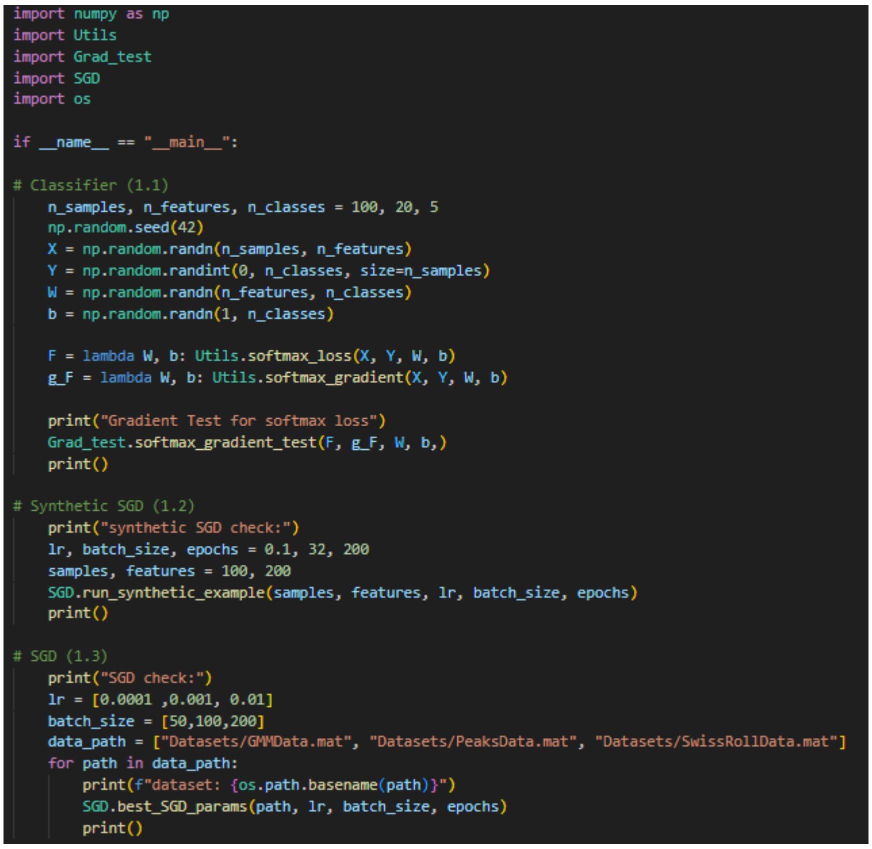
Mini-project: Deep Learning from Scratch

Submitted by Lior Sharony, Eden Miran, Tomer Noama

1. Part I: the classifier and optimizer

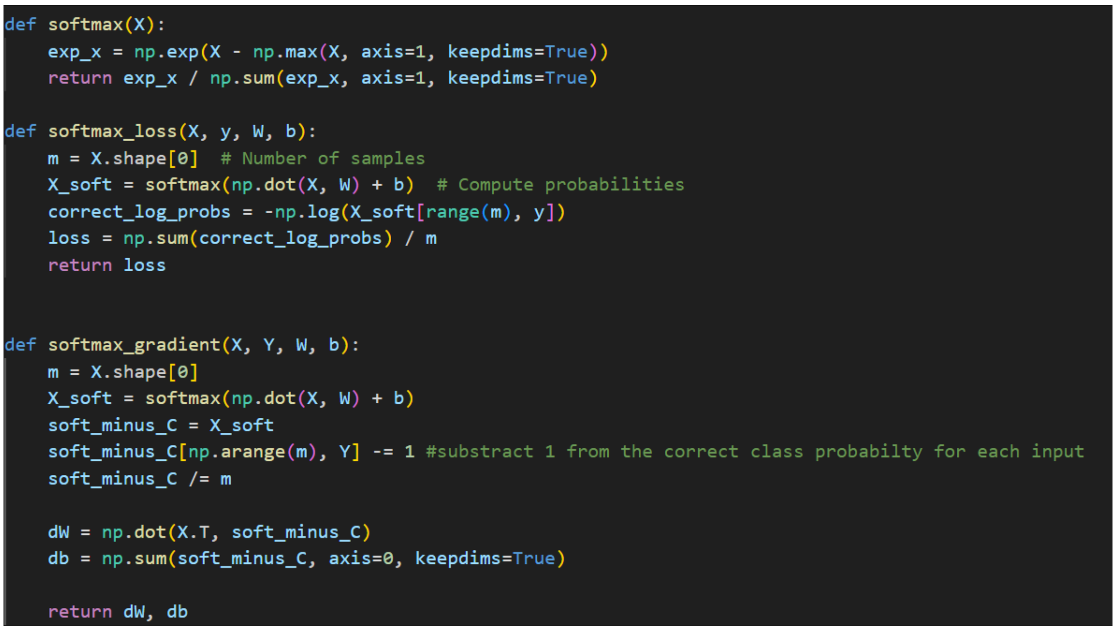


```python
import numpy as np
import Utils
import Grad_test
import SGD
import os

if __name__ == "__main__":

# Classifier (1.1)
    n_samples, n_features, n_classes = 100, 20, 5
    np.random.seed(42)
    X = np.random.randn(n_samples, n_features)
    Y = np.random.randint(0, n_classes, size=n_samples)
    W = np.random.randn(n_features, n_classes)
    b = np.random.randn(1, n_classes)

    F = lambda W, b: Utils.softmax_loss(X, Y, W, b)
    g_F = lambda W, b: Utils.softmax_gradient(X, Y, W, b)

    print("Gradient Test for softmax loss")
    Grad_test.softmax_gradient_test(F, g_F, W, b,)
    print()

# Synthetic SGD (1.2)
    print("synthetic SGD check:")
    lr, batch_size, epochs = 0.1, 32, 200
    samples, features = 100, 200
    SGD.run_synthetic_example(samples, features, lr, batch_size, epochs)
    print()

# SGD (1.3)
    print("SGD check:")
    lr = [0.0001 ,0.001, 0.01]
    batch_size = [50,100,200]
    data_path = ["Datasets/GMMData.mat", "Datasets/PeaksData.mat", "Datasets/SwissRollData.mat"]
    for path in data_path:
        print(f"dataset: {os.path.basename(path)}")
        SGD.best_SGD_params(path, lr, batch_size, epochs)
        print()
```

Fig 1: The main function of part 1

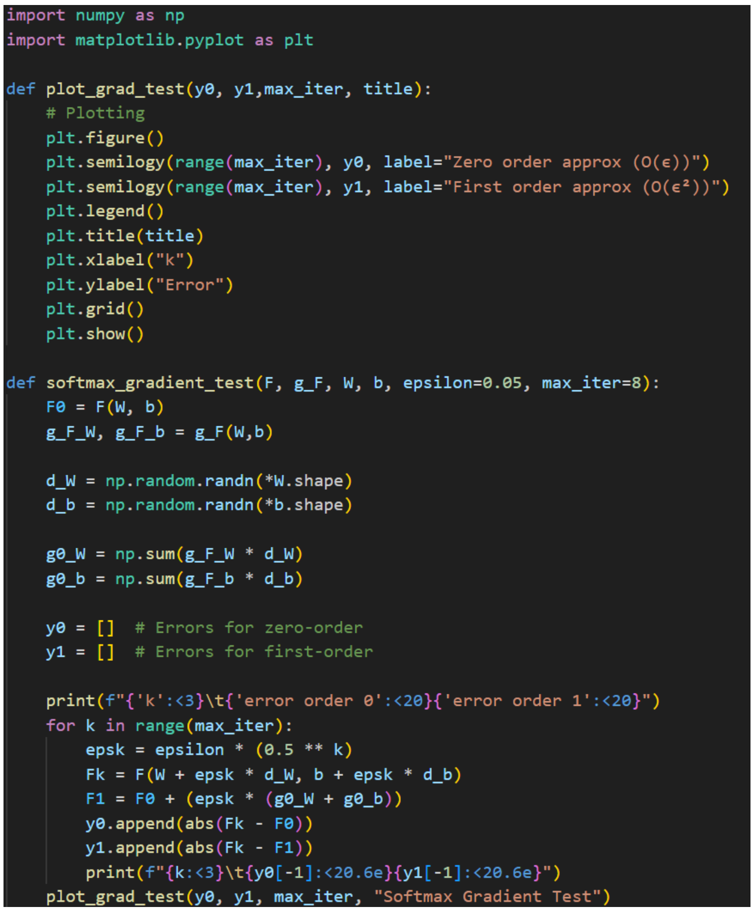
1.1. loss function “soft-max regression” and its gradient

We have tested the correctness of our soft-max regression gradient using the gradient test as shown in the class, with respect to the weights and biases, using 8 iteration and an epsilon value of 0.5 (shown in the code).



```python
def softmax(X):
    exp_x = np.exp(X - np.max(X, axis=1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)

def softmax_loss(X, y, W, b):
    m = X.shape[0]  # Number of samples
    X_soft = softmax(np.dot(X, W) + b)  # Compute probabilities
    correct_log_probs = -np.log(X_soft[range(m), y])
    loss = np.sum(correct_log_probs) / m
    return loss


def softmax_gradient(X, Y, W, b):
    m = X.shape[0]
    X_soft = softmax(np.dot(X, W) + b)
    soft_minus_C = X_soft
    soft_minus_C[np.arange(m), Y] -= 1 #substract 1 from the correct class probabilty for each input
    soft_minus_C /= m

    dW = np.dot(X.T, soft_minus_C)
    db = np.sum(soft_minus_C, axis=0, keepdims=True)

    return dW, db
```

Fig 2: Softmax function, loss and gradient



```python
import numpy as np
import matplotlib.pyplot as plt

def plot_grad_test(y0, y1,max_iter, title):
    # Plotting
    plt.figure()
    plt.semilogy(range(max_iter), y0, label="Zero order approx (O(ϵ))")
    plt.semilogy(range(max_iter), y1, label="First order approx (O(ϵ²))")
    plt.legend()
    plt.title(title)
    plt.xlabel("k")
    plt.ylabel("Error")
    plt.grid()
    plt.show()

def softmax_gradient_test(F, g_F, W, b, epsilon=0.05, max_iter=8):
    F0 = F(W, b)
    g_F_W, g_F_b = g_F(W,b)

    d_W = np.random.randn(*W.shape)
    d_b = np.random.randn(*b.shape)

    g0_W = np.sum(g_F_W * d_W)
    g0_b = np.sum(g_F_b * d_b)

    y0 = []  # Errors for zero-order
    y1 = []  # Errors for first-order

    print(f"{'k':<3}\t{'error order 0':<20}{'error order 1':<20}")
    for k in range(max_iter):
        epsk = epsilon * (0.5 ** k)
        Fk = F(W + epsk * d_W, b + epsk * d_b)
        F1 = F0 + (epsk * (g0_W + g0_b))
        y0.append(abs(Fk - F0))
        y1.append(abs(Fk - F1))
        print(f"{k:<3}\t{y0[-1]:<20.6e}{y1[-1]:<20.6e}")
    plot_grad_test(y0, y1, max_iter, "Softmax Gradient Test")
```

