuracies': [],
        'val_losses': [],
        'val_accuracies': []
    }

    for epoch in range(epochs):
        # Shuffle the training data
        indices = np.arange(m)
        np.random.shuffle(indices)
        mini_batches = self.optimizer.create_mini_batches(indices, batch_size)

        epoch_loss = 0

        # Mini-batch training
        for mini_batch in mini_batches:
            X_batch = X_train[:, mini_batch]
            y_batch = y_train[mini_batch]
            C_batch = C_train[mini_batch, :]

            # Forward pass
            predictions = self.forward(X_batch)

            # Compute loss
            loss = loss_function.cross_entropy_loss(predictions, C_batch)
            epoch_loss += loss
```

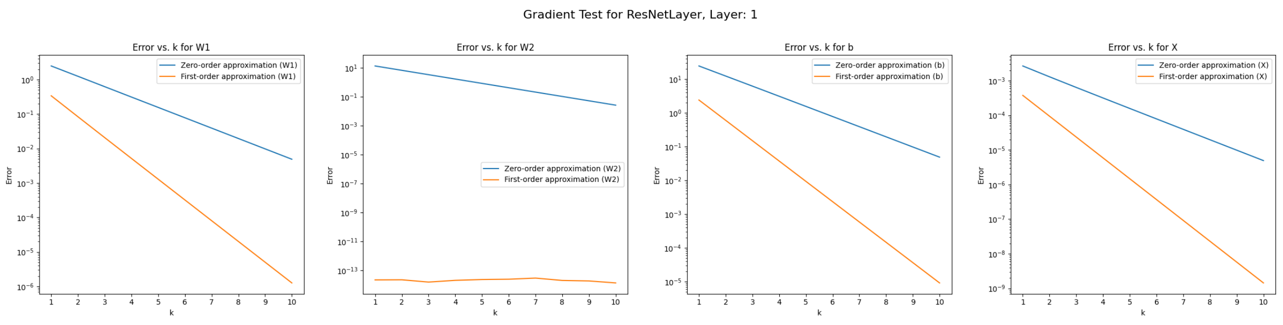


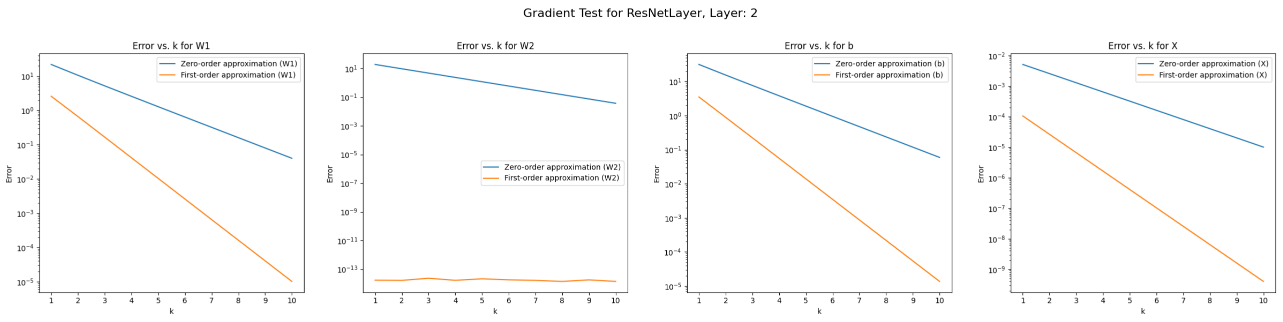
```python
            # Backward pass
            self.backward(predictions, C_batch)

        # Average loss for the epoch
        epoch_loss /= len(mini_batches)

        # Compute accuracy for the entire training set
        train_predictions = self.forward(X_train)
        train_predicted_classes = loss_function.softmax_predictions(train_predictions)
        train_accuracy = np.mean(train_predicted_classes == y_train)

        metrics['train_losses'].append(epoch_loss)
        metrics['train_accuracies'].append(train_accuracy)

        # Validation loss and accuracy
        if X_val is not None and y_val is not None and C_val is not None:
            val_predictions = self.forward(X_val)
            val_loss = loss_function.cross_entropy_loss(val_predictions, C_val)
            val_predicted_classes = loss_function.softmax_predictions(val_predictions)
            val_accuracy = np.mean(val_predicted_classes == y_val)

            metrics['val_losses'].append(val_loss)
            metrics['val_accuracies'].append(val_accuracy)

            print(f"Epoch {epoch + 1}/{epochs}, Training Loss: {epoch_loss}, Training Accuracy: {train_accuracy}, Validation Loss: {val_loss}, Validation Accuracy: {val_accuracy}")
        else:
            print(f"Epoch {epoch + 1}/{epochs}, Training Loss: {epoch_loss}, Training Accuracy: {train_accuracy}")

    return metrics
```

2.4.3 Jacobian test - Layers

Here is a Jacobian test result for "Swiss Roll" dataset with the following ResNet:

$$model = ResNet([(2, 10, 'tanh'), (10, 10, 'tanh'), (10, 2, 'softmax')], 0.01, 32)$$





The Jacobian test results for the ResNet layer exhibit the anticipated behavior across different parameters. The errors associated with X, W_1, and b follow a consistent trend, where both zero-order and first-order errors diminish as the perturbation scale ϵ decreases. This behavior confirms that the computed gradients are accurate.