Fig 3: The gradient test code

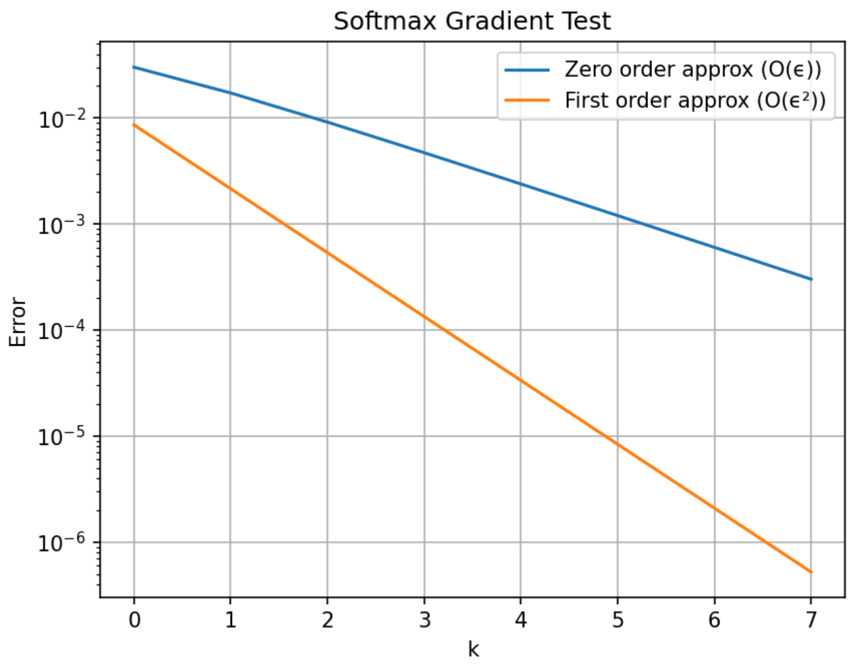
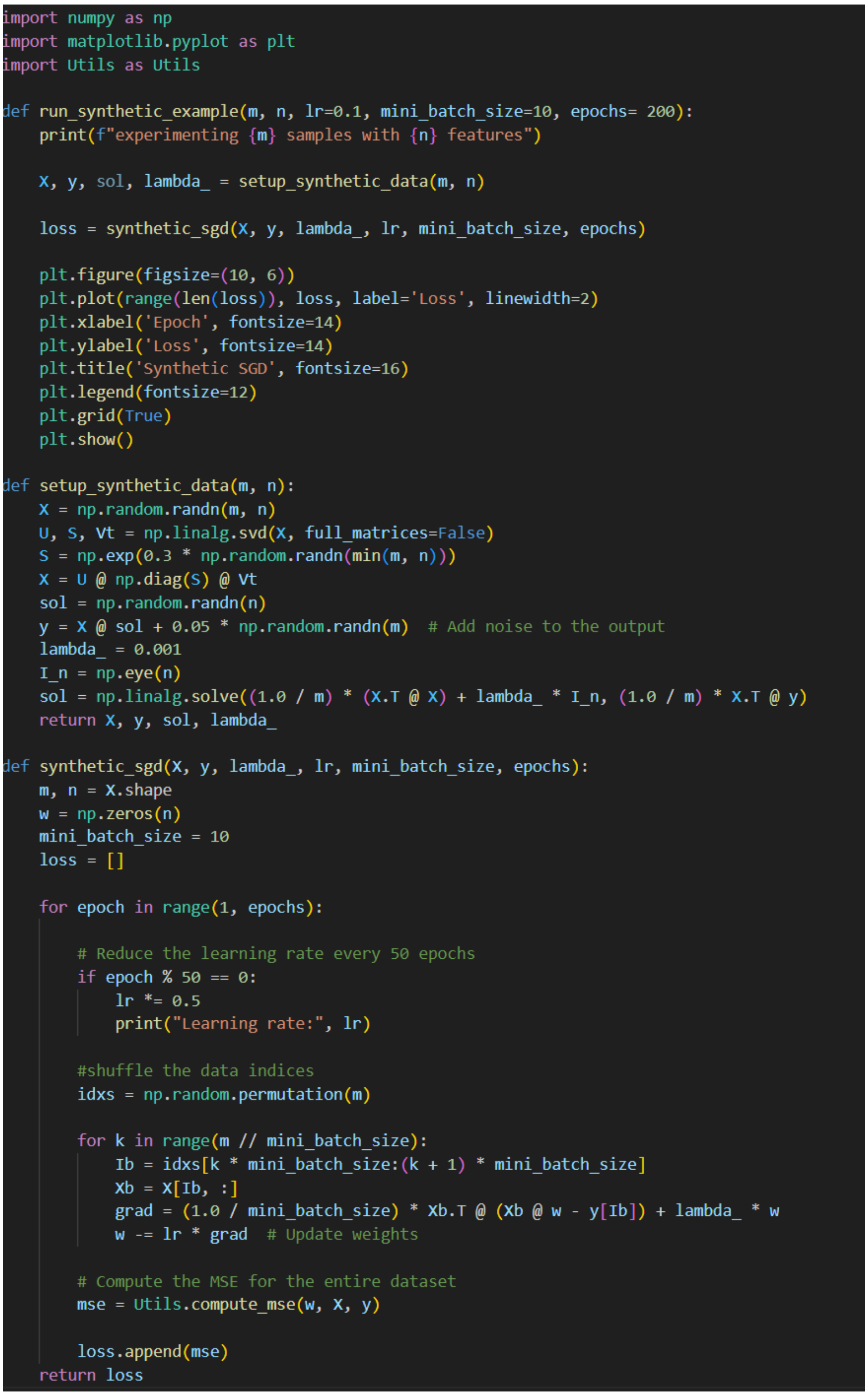


Fig 4: Softmax gradient plot

We can see in the result that the graph is linear with different slopes, so that the zero order decreases linearly (on a semilogarithmic scale) while the first order converges quadratically (on the same scale).

1.2. Synthetic SGD

We have implemented the SGD and tested it on a small synthetic data (as shown below), We implemented the data setup method as shown in the notes, the data we produced consisted of 100 samples with 200 features and the loss was calculated using mse.
The SGD learning rate was reduced by half every 50 epochs.



```python
import numpy as np
import matplotlib.pyplot as plt
import Utils as Utils

def run_synthetic_example(m, n, lr=0.1, mini_batch_size=10, epochs= 200):
    print(f"experimenting {m} samples with {n} features")

    X, y, sol, lambda_ = setup_synthetic_data(m, n)

    loss = synthetic_sgd(X, y, lambda_, lr, mini_batch_size, epochs)

    plt.figure(figsize=(10, 6))
    plt.plot(range(len(loss)), loss, label='Loss', linewidth=2)
    plt.xlabel('Epoch', fontsize=14)
    plt.ylabel('Loss', fontsize=14)
    plt.title('Synthetic SGD', fontsize=16)
    plt.legend(fontsize=12)
    plt.grid(True)
    plt.show()

def setup_synthetic_data(m, n):
    X = np.random.randn(m, n)
    U, S, Vt = np.linalg.svd(X, full_matrices=False)
    S = np.exp(0.3 * np.random.randn(min(m, n)))
    X = U @ np.diag(S) @ Vt
    sol = np.random.randn(n)
    y = X @ sol + 0.05 * np.random.randn(m)  # Add noise to the output
    lambda_ = 0.001
    I_n = np.eye(n)
    sol = np.linalg.solve((1.0 / m) * (X.T @ X) + lambda_ * I_n, (1.0 / m) * X.T @ y)
    return X, y, sol, lambda_

def synthetic_sgd(X, y, lambda_, lr, mini_batch_size, epochs):
    m, n = X.shape
    w = np.zeros(n)
    mini_batch_size = 10
    loss = []

    for epoch in range(1, epochs):

        # Reduce the learning rate every 50 epochs
        if epoch % 50 == 0:
            lr *= 0.5
            print("Learning rate:", lr)

        #shuffle the data indices
        idxs = np.random.permutation(m)

        for k in range(m // mini_batch_size):
            Ib = idxs[k * mini_batch_size:(k + 1) * mini_batch_size]
            Xb = X[Ib, :]
            grad = (1.0 / mini_batch_size) * Xb.T @ (Xb @ w - y[Ib]) + lambda_ * w
            w -= lr * grad  # Update weights

        # Compute the MSE for the entire dataset
        mse = Utils.compute_mse(w, X, y)

        loss.append(mse)
    return loss
```

Fig 5: The synthetic SGD and the data setup

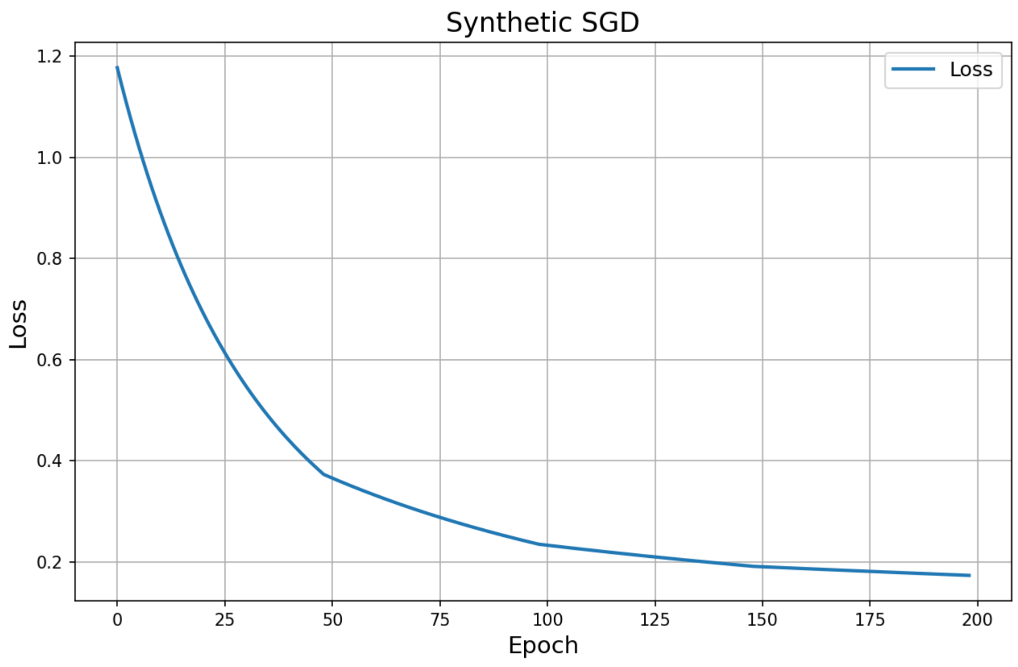


Fig 6: The Synthetic SGD loss plot

*We can see that the SGD loss converges around 0.17.

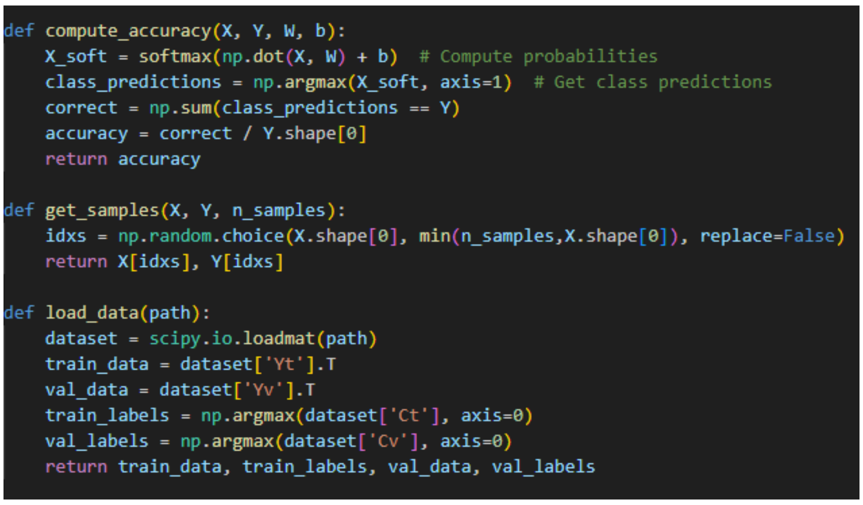
1.3. Softmax SGD

We've tried the following parameters:

- learning rates: [0.0001, 0.001, 0.01]
- mini-batch sizes: [50, 100, 200]

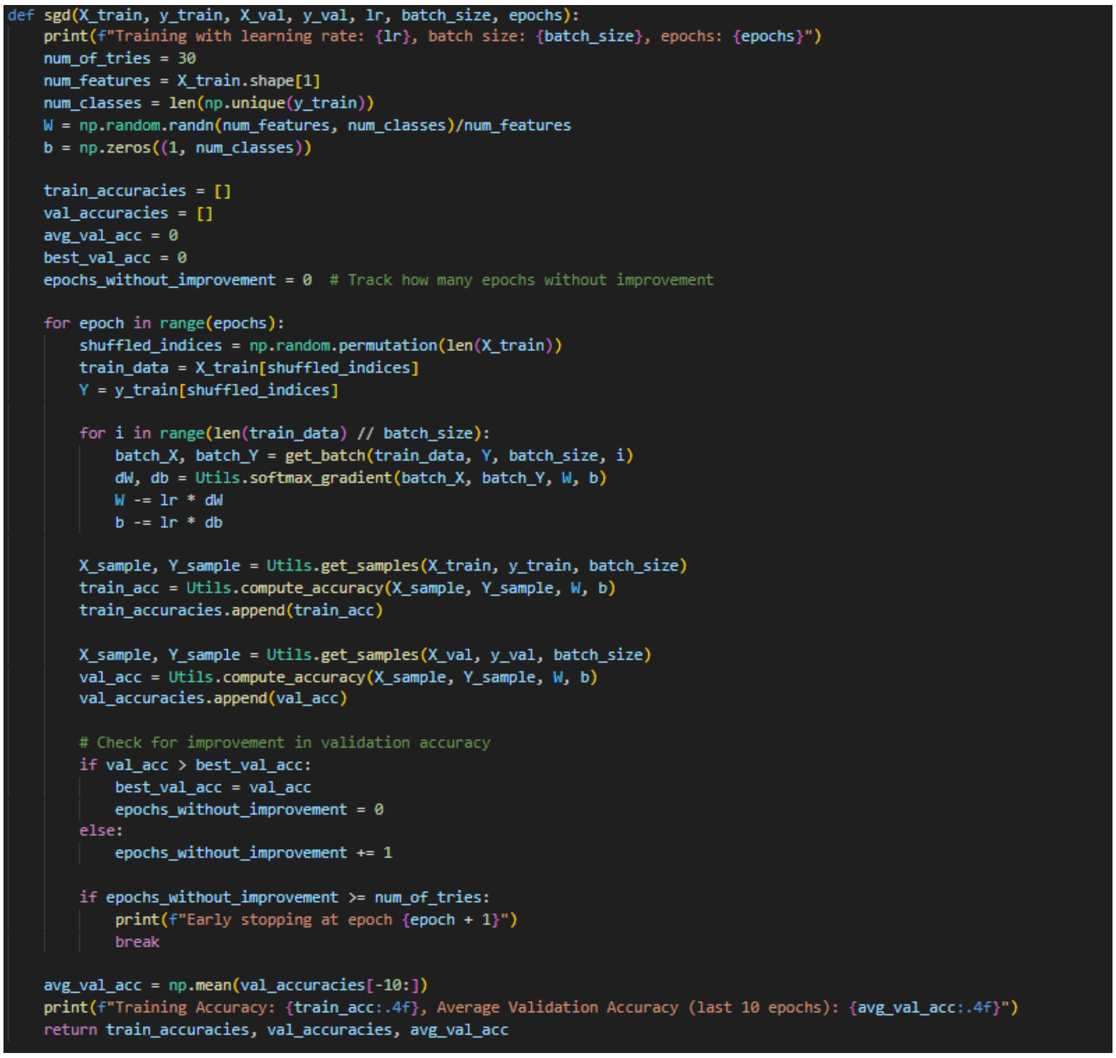
For each data set we saved the best validation accuracy along with its parameters and the plot referring to that.

*We have implemented in our code a mechanism which breaks the current SGD run whenever the validation accuracy failed to improve after 30 consecutive epochs.



```
def compute_accuracy(X, Y, W, b):
    X_soft = softmax(np.dot(X, W) + b)  # Compute probabilities
    class_predictions = np.argmax(X_soft, axis=1)  # Get class predictions
    correct = np.sum(class_predictions == Y)
    accuracy = correct / Y.shape[0]
    return accuracy

def get_samples(X, Y, n_samples):
    idxs = np.random.choice(X.shape[0], min(n_samples,X.shape[0]), replace=False)
    return X[idxs], Y[idxs]

def load_data(path):
    dataset = scipy.io.loadmat(path)
    train_data = dataset['Yt'].T
    val_data = dataset['Yv'].T
    train_labels = np.argmax(dataset['Ct'], axis=0)
    val_labels = np.argmax(dataset['Cv'], axis=0)
    return train_data, train_labels, val_data, val_labels
```

Fig 7: compute accuracy, get samples, load data methods

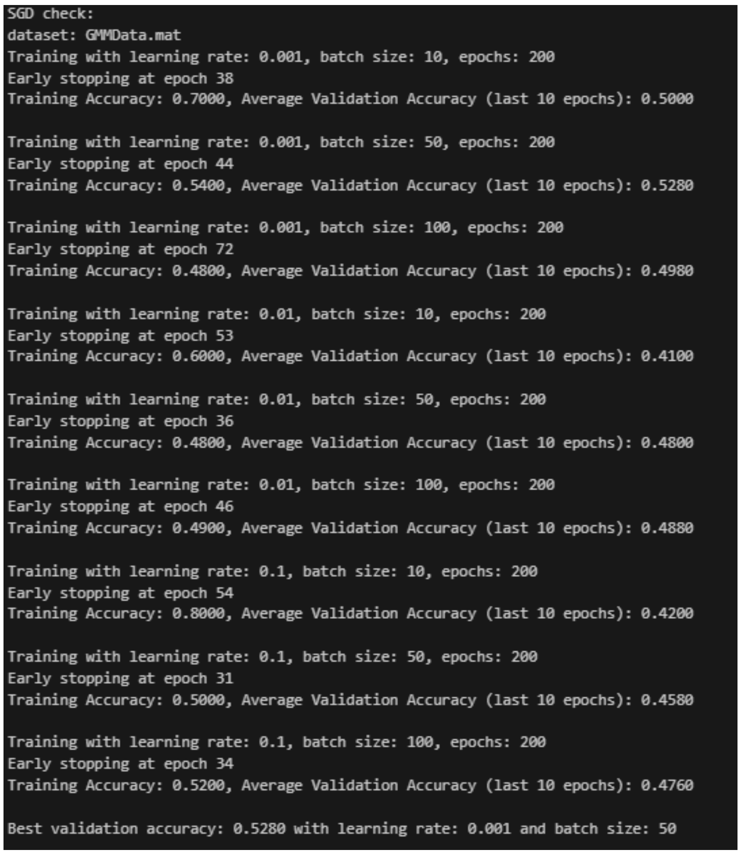


```python
def sgd(X_train, y_train, X_val, y_val, lr, batch_size, epochs):
    print(f"Training with learning rate: {lr}, batch size: {batch_size}, epochs: {epochs}")
    num_of_tries = 30
    num_features = X_train.shape[1]
    num_classes = len(np.unique(y_train))
    W = np.random.randn(num_features, num_classes)/num_features
    b = np.zeros((1, num_classes))

    train_accuracies = []
    val_accuracies = []
    avg_val_acc = 0
    best_val_acc = 0
    epochs_without_improvement = 0  # Track how many epochs without improvement

    for epoch in range(epochs):
        shuffled_indices = np.random.permutation(len(X_train))
        train_data = X_train[shuffled_indices]
        Y = y_train[shuffled_indices]

        for i in range(len(train_data) // batch_size):
            batch_X, batch_Y = get_batch(train_data, Y, batch_size, i)
            dW, db = Utils.softmax_gradient(batch_X, batch_Y, W, b)
            W -= lr * dW
            b -= lr * db

        X_sample, Y_sample = Utils.get_samples(X_train, y_train, batch_size)
        train_acc = Utils.compute_accuracy(X_sample, Y_sample, W, b)
        train_accuracies.append(train_acc)

        X_sample, Y_sample = Utils.get_samples(X_val, y_val, batch_size)
        val_acc = Utils.compute_accuracy(X_sample, Y_sample, W, b)
        val_accuracies.append(val_acc)

        # Check for improvement in validation accuracy
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            epochs_without_improvement = 0
        else:
            epochs_without_improvement += 1

        if epochs_without_improvement >= num_of_tries:
            print(f"Early stopping at epoch {epoch + 1}")
            break

    avg_val_acc = np.mean(val_accuracies[-10:])
    print(f"Training Accuracy: {train_acc:.4f}, Average Validation Accuracy (last 10 epochs): {avg_val_acc:.4f}")
    return train_accuracies, val_accuracies, avg_val_acc
```

Fig 8: The SGD

Results

- GMMData
  - Best validation accuracy: 0.5280
  - Best lr: 0.001
  - Best mini-batch size: 50



```
SGD check:
dataset: GMMData.mat
Training with learning rate: 0.001, batch size: 10, epochs: 200
Early stopping at epoch 38
Training Accuracy: 0.7000, Average Validation Accuracy (last 10 epochs): 0.5000

Training with learning rate: 0.001, batch size: 50, epochs: 200
Early stopping at epoch 44
Training Accuracy: 0.5400, Average Validation Accuracy (last 10 epochs): 0.5280

Training with learning rate: 0.001, batch size: 100, epochs: 200
Early stopping at epoch 72
Training Accuracy: 0.4800, Average Validation Accuracy (last 10 epochs): 0.4980

Training with learning rate: 0.01, batch size: 10, epochs: 200
Early stopping at epoch 53
Training Accuracy: 0.6000, Average Validation Accuracy (last 10 epochs): 0.4100

Training with learning rate: 0.01, batch size: 50, epochs: 200
Early stopping at epoch 36
Training Accuracy: 0.4800, Average Validation Accuracy (last 10 epochs): 0.4800

Training with learning rate: 0.01, batch size: 100, epochs: 200
Early stopping at epoch 46
Training Accuracy: 0.4900, Average Validation Accuracy (last 10 epochs): 0.4880

Training with learning rate: 0.1, batch size: 10, epochs: 200
Early stopping at epoch 54
Training Accuracy: 0.8000, Average Validation Accuracy (last 10 epochs): 0.4200

Training with learning rate: 0.1, batch size: 50, epochs: 200
Early stopping at epoch 31
Training Accuracy: 0.5000, Average Validation Accuracy (last 10 epochs): 0.4580

Training with learning rate: 0.1, batch size: 100, epochs: 200
Early stopping at epoch 34
Training Accuracy: 0.5200, Average Validation Accuracy (last 10 epochs): 0.4760

Best validation accuracy: 0.5280 with learning rate: 0.001 and batch size: 50
```

Fig 9: GMMData output for different parameters

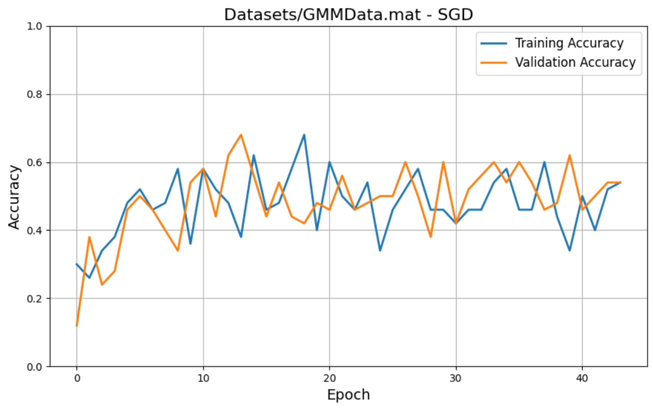
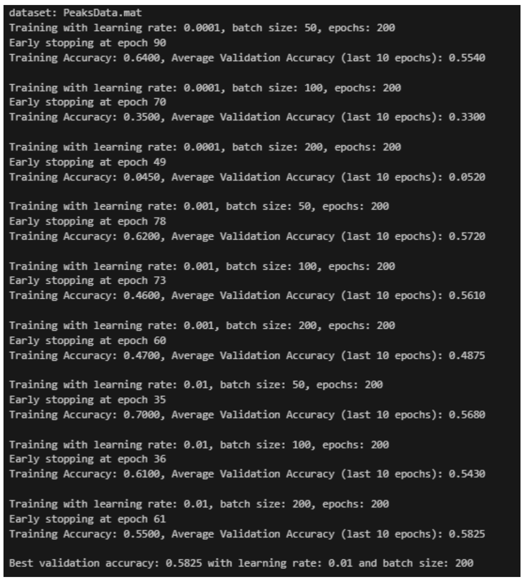


Fig 10: GMMData SGD training and validation accuracy

PeaksData

- Best validation accuracy: 0.5825
- Best lr: 0.01
- Best mini-batch size: 200



```
dataset: PeaksData.mat
Training with learning rate: 0.0001, batch size: 50, epochs: 200
Early stopping at epoch 90
Training Accuracy: 0.6400, Average Validation Accuracy (last 10 epochs): 0.5540

Training with learning rate: 0.0001, batch size: 100, epochs: 200
Early stopping at epoch 70
Training Accuracy: 0.3500, Average Validation Accuracy (last 10 epochs): 0.3300

Training with learning rate: 0.0001, batch size: 200, epochs: 200
Early stopping at epoch 49
Training Accuracy: 0.0450, Average Validation Accuracy (last 10 epochs): 0.0520

Training with learning rate: 0.001, batch size: 50, epochs: 200
Early stopping at epoch 78
Training Accuracy: 0.6200, Average Validation Accuracy (last 10 epochs): 0.5720

Training with learning rate: 0.001, batch size: 100, epochs: 200
Early stopping at epoch 73
Training Accuracy: 0.4600, Average Validation Accuracy (last 10 epochs): 0.5610

Training with learning rate: 0.001, batch size: 200, epochs: 200
Early stopping at epoch 60
Training Accuracy: 0.4700, Average Validation Accuracy (last 10 epochs): 0.4875

Training with learning rate: 0.01, batch size: 50, epochs: 200
Early stopping at epoch 35
Training Accuracy: 0.7000, Average Validation Accuracy (last 10 epochs): 0.5680

Training with learning rate: 0.01, batch size: 100, epochs: 200
Early stopping at epoch 36
Training Accuracy: 0.6100, Average Validation Accuracy (last 10 epochs): 0.5430

Training with learning rate: 0.01, batch size: 200, epochs: 200
Early stopping at epoch 61
Training Accuracy: 0.5500, Average Validation Accuracy (last 10 epochs): 0.5825

Best validation accuracy: 0.5825 with learning rate: 0.01 and batch size: 200
```