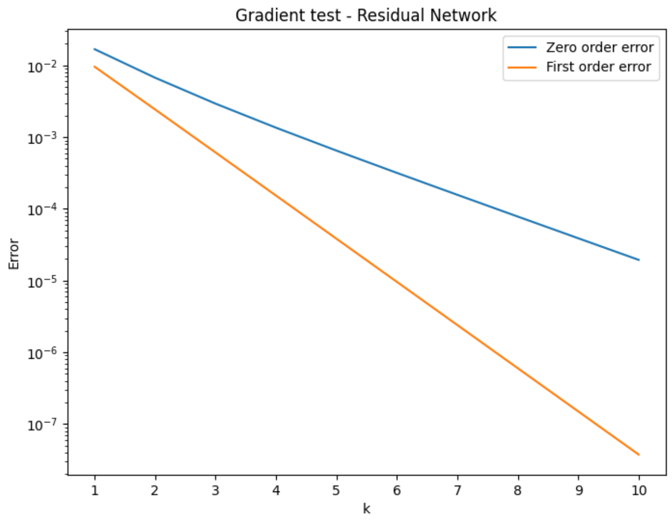
However, a different pattern is observed for W_2, particularly in the first-order error, which is notably small (approximately 10^{-13}, effectively zero in terms of numerical precision). This phenomenon arises due to the structural properties of the layer, where W_2 interacts with $\sigma(W_1X + b)$, causing its gradient to depend indirectly on the preceding transformations.

Since the function is linear with respect to W_2, its gradient remains constant and does not depend on W_2 itself but only on the output of the activation function $\sigma(W_1X + b)$. As a result, small perturbations in W_2 induce minimal immediate effects, leading the first-order error to approach zero. This confirms that the gradient with respect to W_2 is correctly computed and that numerical approximations closely align with analytical derivatives.

These findings affirm that the backpropagation mechanism is correctly implemented for this ResNet configuration. The accuracy of the computed gradients reinforces the validity of the backpropagation process, ensuring reliable optimization and learning stability within the network.

2.4.4 Gradient test - Layers + Classifier

Building upon the previous example, here is the gradient test of the entire residual network:



The graph from the gradient test for the entire network illustrates the declining trends of both zero-order and first-order errors across all layers, including the softmax layer. As the perturbation scale ϵ decreases, both error measures systematically reduce, with the first-order error exhibiting a more rapid decline.

This trend strongly suggests that the computed gradients through backpropagation align well with their numerical approximations. The consistent decrease in error validates the correctness of the gradient computations, reinforcing the reliability of the network's backpropagation mechanism.

Consequently, this ensures that the optimization process remains stable and effective, allowing the model to efficiently update its parameters and improve learning performance over successive training iterations.

2.5 Training

2.5.1 Neural Network Training Analysis

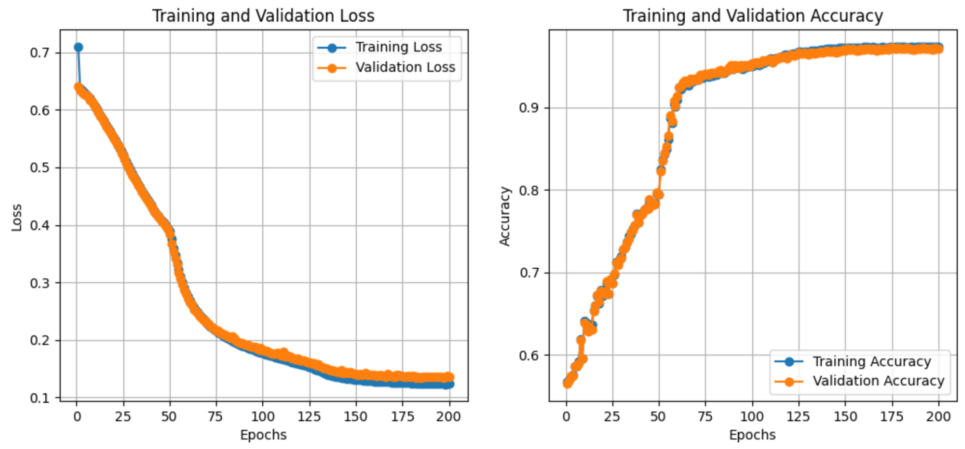
This section presents the evaluation of the neural network model across multiple datasets. The primary aim of these experiments was to assess the network's learning capacity and effectiveness under various conditions. Key hyperparameters such as learning rate, batch size, and the number of training epochs were tuned to optimize performance. The training phase was monitored using loss and accuracy metrics, helping to track the model's ability to generalize to unseen data.

Dataset	LR	Batch	Epochs	Architecture	Train Loss	Val Loss	Train Acc.	Val Acc.