Fig 11: PeaksData output for different parameters

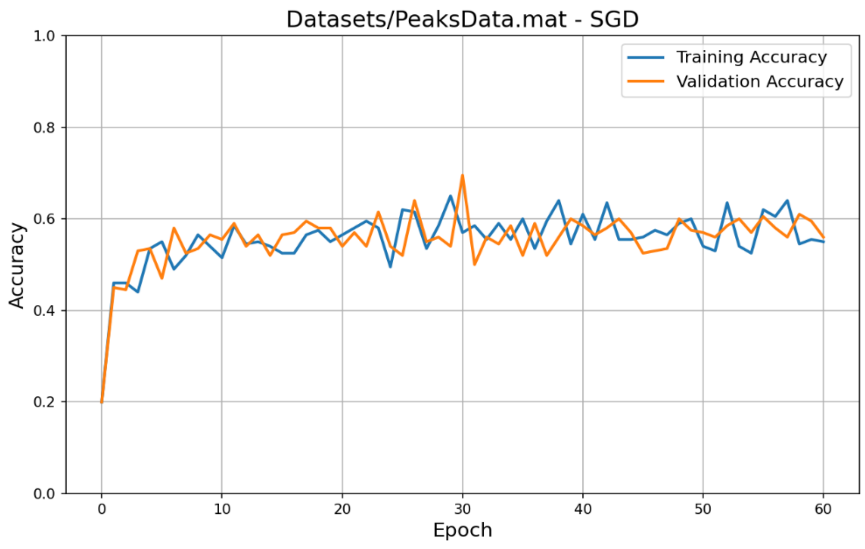
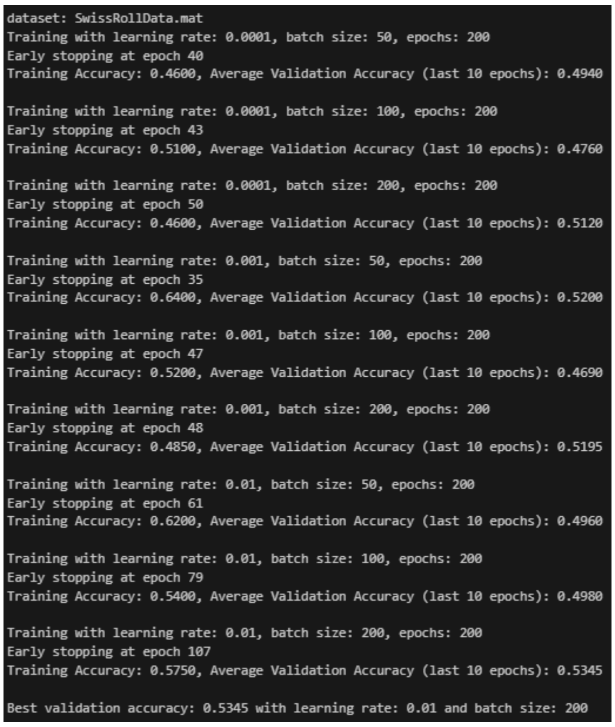


Fig 12: PeaksData SGD training and validation accuracy

SwissRollData

- Best validation accuracy: 0.5345
- Best lr: 0.01
- Best mini-batch size: 200



```
dataset: SwissRollData.mat
Training with learning rate: 0.0001, batch size: 50, epochs: 200
Early stopping at epoch 40
Training Accuracy: 0.4600, Average Validation Accuracy (last 10 epochs): 0.4940

Training with learning rate: 0.0001, batch size: 100, epochs: 200
Early stopping at epoch 43
Training Accuracy: 0.5100, Average Validation Accuracy (last 10 epochs): 0.4760

Training with learning rate: 0.0001, batch size: 200, epochs: 200
Early stopping at epoch 50
Training Accuracy: 0.4600, Average Validation Accuracy (last 10 epochs): 0.5120

Training with learning rate: 0.001, batch size: 50, epochs: 200
Early stopping at epoch 35
Training Accuracy: 0.6400, Average Validation Accuracy (last 10 epochs): 0.5200

Training with learning rate: 0.001, batch size: 100, epochs: 200
Early stopping at epoch 47
Training Accuracy: 0.5200, Average Validation Accuracy (last 10 epochs): 0.4690

Training with learning rate: 0.001, batch size: 200, epochs: 200
Early stopping at epoch 48
Training Accuracy: 0.4850, Average Validation Accuracy (last 10 epochs): 0.5195

Training with learning rate: 0.01, batch size: 50, epochs: 200
Early stopping at epoch 61
Training Accuracy: 0.6200, Average Validation Accuracy (last 10 epochs): 0.4960

Training with learning rate: 0.01, batch size: 100, epochs: 200
Early stopping at epoch 79
Training Accuracy: 0.5400, Average Validation Accuracy (last 10 epochs): 0.4980

Training with learning rate: 0.01, batch size: 200, epochs: 200
Early stopping at epoch 107
Training Accuracy: 0.5750, Average Validation Accuracy (last 10 epochs): 0.5345

Best validation accuracy: 0.5345 with learning rate: 0.01 and batch size: 200
```

Fig 13: SwissRollData output for different parameters

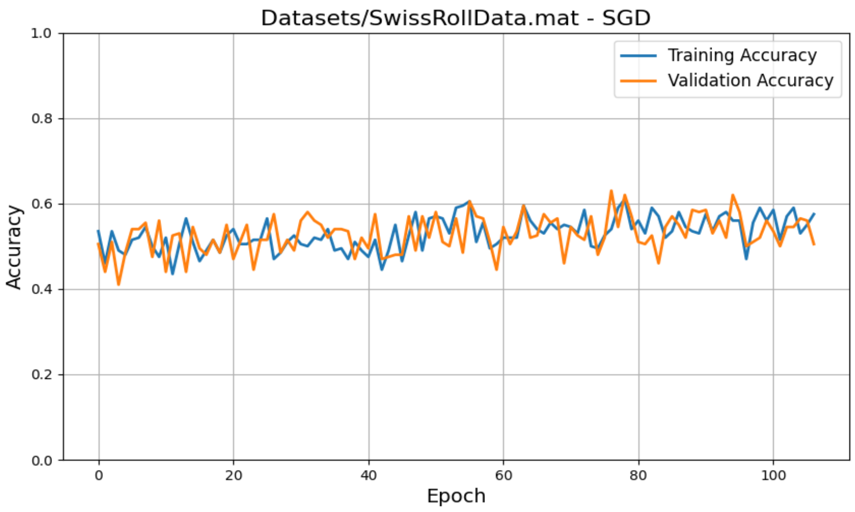


Fig 14: SwissRollData SGD training and validation accuracy

2. Part II: the neural network

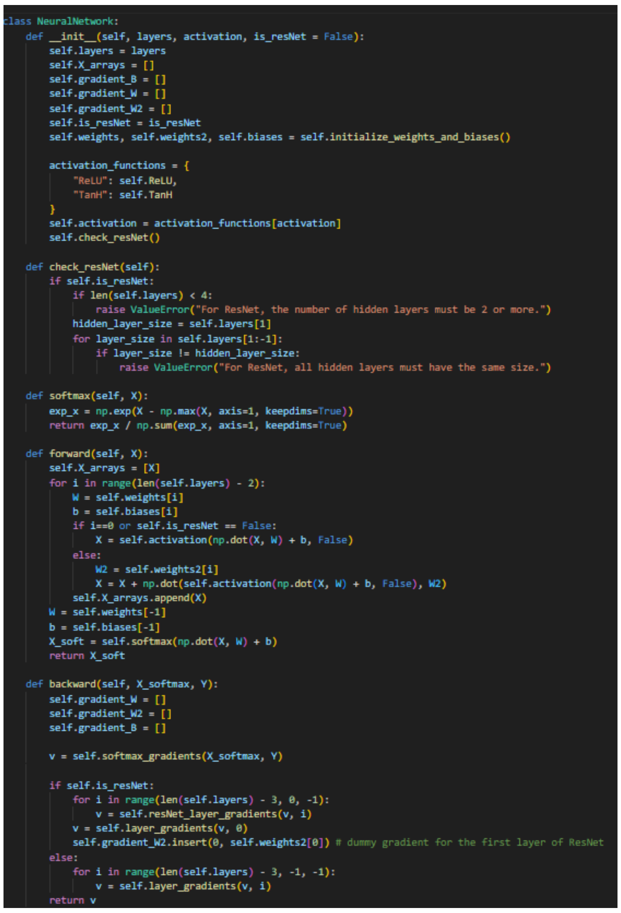
Neural Network implementation

Our neural network implementation is encapsulated within a class called NeuralNetwork. The goal of this class is to provide a dynamic, flexible structure that integrates all the necessary methods and fields for creating, training, and using a neural network model.

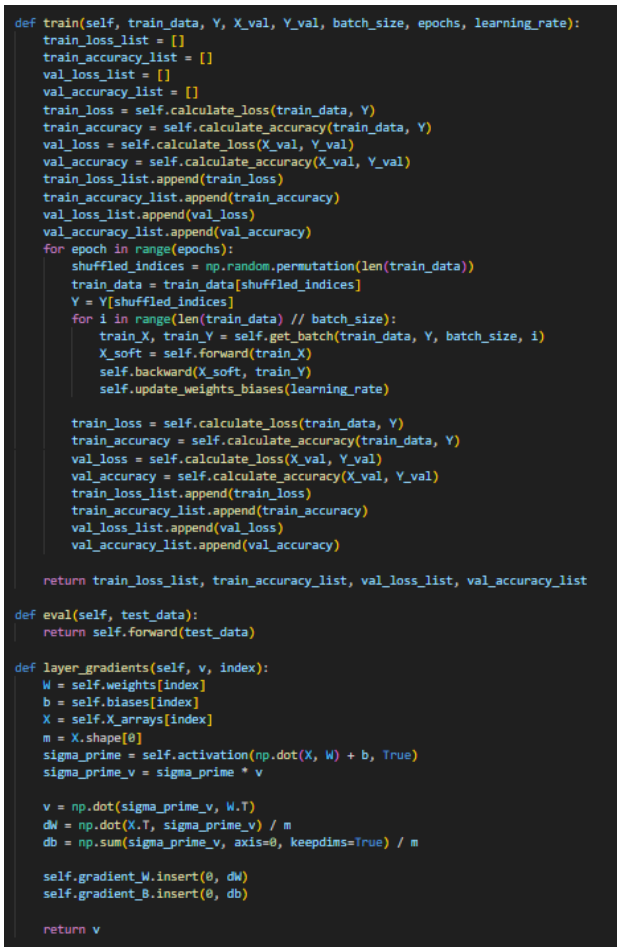
Key Features:

1. **Initialization:**
   - The user specifies the network's architecture by providing the layer structure, the activation function (ReLU or TanH), and whether the network includes residual connections (ResNet).
   - During initialization, the class automatically sets up:
     - **Weights:** Initialized with random values and normalized to ensure numerical stability.
     - **Biases:** Initialized as zero vectors.
   - If ResNet is enabled, the initialization process includes additional validations:
     - All hidden layers must have the same size to ensure compatibility with residual connections.
     - The network must include at least two hidden layers. If either condition is not met, an error is raised.
2. **Dynamic Training:**
   - The train method implements stochastic gradient descent (SGD) and ensures data variety during training by shuffling the dataset indices at the start of each epoch. This prevents the model from overfitting to a fixed data order and improves generalization.

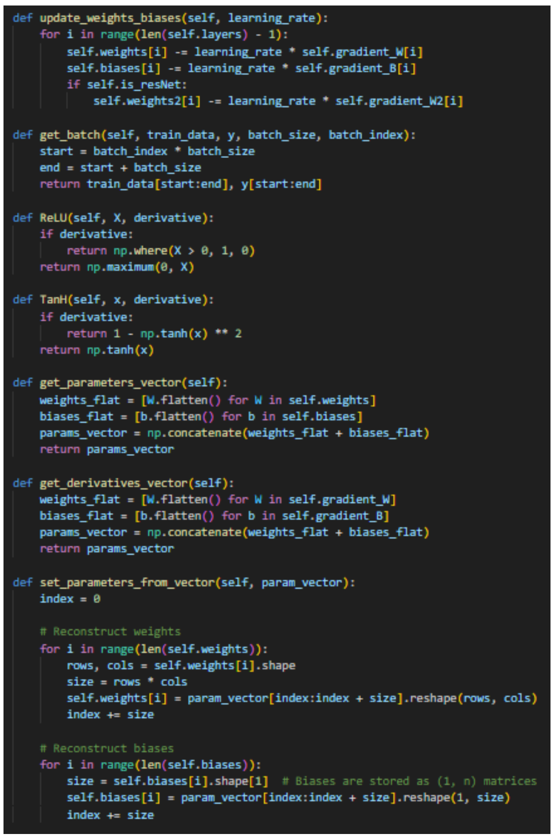
This unified design ensures that the NeuralNetwork class is highly reusable, allowing users to easily define, customize, and train their models with minimal setup.



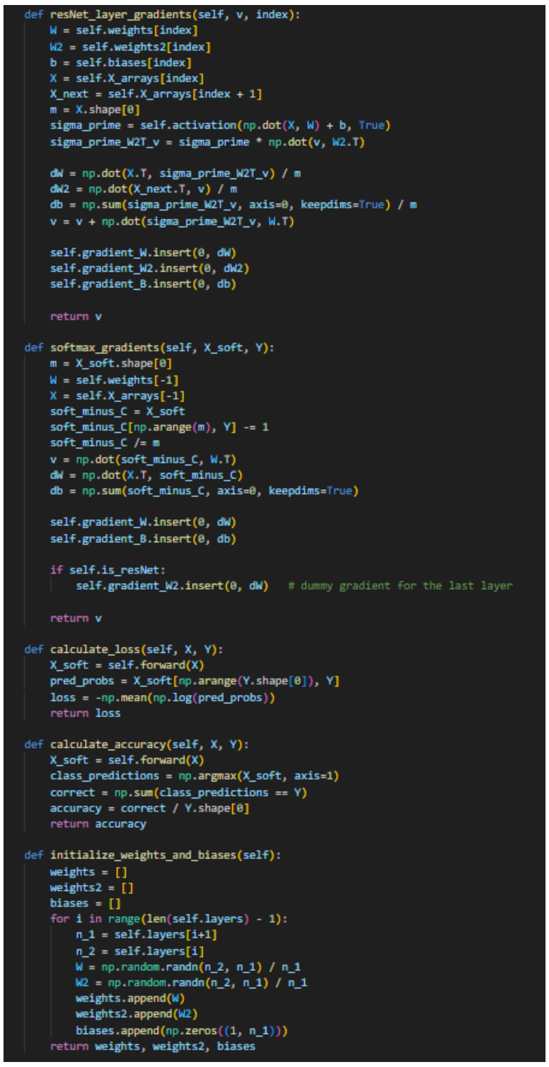
```python
class NeuralNetwork:
    def __init__(self, layers, activation, is_resNet = False):
        self.layers = layers
        self.X_arrays = []
        self.gradient_B = []
        self.gradient_W = []
        self.gradient_W2 = []
        self.is_resNet = is_resNet
        self.weights, self.weights2, self.biases = self.initialize_weights_and_biases()

        activation_functions = {
            "ReLU": self.ReLU,
            "TanH": self.TanH
        }
        self.activation = activation_functions[activation]
        self.check_resNet()

    def check_resNet(self):
        if self.is_resNet:
            if len(self.layers) < 4:
                raise ValueError("For ResNet, the number of hidden layers must be 2 or more.")
            hidden_layer_size = self.layers[1]
            for layer_size in self.layers[1:-1]:
                if layer_size != hidden_layer_size:
                    raise ValueError("For ResNet, all hidden layers must have the same size.")

    def softmax(self, X):
        exp_x = np.exp(X - np.max(X, axis=1, keepdims=True))
        return exp_x / np.sum(exp_x, axis=1, keepdims=True)

    def forward(self, X):
        self.X_arrays = [X]
        for i in range(len(self.layers) - 2):
            W = self.weights[i]
            b = self.biases[i]
            if i==0 or self.is_resNet == False:
                X = self.activation(np.dot(X, W) + b, False)
            else:
                W2 = self.weights2[i]
                X = X + np.dot(self.activation(np.dot(X, W) + b, False), W2)
            self.X_arrays.append(X)
        W = self.weights[-1]
        b = self.biases[-1]
        X_soft = self.softmax(np.dot(X, W) + b)
        return X_soft

    def backward(self, X_softmax, Y):
        self.gradient_W = []
        self.gradient_W2 = []
        self.gradient_B = []

        v = self.softmax_gradients(X_softmax, Y)

        if self.is_resNet:
            for i in range(len(self.layers) - 3, 0, -1):
                v = self.resNet_layer_gradients(v, i)
            v = self.layer_gradients(v, 0)
            self.gradient_W2.insert(0, self.weights2[0]) # dummy gradient for the first layer of ResNet
        else:
            for i in range(len(self.layers) - 3, -1, -1):
                v = self.layer_gradients(v, i)
        return v
```



```python
def train(self, train_data, Y, X_val, Y_val, batch_size, epochs, learning_rate):
    train_loss_list = []
    train_accuracy_list = []
    val_loss_list = []
    val_accuracy_list = []
    train_loss = self.calculate_loss(train_data, Y)
    train_accuracy = self.calculate_accuracy(train_data, Y)
    val_loss = self.calculate_loss(X_val, Y_val)
    val_accuracy = self.calculate_accuracy(X_val, Y_val)
    train_loss_list.append(train_loss)
    train_accuracy_list.append(train_accuracy)
    val_loss_list.append(val_loss)
    val_accuracy_list.append(val_accuracy)
    for epoch in range(epochs):
        shuffled_indices = np.random.permutation(len(train_data))
        train_data = train_data[shuffled_indices]
        Y = Y[shuffled_indices]
        for i in range(len(train_data) // batch_size):
            train_X, train_Y = self.get_batch(train_data, Y, batch_size, i)
            X_soft = self.forward(train_X)
            self.backward(X_soft, train_Y)
            self.update_weights_biases(learning_rate)

        train_loss = self.calculate_loss(train_data, Y)
        train_accuracy = self.calculate_accuracy(train_data, Y)
        val_loss = self.calculate_loss(X_val, Y_val)
        val_accuracy = self.calculate_accuracy(X_val, Y_val)
        train_loss_list.append(train_loss)
        train_accuracy_list.append(train_accuracy)
        val_loss_list.append(val_loss)
        val_accuracy_list.append(val_accuracy)

    return train_loss_list, train_accuracy_list, val_loss_list, val_accuracy_list

def eval(self, test_data):
    return self.forward(test_data)

def layer_gradients(self, v, index):
    W = self.weights[index]
    b = self.biases[index]
    X = self.X_arrays[index]
    m = X.shape[0]
    sigma_prime = self.activation(np.dot(X, W) + b, True)
    sigma_prime_v = sigma_prime * v

    v = np.dot(sigma_prime_v, W.T)
    dW = np.dot(X.T, sigma_prime_v) / m
    db = np.sum(sigma_prime_v, axis=0, keepdims=True) / m

    self.gradient_W.insert(0, dW)
    self.gradient_B.insert(0, db)

    return v
```



```python
def update_weights_biases(self, learning_rate):
    for i in range(len(self.layers) - 1):
        self.weights[i] -= learning_rate * self.gradient_W[i]
        self.biases[i] -= learning_rate * self.gradient_B[i]
        if self.is_resNet:
            self.weights2[i] -= learning_rate * self.gradient_W2[i]

def get_batch(self, train_data, y, batch_size, batch_index):
    start = batch_index * batch_size
    end = start + batch_size
    return train_data[start:end], y[start:end]

def ReLU(self, X, derivative):
    if derivative:
        return np.where(X > 0, 1, 0)
    return np.maximum(0, X)

def TanH(self, x, derivative):
    if derivative:
        return 1 - np.tanh(x) ** 2
    return np.tanh(x)

def get_parameters_vector(self):
    weights_flat = [W.flatten() for W in self.weights]
    biases_flat = [b.flatten() for b in self.biases]
    params_vector = np.concatenate(weights_flat + biases_flat)
    return params_vector

def get_derivatives_vector(self):
    weights_flat = [W.flatten() for W in self.gradient_W]
    biases_flat = [b.flatten() for b in self.gradient_B]
    params_vector = np.concatenate(weights_flat + biases_flat)
    return params_vector

def set_parameters_from_vector(self, param_vector):
    index = 0

    # Reconstruct weights
    for i in range(len(self.weights)):
        rows, cols = self.weights[i].shape
        size = rows * cols
        self.weights[i] = param_vector[index:index + size].reshape(rows, cols)
        index += size

    # Reconstruct biases
    for i in range(len(self.biases)):
        size = self.biases[i].shape[1]  # Biases are stored as (1, n) matrices
        self.biases[i] = param_vector[index:index + size].reshape(1, size)
        index += size
```



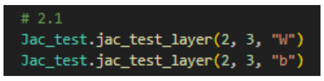
```python
def resNet_layer_gradients(self, v, index):
    W = self.weights[index]
    W2 = self.weights2[index]
    b = self.biases[index]
    X = self.X_arrays[index]
    X_next = self.X_arrays[index + 1]
    m = X.shape[0]
    sigma_prime = self.activation(np.dot(X, W) + b, True)
    sigma_prime_W2T_v = sigma_prime * np.dot(v, W2.T)

    dW = np.dot(X.T, sigma_prime_W2T_v) / m
    dW2 = np.dot(X_next.T, v) / m
    db = np.sum(sigma_prime_W2T_v, axis=0, keepdims=True) / m
    v = v + np.dot(sigma_prime_W2T_v, W.T)

    self.gradient_W.insert(0, dW)
    self.gradient_W2.insert(0, dW2)
    self.gradient_B.insert(0, db)

    return v

def softmax_gradients(self, X_soft, Y):
    m = X_soft.shape[0]
    W = self.weights[-1]
    X = self.X_arrays[-1]
    soft_minus_C = X_soft
    soft_minus_C[np.arange(m), Y] -= 1
    soft_minus_C /= m
    v = np.dot(soft_minus_C, W.T)
    dW = np.dot(X.T, soft_minus_C)
    db = np.sum(soft_minus_C, axis=0, keepdims=True)

    self.gradient_W.insert(0, dW)
    self.gradient_B.insert(0, db)

    if self.is_resNet:
        self.gradient_W2.insert(0, dW)   # dummy gradient for the last layer

    return v

def calculate_loss(self, X, Y):
    X_soft = self.forward(X)
    pred_probs = X_soft[np.arange(Y.shape[0]), Y]
    loss = -np.mean(np.log(pred_probs))
    return loss

def calculate_accuracy(self, X, Y):
    X_soft = self.forward(X)
    class_predictions = np.argmax(X_soft, axis=1)
    correct = np.sum(class_predictions == Y)
    accuracy = correct / Y.shape[0]
    return accuracy

def initialize_weights_and_biases(self):
    weights = []
    weights2 = []
    biases = []
    for i in range(len(self.layers) - 1):
        n_1 = self.layers[i+1]
        n_2 = self.layers[i]
        W = np.random.randn(n_2, n_1) / n_1
        W2 = np.random.randn(n_2, n_1) / n_1
        weights.append(W)
        weights2.append(W2)
        biases.append(np.zeros((1, n_1)))
    return weights, weights2, biases
```

Fig 15-18: The neural network class

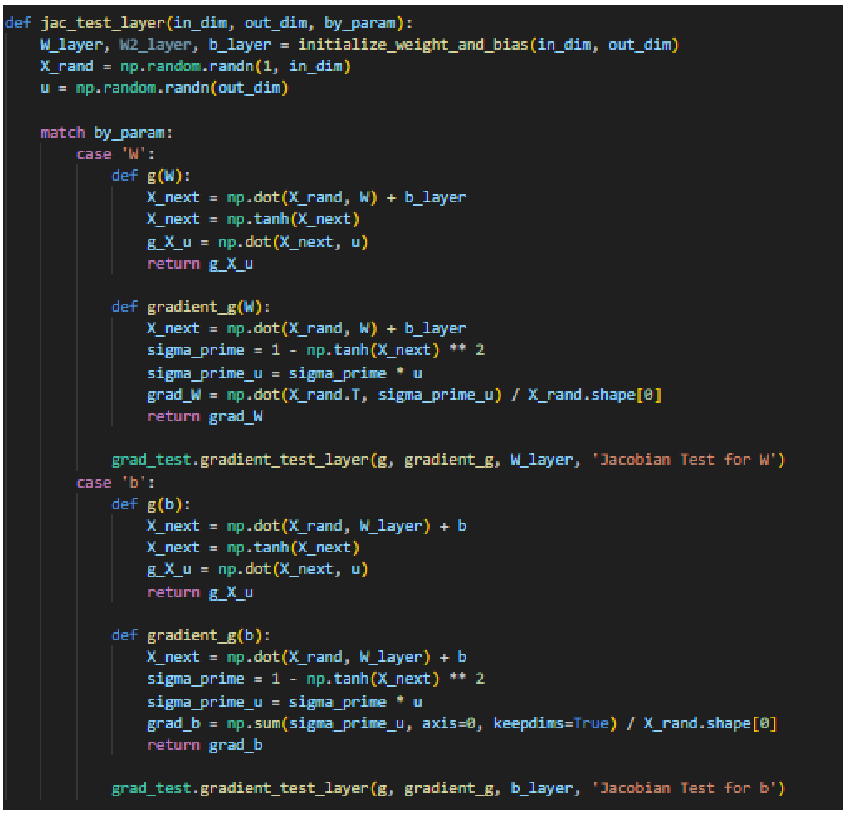
2.1. Jacobian Test to the layers

We validated the correctness of our forward and backward passes using the Jacobian test. Specifically, we implemented the **Direct Jacobian Transposed Test** as outlined in the course notes.