Swiss Roll	0.01	32	200	$2 \to 16 \to 16 \to 2$ (Tanh)	0.132	0.135	97.22%	97.08%
Peaks	0.01	128	200	$2 \to 20 \to 20 \to 5$ (ReLU)	0.278	0.295	93.08%	92.62%
GMM	0.01	256	200	$5 \to 24 \to 5$ (ReLU)	0.249	0.247	95.10%	94.94%

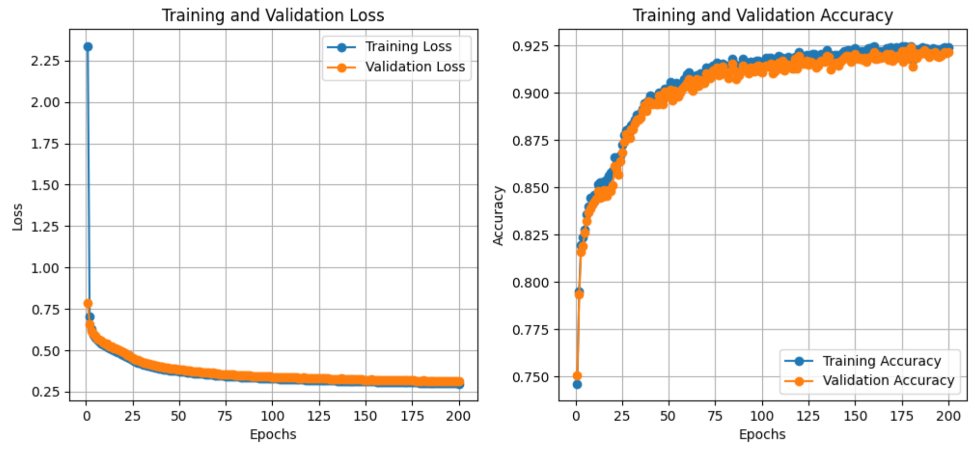
Table 1: Performance analysis of different neural network configurations across datasets.

The following figures illustrate the training progression for different datasets, depicting accuracy trends and loss reduction over time. These visualizations provide insight into how effectively the model adapts to various learning scenarios.

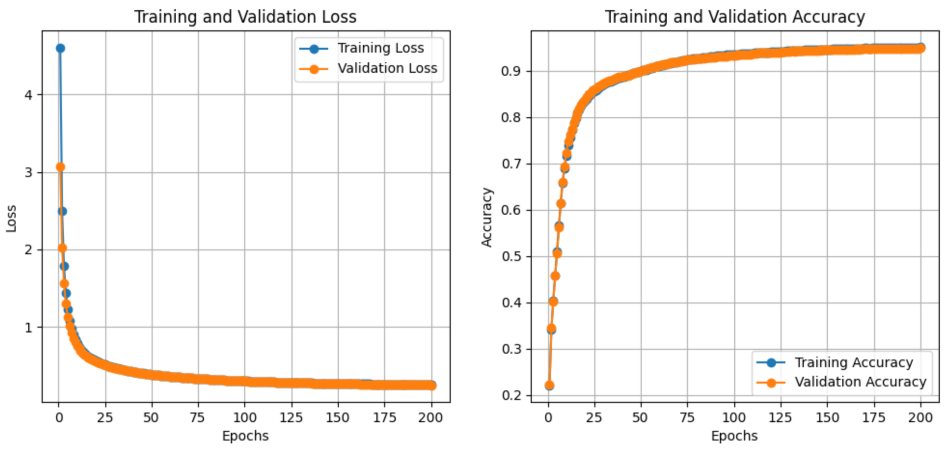
Swiss Roll:



Peaks:



GMM:



The experimental results demonstrate that the chosen neural network configurations effectively learned patterns in the data while maintaining stable performance across different datasets.

For the Swiss Roll dataset, the network consistently achieved high accuracy with minimal training and validation loss, indicating its capability to model the underlying structure of the data effectively.

The Peaks dataset presented a greater challenge due to its complexity, yet the model was able to generalize well, achieving strong classification accuracy despite slightly elevated loss values.

In the case of the GMM dataset, the network exhibited particularly strong performance, maintaining low loss and high accuracy, suggesting that the architecture adapted well to the distribution of the data.

Observing the accuracy curves, a clear upward trend is visible in both training and validation accuracy, reflecting the model's ability to refine its predictions over time. Concurrently, the loss curves exhibit a smooth and consistent decline, showcasing the efficiency of the optimization process in reducing classification errors. These findings confirm that the selected hyperparameters and network structure provided a well-balanced approach to learning, ensuring effective training and reliable performance across varying datasets.

2.5.2 Residual Neural Network Training Analysis

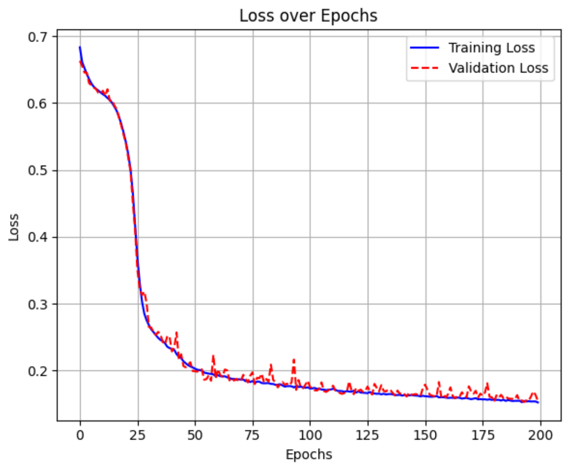
This section explores the training performance of the ResNet model, focusing on its behavior across different hyperparameter configurations. The aim is to examine how variations in training settings impact learning efficiency and generalization capability.

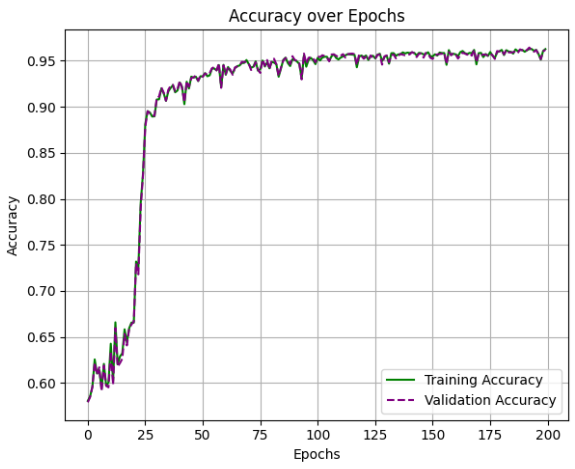
Dataset	LR	Batch	Epochs	Architecture	Train Loss	Val Loss	Train Acc.	Val Acc.
Swiss Roll	0.01	32	200	$2 \to 16 \to 16 \to 2$ (Tanh)	0.152	0.154	96.27%	96.20%
Peaks	0.01	128	200	$2 \to 20 \to 20 \to 5$ (ReLU)	0.309	0.344	90.9%	90.08%
GMM	0.01	256	200	$5 \to 24 \to 5$ (ReLU)	0.236	0.24	95.58%	95.86%

Table 2: Performance analysis of different neural network configurations across datasets.

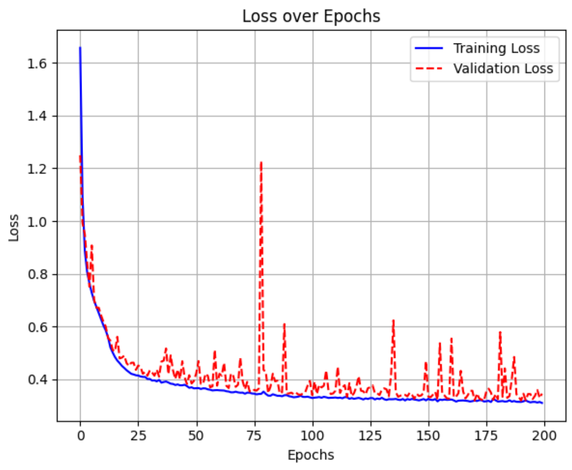
The following figures illustrate the training progression for different datasets, depicting accuracy trends and loss reduction over time. These visualizations provide insight into how effectively the model adapts to various learning scenarios.

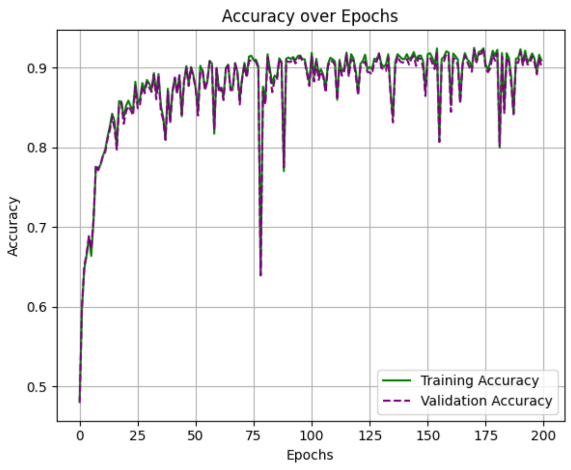
Swiss Roll:



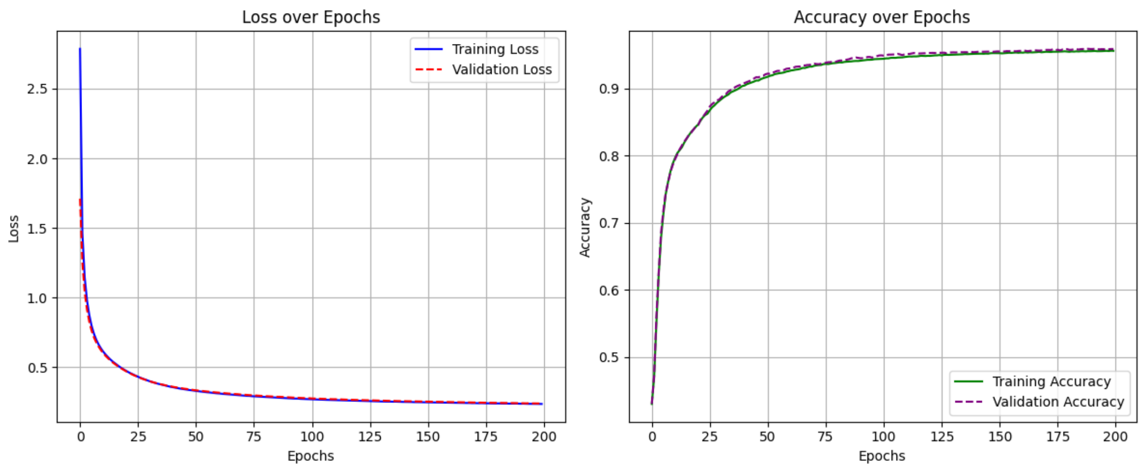


Peaks:





GMM:



The ResNet architecture demonstrated strong performance across all datasets, successfully adapting to different data distributions. The results indicate that appropriate hyperparameter selection is crucial for achieving stable convergence. The Swiss Roll and GMM datasets exhibited smooth learning curves, whereas Peaks required careful tuning to manage fluctuations. Overall, the experiment highlights the effectiveness of residual connections in deep networks, ensuring stable gradient propagation and improved classification accuracy.