We applied this test to the "regular" model layer.



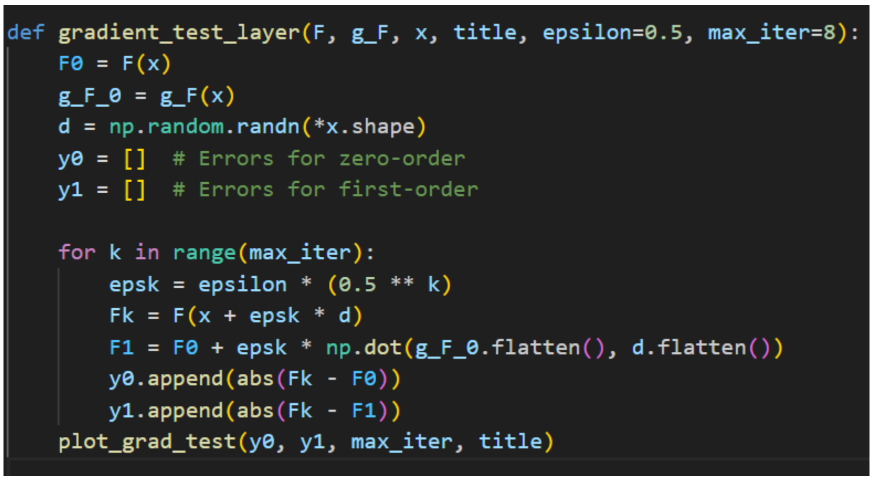
```python
# 2.1
Jac_test.jac_test_layer(2, 3, "W")
Jac_test.jac_test_layer(2, 3, "b")
```



```python
def jac_test_layer(in_dim, out_dim, by_param):
    W_layer, W2_layer, b_layer = initialize_weight_and_bias(in_dim, out_dim)
    X_rand = np.random.randn(1, in_dim)
    u = np.random.randn(out_dim)

    match by_param:
        case 'W':
            def g(W):
                X_next = np.dot(X_rand, W) + b_layer
                X_next = np.tanh(X_next)
                g_X_u = np.dot(X_next, u)
                return g_X_u

            def gradient_g(W):
                X_next = np.dot(X_rand, W) + b_layer
                sigma_prime = 1 - np.tanh(X_next) ** 2
                sigma_prime_u = sigma_prime * u
                grad_W = np.dot(X_rand.T, sigma_prime_u) / X_rand.shape[0]
                return grad_W

            grad_test.gradient_test_layer(g, gradient_g, W_layer, 'Jacobian Test for W')
        case 'b':
            def g(b):
                X_next = np.dot(X_rand, W_layer) + b
                X_next = np.tanh(X_next)
                g_X_u = np.dot(X_next, u)
                return g_X_u

            def gradient_g(b):
                X_next = np.dot(X_rand, W_layer) + b
                sigma_prime = 1 - np.tanh(X_next) ** 2
                sigma_prime_u = sigma_prime * u
                grad_b = np.sum(sigma_prime_u, axis=0, keepdims=True) / X_rand.shape[0]
                return grad_b

            grad_test.gradient_test_layer(g, gradient_g, b_layer, 'Jacobian Test for b')
```

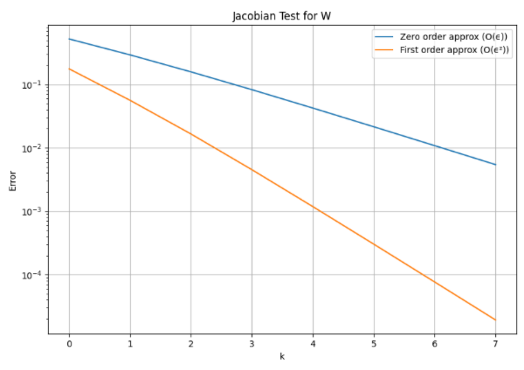
Fig 19-20: Jacobian test layer

To use the **Direct Jacobian Transposed test,** we implemented the "grad test" as shown in class (with small changes of dimensions transformation).



```python
def gradient_test_layer(F, g_F, x, title, epsilon=0.5, max_iter=8):
    F0 = F(x)
    g_F_0 = g_F(x)
    d = np.random.randn(*x.shape)
    y0 = []  # Errors for zero-order
    y1 = []  # Errors for first-order

    for k in range(max_iter):
        epsk = epsilon * (0.5 ** k)
        Fk = F(x + epsk * d)
        F1 = F0 + epsk * np.dot(g_F_0.flatten(), d.flatten())
        y0.append(abs(Fk - F0))
        y1.append(abs(Fk - F1))
    plot_grad_test(y0, y1, max_iter, title)
```

Fig 21: the gradient test



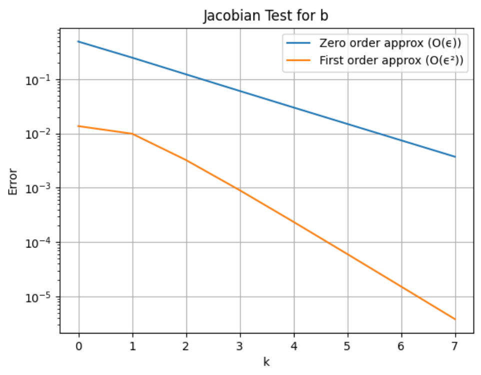


Fig 22-23: the gradient test plots w.r.t W and b

2.2. Jacobian Test - Residual Neural Network

As previously mentioned, the implementation of the ResNet model is integrated within the NeuralNetwork class. The ResNet architecture imposes specific constraints on the model's structure:

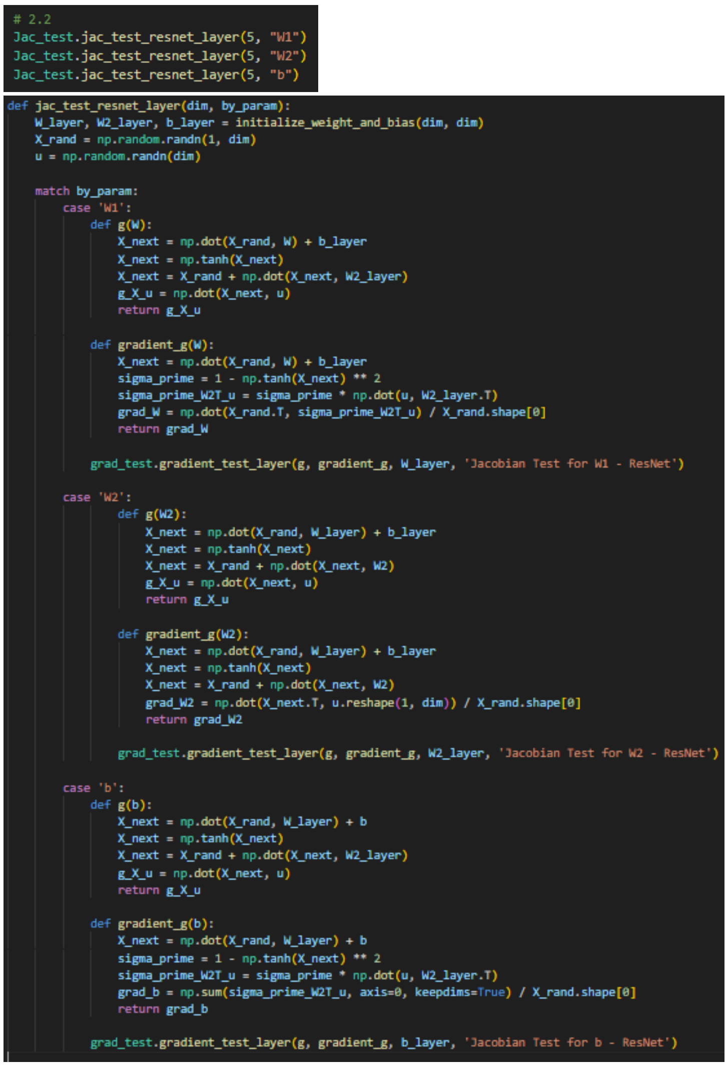
- The network must have at least **two hidden layers.**
- All hidden layers must be of the **same size** to ensure compatibility with residual connections.

Like the "regular" model, the ResNet implementation was rigorously tested for correctness in both forward and backward passes using the **Direct Jacobian Transposed Test.**

* The ResNet architecture relies on the following equation for residual connections:

$$x^{l+1} = x^l + W_2^l \, \sigma\left(W_1^l \, x^l + b^l\right)$$

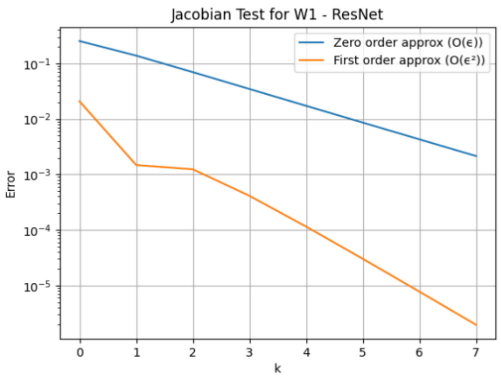
For the addition $x^l + W_2^l \, \sigma(\dots)$ to be valid, the dimensions of x^l and the output of $W_2^l \, \sigma(\dots)$ must match. Hence, all hidden layers must maintain the same size to ensure dimensional consistency.

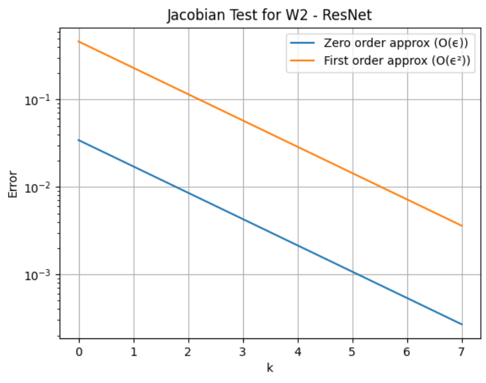


```python
# 2.2
Jac_test.jac_test_resnet_layer(5, "W1")
Jac_test.jac_test_resnet_layer(5, "W2")
Jac_test.jac_test_resnet_layer(5, "b")
```

```python
def jac_test_resnet_layer(dim, by_param):
    W_layer, W2_layer, b_layer = initialize_weight_and_bias(dim, dim)
    X_rand = np.random.randn(1, dim)
    u = np.random.randn(dim)

    match by_param:
        case 'W1':
            def g(W):
                X_next = np.dot(X_rand, W) + b_layer
                X_next = np.tanh(X_next)
                X_next = X_rand + np.dot(X_next, W2_layer)
                g_X_u = np.dot(X_next, u)
                return g_X_u

            def gradient_g(W):
                X_next = np.dot(X_rand, W) + b_layer
                sigma_prime = 1 - np.tanh(X_next) ** 2
                sigma_prime_W2T_u = sigma_prime * np.dot(u, W2_layer.T)
                grad_W = np.dot(X_rand.T, sigma_prime_W2T_u) / X_rand.shape[0]
                return grad_W

            grad_test.gradient_test_layer(g, gradient_g, W_layer, 'Jacobian Test for W1 - ResNet')

        case 'W2':
                def g(W2):
                    X_next = np.dot(X_rand, W_layer) + b_layer
                    X_next = np.tanh(X_next)
                    X_next = X_rand + np.dot(X_next, W2)
                    g_X_u = np.dot(X_next, u)
                    return g_X_u

                def gradient_g(W2):
                    X_next = np.dot(X_rand, W_layer) + b_layer
                    X_next = np.tanh(X_next)
                    X_next = X_rand + np.dot(X_next, W2)
                    grad_W2 = np.dot(X_next.T, u.reshape(1, dim)) / X_rand.shape[0]
                    return grad_W2

                grad_test.gradient_test_layer(g, gradient_g, W2_layer, 'Jacobian Test for W2 - ResNet')

        case 'b':
            def g(b):
                X_next = np.dot(X_rand, W_layer) + b
                X_next = np.tanh(X_next)
                X_next = X_rand + np.dot(X_next, W2_layer)
                g_X_u = np.dot(X_next, u)
                return g_X_u

            def gradient_g(b):
                X_next = np.dot(X_rand, W_layer) + b
                sigma_prime = 1 - np.tanh(X_next) ** 2
                sigma_prime_W2T_u = sigma_prime * np.dot(u, W2_layer.T)
                grad_b = np.sum(sigma_prime_W2T_u, axis=0, keepdims=True) / X_rand.shape[0]
                return grad_b

            grad_test.gradient_test_layer(g, gradient_g, b_layer, 'Jacobian Test for b - ResNet')
```