2.5.3 Effect of Network Depth

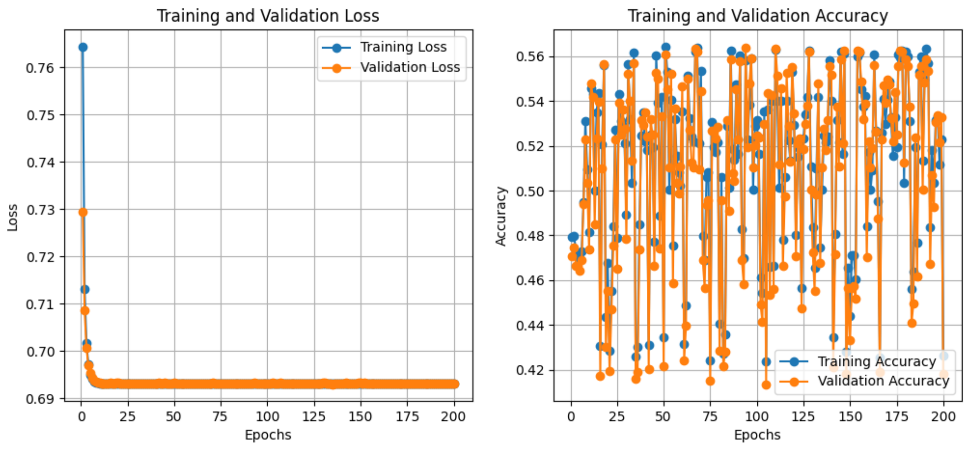
To investigate how network depth influences model performance, we conducted an experiment on the Swiss Roll dataset using the NeuralNetwork framework. The objective was to evaluate how varying the number of hidden layers affects the network's ability to learn, converge, and maintain stability during training. By systematically modifying the depth of the model, we aimed to analyze its impact on accuracy and generalization.

For this experiment, we utilized the following hyperparameters: tanh as the activation function, a learning rate of 0.01, a batch size of 64, and a total of 200 training epochs.

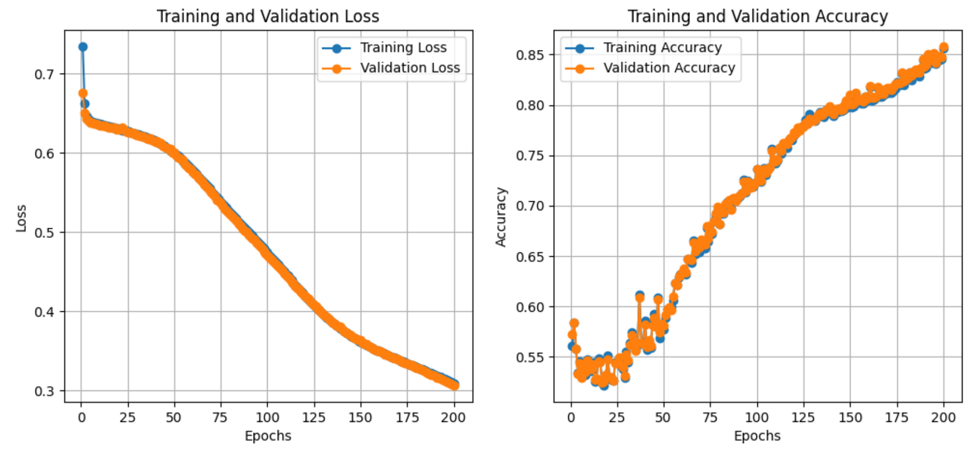
Network Depth	Architecture	Train Loss	Val Loss	Train Acc.	Val Acc.
0 Hidden Layer	$2 \to 5$	0.69	0.69	55.14%	55.70%
2 Hidden Layer	$2 \to 20 \to 5$	0.396	0.395	79.21%	79.00%
4 Hidden Layers	$2 \to 20 \to 20 \to 5$	0.116	0.128	97.39%	97.24%
6 Hidden Layers	$2 \to 20 \to 20 \to 20 \to 20 \to 20 \to 20 \to 5$	0.111	0.122	97.39%	97.06%
8 Hidden Layers	$2 \to 20 \to 20 \to 20 \to 20 \to 20 \to 20 \to 20 \to 20 \to 5$	0.119	0.148	96.14%	95.86%

The following figures illustrate the training progression for different datasets, depicting accuracy trends and loss reduction over time. These visualizations provide insight into how effectively the model adapts to various learning scenarios.

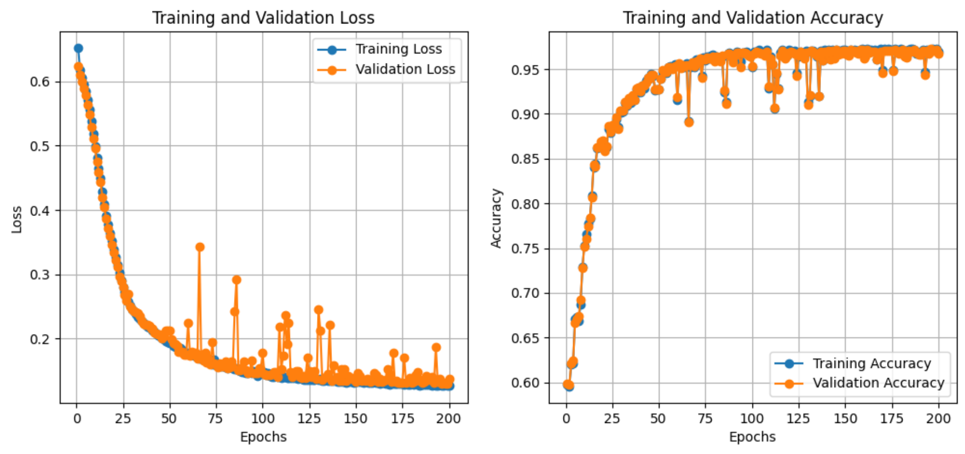
0 Hidden Layer:



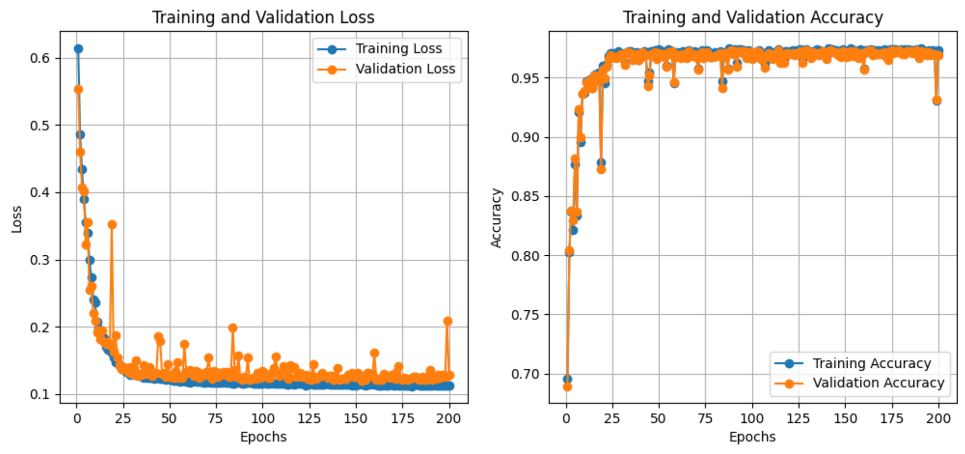
2 Hidden Layer:



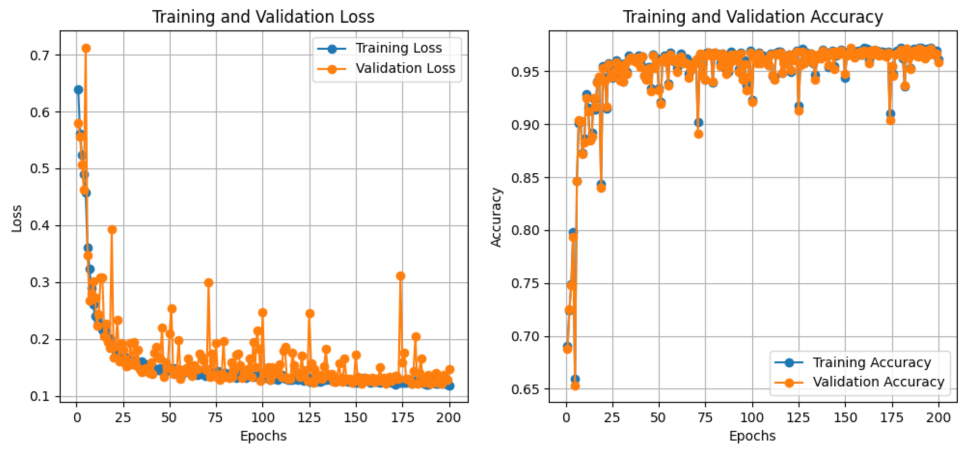
4 Hidden Layers:



6 Hidden Layers:



8 Hidden Layers:



Summary

This experiment examines how increasing the depth of a neural network influences its ability to learn, converge, and generalize on the **Swiss Roll** dataset. We tested architectures with different numbers of hidden layers, ranging from 0 to 8, while keeping the hyperparameters constant: **Tanh activation, learning rate of 0.01, batch size of 64, and 200 training epochs**.

The results demonstrate that **shallow networks (0-2 hidden layers) struggled to fully capture complex patterns**, resulting in lower accuracy and higher loss values. Networks with **4 and 6 hidden layers** achieved strong convergence, leading to high accuracy with minimal loss. However, at **8 layers, a slight drop in validation accuracy** was observed, suggesting potential overfitting or diminishing returns from additional depth.

From the loss curves, we observe that deeper networks (4, 6, and 8 layers) **converge faster and reach lower training loss**, whereas shallower networks (0 and 2 layers) exhibit slower convergence and instability in validation accuracy. The fluctuations in validation accuracy and loss for deeper architectures suggest **increased sensitivity to training dynamics**, though overall performance remains strong.

Conclusion

- **Shallow networks (0-2 layers) lack sufficient learning capacity**, showing lower validation accuracy and higher loss values, indicating difficulty in modeling complex decision boundaries.
- **Optimal performance is achieved with 4-6 hidden layers**, balancing expressive power and stability. These networks effectively capture non-linear structures while maintaining strong generalization.
- **Excessive depth (8 layers) leads to minor performance degradation**, likely due to overfitting or vanishing gradient issues, suggesting that increasing depth beyond a certain point does not necessarily improve performance.
- **Validation accuracy trends confirm that deeper models benefit from improved feature extraction**, but excessive depth introduces instability, highlighting the need for an optimal depth range.

Overall, we conclude that **a network depth of 4-6 hidden layers provides the best trade-off between accuracy, convergence speed, and stability** for this dataset.

2.6 Limiting the Network parameters

Understanding the trade-off between model complexity and efficiency is crucial in deep learning. To explore this, we trained neural networks under a strict limit of $100C$ trainable parameters, where C is the number of output classes. The goal was to maintain high accuracy while minimizing the number of parameters, ensuring a compact yet effective model.

2.6.1 Estimating Trainable Parameters

In a standard fully connected neural network, the total number of trainable parameters—comprising weights and biases—is determined by:

$$\sum_{l=1}^{L}(n_{l-1} \cdot n_l + n_l)$$

where:

- L is the total number of layers (excluding the input layer).
- n_0 represents the number of input features.
- n_l denotes the number of neurons in each layer.
- Each layer contributes both weight parameters $(h_{l-1} \times h_l)$ and bias parameters (h_l).

By controlling the network's architecture, we ensured that no model exceeded the defined limit.

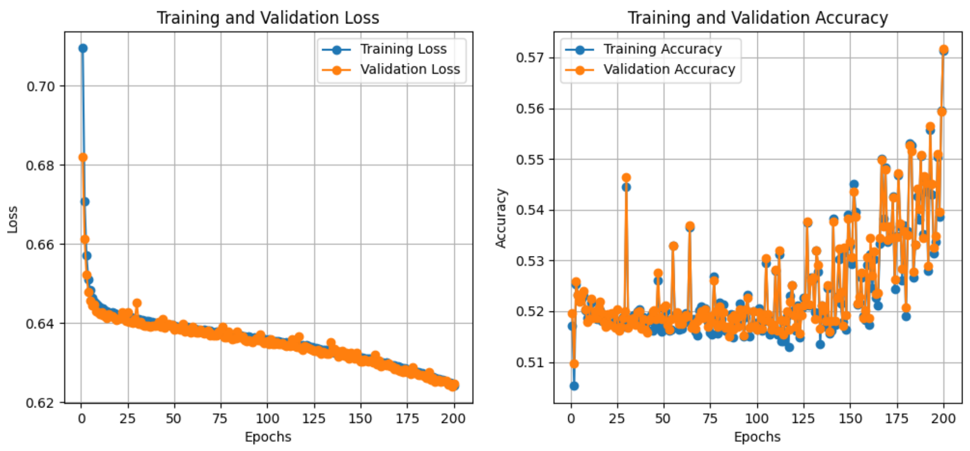
2.6.2 Experimental Design

This study aimed to assess how depth affects learning efficiency while keeping the number of parameters constant. Using the **Swiss Roll dataset**, we tested various architectures, modifying the number of hidden layers while adhering to the **200-parameter budget** ($100C = 200$, with $C = 2$ output classes).

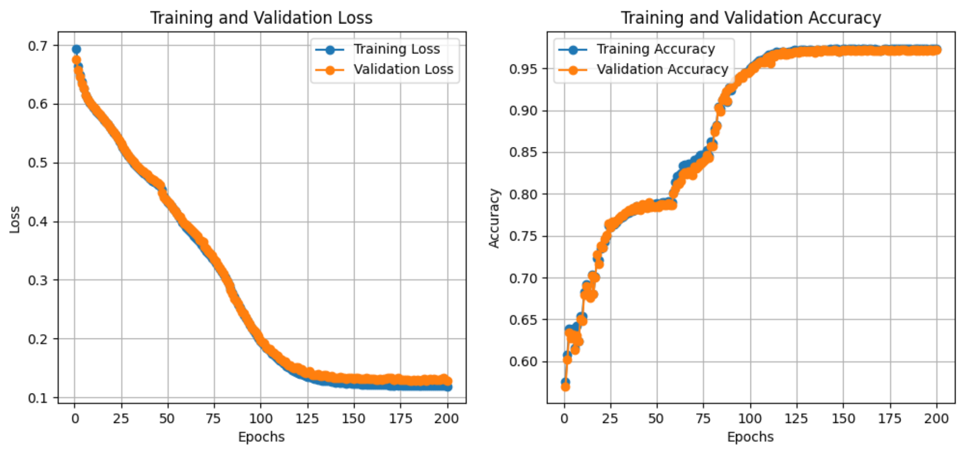
Each model was trained for **200 epochs**, with a **learning rate of 0.01**, a **batch size of 64**, and **tanh activation**. By distributing the available parameters across different depths, we analyzed how architectural choices influence training stability, convergence speed, and overall accuracy.

Network Depth	Architecture	Params	Train Loss	Val Loss	Train Acc.	Val Acc.
1 Hidden Layer	$2 \to 25 \to 2$	127	0.624	0.624	57.14%	57.18%
3 Hidden Layers	$2 \to 10 \to 12 \to 7 \to 2$	175	0.119	0.128	97.38%	97.18%
5 Hidden Layers	$2 \to 7 \to 8 \to 8 \to 6 \to 2$	196	0.118	0.131	96.93%	96.84%
7 Hidden Layers	$2 \to 6 \to 6 \to 5 \to 5 \to 4 \to 3 \to 2$	197	0.148	0.322	90.03%	89.14%
9 Hidden Layers	$2 \to 5 \to 5 \to 4 \to 4 \to 3 \to 3 \to 3 \to\to 3 \to 3 \to 2$	199	0.633	0.633	61.70%	62.84%

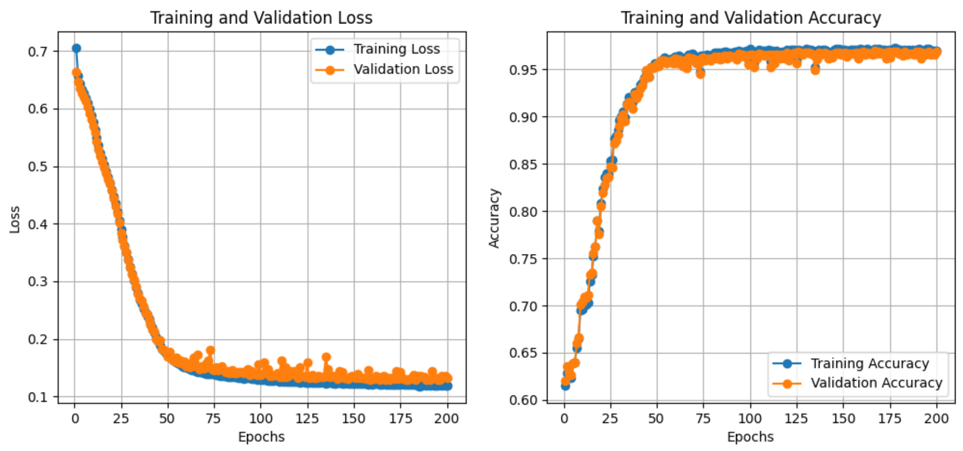
1 hidden layer:



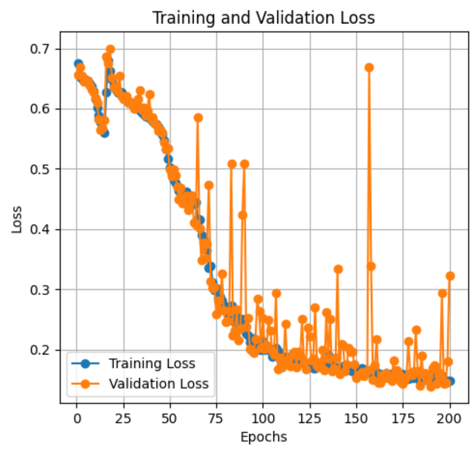
3 hidden layers:

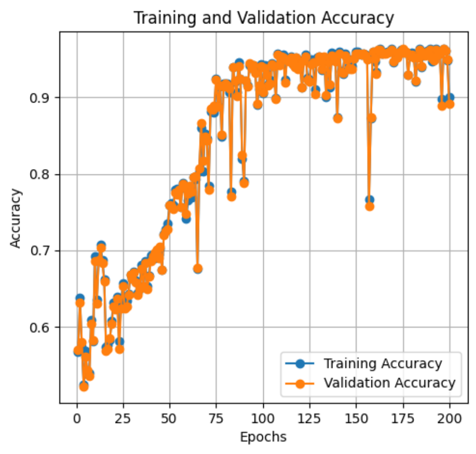


5 hidden layers:

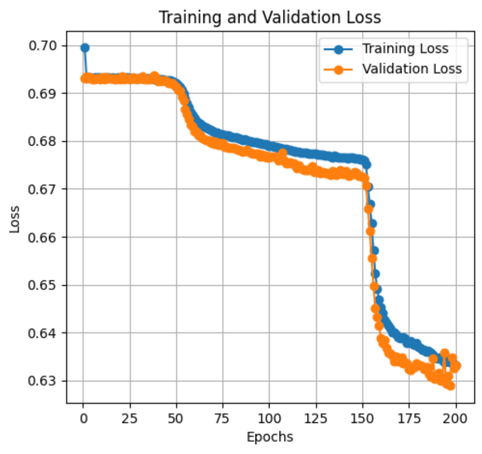


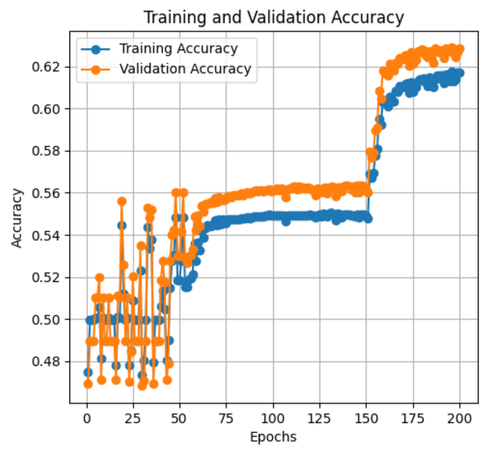
7 hidden layers:





9 hidden layers:





2.6.3 Summary of Results

This study investigated the effect of network depth on model performance while maintaining a strict parameter budget. The experiments were conducted using the Swiss Roll dataset, with different architectures ranging from 1 to 9 hidden layers. Each network was trained with a learning rate of 0.01, a batch size of 64, and the tanh activation function.

The key findings from the experiment are as follows:

- **1 Hidden Layer:** The shallowest network struggled to capture complex data patterns, resulting in relatively poor accuracy and higher loss values.
- **3-5 Hidden Layers:** These architectures showed significant improvements in training and validation accuracy, achieving the best generalization with validation accuracy reaching approximately 97%.
- **7 Hidden Layers:** Performance was slightly degraded compared to the optimal range (3-5 layers), as training became less stable.
- **9 Hidden Layers:** The deepest network exhibited signs of instability, with fluctuating accuracy and loss values. The validation accuracy remained lower, indicating difficulties in effective learning.

2.6.4 Conclusion

The experiment demonstrates that increasing network depth improves performance up to a certain point, beyond which additional layers can lead to diminishing returns or even degradation in accuracy. The optimal depth for this dataset, under the given parameter constraints, appears to be within the range of 3 to 5 hidden layers.

Key takeaways:

- **Shallow networks** (1 hidden layer) lack sufficient capacity to model complex relationships in the data.
- **Moderate-depth networks** (3-5 hidden layers) offer the best trade-off between performance and stability, achieving high accuracy with minimal overfitting.
- **Deeper networks** (7+ hidden layers) show increased instability, and in the case of 9 layers, training difficulties become evident.
- **Parameter efficiency** is crucial; distributing available parameters effectively across layers is key to maximizing accuracy while maintaining stability.

In conclusion, while deeper architectures have the potential to learn complex patterns, excessive depth under a limited parameter budget can lead to instability and reduced generalization. Future work could explore alternative regularization techniques or different activation functions to mitigate these challenges.