Fig 24-25: Jacobian ResNet test





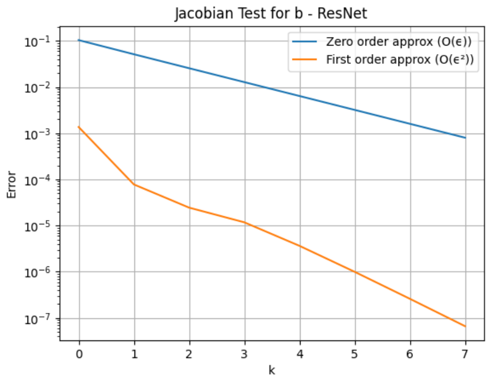
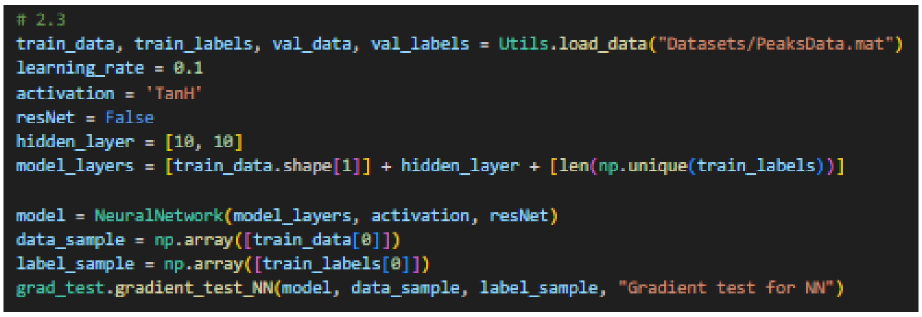


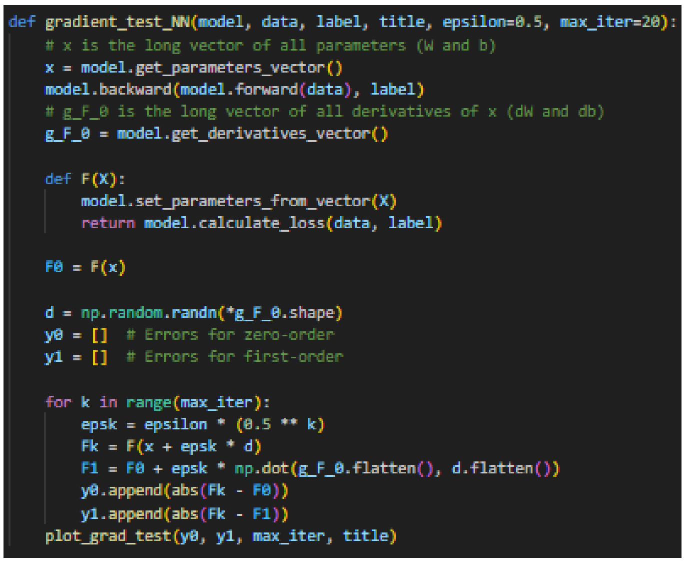
Fig 26-28: the gradient test plots w.r.t W1, W2 and b

2.3. Neural Network – Gradient Test

We validated the entire network using the Gradient Test, where we defined the complete forward pass (including the loss function) as the function and the entire backward pass as its gradient.
The test was conducted on a network with two hidden layers, ensuring that both the forward and backward computations were correctly implemented.
As illustrated in the plot below, the gradient test confirmed the accuracy of the network's computations, demonstrating successful validation of the entire architecture.



```
# 2.3
train_data, train_labels, val_data, val_labels = Utils.load_data("Datasets/PeaksData.mat")
learning_rate = 0.1
activation = 'TanH'
resNet = False
hidden_layer = [10, 10]
model_layers = [train_data.shape[1]] + hidden_layer + [len(np.unique(train_labels))]

model = NeuralNetwork(model_layers, activation, resNet)
data_sample = np.array([train_data[0]])
label_sample = np.array([train_labels[0]])
grad_test.gradient_test_NN(model, data_sample, label_sample, "Gradient test for NN")
```



```
def gradient_test_NN(model, data, label, title, epsilon=0.5, max_iter=20):
    # x is the long vector of all parameters (W and b)
    x = model.get_parameters_vector()
    model.backward(model.forward(data), label)
    # g_F_0 is the long vector of all derivatives of x (dW and db)
    g_F_0 = model.get_derivatives_vector()

    def F(X):
        model.set_parameters_from_vector(X)
        return model.calculate_loss(data, label)

    F0 = F(x)

    d = np.random.randn(*g_F_0.shape)
    y0 = []  # Errors for zero-order
    y1 = []  # Errors for first-order

    for k in range(max_iter):
        epsk = epsilon * (0.5 ** k)
        Fk = F(x + epsk * d)
        F1 = F0 + epsk * np.dot(g_F_0.flatten(), d.flatten())
        y0.append(abs(Fk - F0))
        y1.append(abs(Fk - F1))
    plot_grad_test(y0, y1, max_iter, title)
```

Fig 29-30: code implementation for gradient test on a full Neural Network

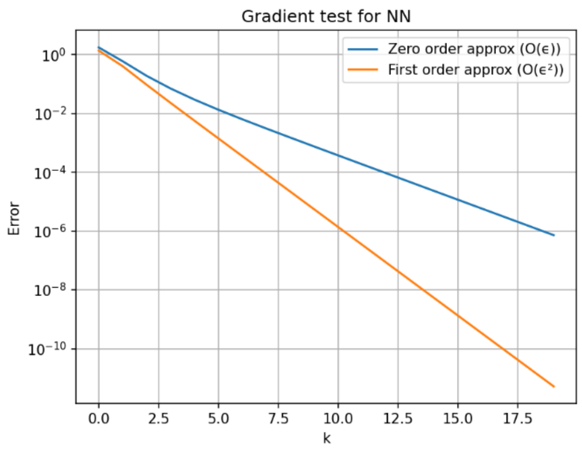
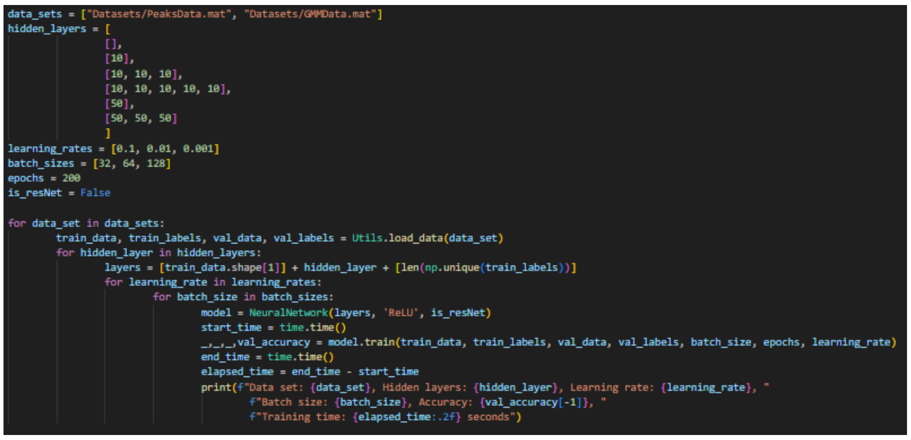


Fig 31: the gradient test for the Neural Network

2.4. Neural Network experiments with different parameters

We ran our NN with the datasets GMMData and PeaksData with the following parameters:

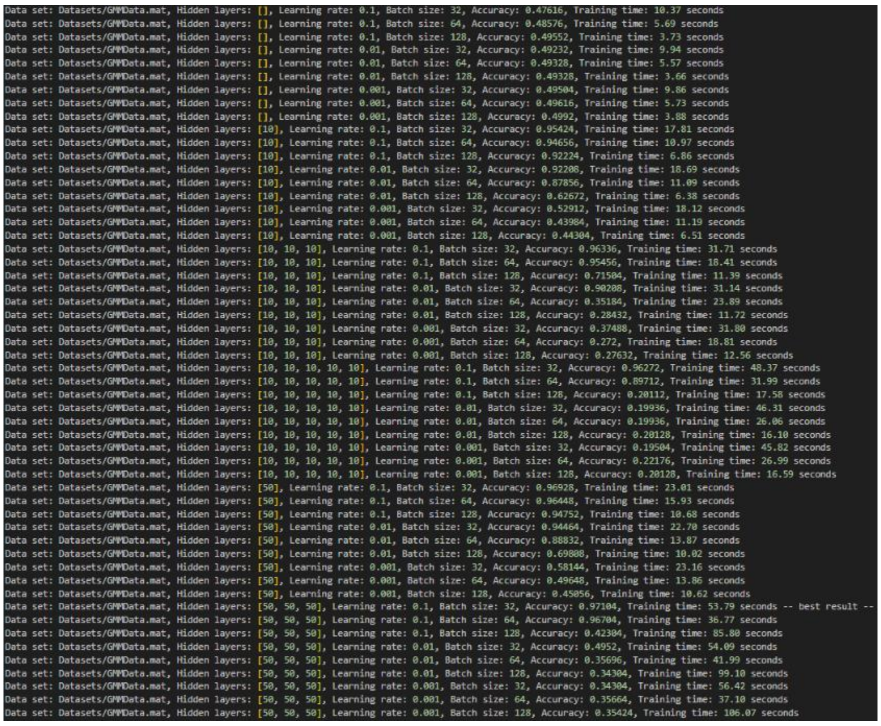
- Learning rates: [0.1, 0.01, 0.001]
- Mini-batch sizes: [32, 64, 128]
- Epochs: 200
- Activation function: ReLU
- Hidden layers: [[],[10], [10, 10, 10], [10, 10, 10, 10, 10], [50], [50,50,50]]



```python
data_sets = ["Datasets/PeaksData.mat", "Datasets/GMMData.mat"]
hidden_layers = [
                [],
                [10],
                [10, 10, 10],
                [10, 10, 10, 10, 10],
                [50],
                [50, 50, 50]
                ]
learning_rates = [0.1, 0.01, 0.001]
batch_sizes = [32, 64, 128]
epochs = 200
is_resNet = False

for data_set in data_sets:
        train_data, train_labels, val_data, val_labels = Utils.load_data(data_set)
        for hidden_layer in hidden_layers:
                layers = [train_data.shape[1]] + hidden_layer + [len(np.unique(train_labels))]
                for learning_rate in learning_rates:
                        for batch_size in batch_sizes:
                                model = NeuralNetwork(layers, 'ReLU', is_resNet)
                                start_time = time.time()
                                _,_,_,val_accuracy = model.train(train_data, train_labels, val_data, val_labels, batch_size, epochs, learning_rate)
                                end_time = time.time()
                                elapsed_time = end_time - start_time
                                print(f"Data set: {data_set}, Hidden layers: {hidden_layer}, Learning rate: {learning_rate}, "
                                        f"Batch size: {batch_size}, Accuracy: {val_accuracy[-1]}, "
                                        f"Training time: {elapsed_time:.2f} seconds")
```

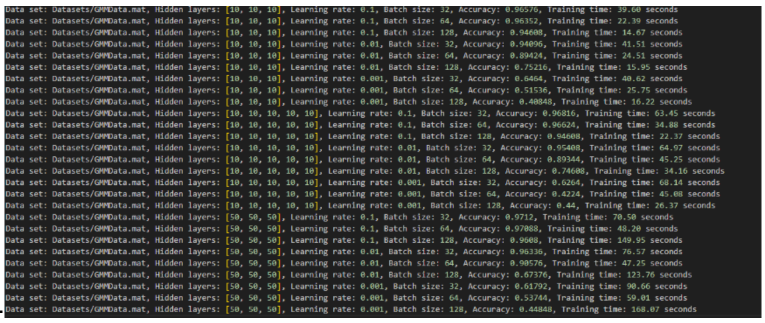
Fig 32: code implementation for running the NN with different parameters

The results given by running the code given above:



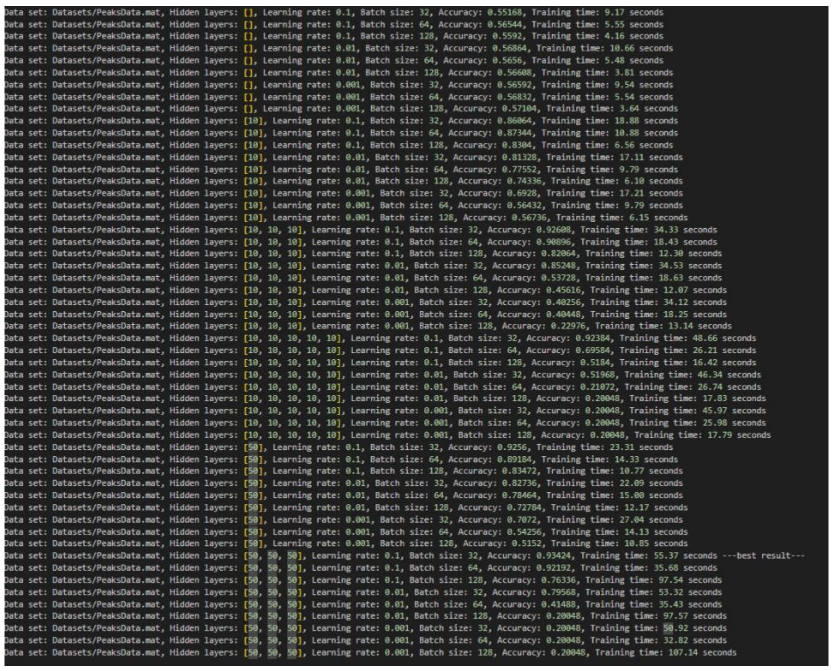
```
Data set: Datasets/GMMData.mat, Hidden layers: [], Learning rate: 0.1, Batch size: 32, Accuracy: 0.47616, Training time: 10.37 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [], Learning rate: 0.1, Batch size: 64, Accuracy: 0.48576, Training time: 5.69 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [], Learning rate: 0.1, Batch size: 128, Accuracy: 0.49552, Training time: 3.73 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [], Learning rate: 0.01, Batch size: 32, Accuracy: 0.49232, Training time: 9.94 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [], Learning rate: 0.01, Batch size: 64, Accuracy: 0.49328, Training time: 5.57 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [], Learning rate: 0.01, Batch size: 128, Accuracy: 0.49328, Training time: 3.66 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [], Learning rate: 0.001, Batch size: 32, Accuracy: 0.49504, Training time: 9.86 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [], Learning rate: 0.001, Batch size: 64, Accuracy: 0.49616, Training time: 5.73 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [], Learning rate: 0.001, Batch size: 128, Accuracy: 0.4992, Training time: 3.88 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10], Learning rate: 0.1, Batch size: 32, Accuracy: 0.95424, Training time: 17.81 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10], Learning rate: 0.1, Batch size: 64, Accuracy: 0.94656, Training time: 10.97 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10], Learning rate: 0.1, Batch size: 128, Accuracy: 0.92224, Training time: 6.86 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10], Learning rate: 0.01, Batch size: 32, Accuracy: 0.92208, Training time: 18.69 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10], Learning rate: 0.01, Batch size: 64, Accuracy: 0.87856, Training time: 11.09 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10], Learning rate: 0.01, Batch size: 128, Accuracy: 0.62672, Training time: 6.38 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10], Learning rate: 0.001, Batch size: 32, Accuracy: 0.52912, Training time: 18.12 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10], Learning rate: 0.001, Batch size: 64, Accuracy: 0.43984, Training time: 11.19 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10], Learning rate: 0.001, Batch size: 128, Accuracy: 0.44304, Training time: 6.51 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.1, Batch size: 32, Accuracy: 0.96336, Training time: 31.71 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.1, Batch size: 64, Accuracy: 0.95456, Training time: 18.41 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.1, Batch size: 128, Accuracy: 0.71504, Training time: 11.39 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.01, Batch size: 32, Accuracy: 0.90208, Training time: 31.14 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.01, Batch size: 64, Accuracy: 0.35184, Training time: 23.89 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.01, Batch size: 128, Accuracy: 0.28432, Training time: 11.72 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.001, Batch size: 32, Accuracy: 0.37488, Training time: 31.80 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.001, Batch size: 64, Accuracy: 0.272, Training time: 18.81 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.001, Batch size: 128, Accuracy: 0.27632, Training time: 12.56 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.1, Batch size: 32, Accuracy: 0.96272, Training time: 48.37 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.1, Batch size: 64, Accuracy: 0.89712, Training time: 31.99 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.1, Batch size: 128, Accuracy: 0.20112, Training time: 17.58 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.01, Batch size: 32, Accuracy: 0.19936, Training time: 46.31 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.01, Batch size: 64, Accuracy: 0.19936, Training time: 26.06 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.01, Batch size: 128, Accuracy: 0.20128, Training time: 16.10 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.001, Batch size: 32, Accuracy: 0.19504, Training time: 45.82 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.001, Batch size: 64, Accuracy: 0.22176, Training time: 26.99 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.001, Batch size: 128, Accuracy: 0.20128, Training time: 16.59 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50], Learning rate: 0.1, Batch size: 32, Accuracy: 0.96928, Training time: 23.01 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50], Learning rate: 0.1, Batch size: 64, Accuracy: 0.96448, Training time: 15.93 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50], Learning rate: 0.1, Batch size: 128, Accuracy: 0.94752, Training time: 10.68 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50], Learning rate: 0.01, Batch size: 32, Accuracy: 0.94464, Training time: 22.70 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50], Learning rate: 0.01, Batch size: 64, Accuracy: 0.88832, Training time: 13.87 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50], Learning rate: 0.01, Batch size: 128, Accuracy: 0.69808, Training time: 10.02 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50], Learning rate: 0.001, Batch size: 32, Accuracy: 0.58144, Training time: 23.16 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50], Learning rate: 0.001, Batch size: 64, Accuracy: 0.49648, Training time: 13.86 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50], Learning rate: 0.001, Batch size: 128, Accuracy: 0.45056, Training time: 10.62 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.1, Batch size: 32, Accuracy: 0.97104, Training time: 53.79 seconds -- best result --
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.1, Batch size: 64, Accuracy: 0.96704, Training time: 36.77 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.1, Batch size: 128, Accuracy: 0.42304, Training time: 85.80 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.01, Batch size: 32, Accuracy: 0.4952, Training time: 54.09 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.01, Batch size: 64, Accuracy: 0.35696, Training time: 41.99 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.01, Batch size: 128, Accuracy: 0.34304, Training time: 99.10 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.001, Batch size: 32, Accuracy: 0.34304, Training time: 56.42 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.001, Batch size: 64, Accuracy: 0.35664, Training time: 37.10 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.001, Batch size: 128, Accuracy: 0.35424, Training time: 106.07 seconds
```

Fig 33: GMMData without ResNet



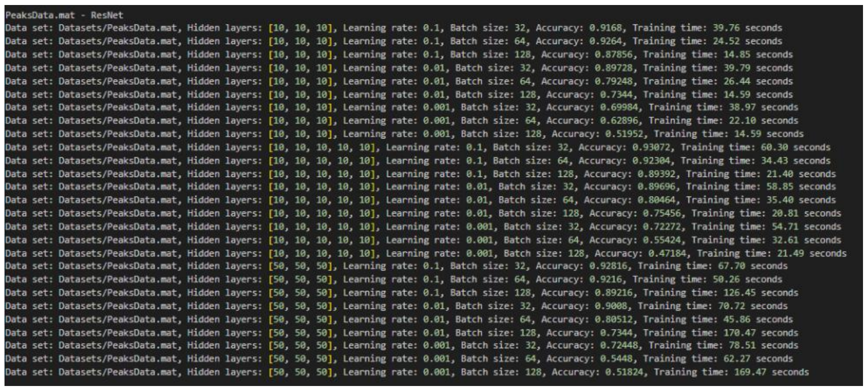
```
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.1, Batch size: 32, Accuracy: 0.96576, Training time: 39.60 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.1, Batch size: 64, Accuracy: 0.96352, Training time: 22.39 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.1, Batch size: 128, Accuracy: 0.94608, Training time: 14.67 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.01, Batch size: 32, Accuracy: 0.94096, Training time: 41.51 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.01, Batch size: 64, Accuracy: 0.89424, Training time: 24.51 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.01, Batch size: 128, Accuracy: 0.75216, Training time: 15.95 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.001, Batch size: 32, Accuracy: 0.6464, Training time: 40.62 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.001, Batch size: 64, Accuracy: 0.51536, Training time: 25.75 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.001, Batch size: 128, Accuracy: 0.40848, Training time: 16.22 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.1, Batch size: 32, Accuracy: 0.96816, Training time: 63.45 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.1, Batch size: 64, Accuracy: 0.96624, Training time: 34.88 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.1, Batch size: 128, Accuracy: 0.94608, Training time: 22.37 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.01, Batch size: 32, Accuracy: 0.95408, Training time: 64.97 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.01, Batch size: 64, Accuracy: 0.89344, Training time: 45.25 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.01, Batch size: 128, Accuracy: 0.74608, Training time: 34.16 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.001, Batch size: 32, Accuracy: 0.6264, Training time: 68.14 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.001, Batch size: 64, Accuracy: 0.4224, Training time: 45.08 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.001, Batch size: 128, Accuracy: 0.44, Training time: 26.37 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.1, Batch size: 32, Accuracy: 0.9712, Training time: 70.50 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.1, Batch size: 64, Accuracy: 0.97088, Training time: 48.20 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.1, Batch size: 128, Accuracy: 0.9608, Training time: 149.95 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.01, Batch size: 32, Accuracy: 0.96336, Training time: 76.57 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.01, Batch size: 64, Accuracy: 0.90576, Training time: 47.25 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.01, Batch size: 128, Accuracy: 0.67376, Training time: 123.76 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.001, Batch size: 32, Accuracy: 0.61792, Training time: 90.66 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.001, Batch size: 64, Accuracy: 0.53744, Training time: 59.01 seconds
Data set: Datasets/GMMData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.001, Batch size: 128, Accuracy: 0.44848, Training time: 168.07 seconds
```

Fig 34: *GMMData with ResNet*



```
Data set: Datasets/PeaksData.mat, Hidden layers: [], Learning rate: 0.1, Batch size: 32, Accuracy: 0.55168, Training time: 9.17 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [], Learning rate: 0.1, Batch size: 64, Accuracy: 0.56544, Training time: 5.55 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [], Learning rate: 0.1, Batch size: 128, Accuracy: 0.5592, Training time: 4.16 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [], Learning rate: 0.01, Batch size: 32, Accuracy: 0.56864, Training time: 10.66 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [], Learning rate: 0.01, Batch size: 64, Accuracy: 0.5656, Training time: 5.48 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [], Learning rate: 0.01, Batch size: 128, Accuracy: 0.56608, Training time: 3.81 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [], Learning rate: 0.001, Batch size: 32, Accuracy: 0.56592, Training time: 9.54 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [], Learning rate: 0.001, Batch size: 64, Accuracy: 0.56832, Training time: 5.54 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [], Learning rate: 0.001, Batch size: 128, Accuracy: 0.57104, Training time: 3.64 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10], Learning rate: 0.1, Batch size: 32, Accuracy: 0.86064, Training time: 18.88 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10], Learning rate: 0.1, Batch size: 64, Accuracy: 0.87344, Training time: 10.88 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10], Learning rate: 0.1, Batch size: 128, Accuracy: 0.8304, Training time: 6.56 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10], Learning rate: 0.01, Batch size: 32, Accuracy: 0.81328, Training time: 17.11 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10], Learning rate: 0.01, Batch size: 64, Accuracy: 0.77552, Training time: 9.79 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10], Learning rate: 0.01, Batch size: 128, Accuracy: 0.74336, Training time: 6.10 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10], Learning rate: 0.001, Batch size: 32, Accuracy: 0.6928, Training time: 17.21 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10], Learning rate: 0.001, Batch size: 64, Accuracy: 0.56432, Training time: 9.79 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10], Learning rate: 0.001, Batch size: 128, Accuracy: 0.56736, Training time: 6.15 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.1, Batch size: 32, Accuracy: 0.92608, Training time: 34.33 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.1, Batch size: 64, Accuracy: 0.90896, Training time: 18.43 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.1, Batch size: 128, Accuracy: 0.82064, Training time: 12.30 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.01, Batch size: 32, Accuracy: 0.85248, Training time: 34.53 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.01, Batch size: 64, Accuracy: 0.53728, Training time: 18.63 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.01, Batch size: 128, Accuracy: 0.45616, Training time: 12.07 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.001, Batch size: 32, Accuracy: 0.40256, Training time: 34.12 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.001, Batch size: 64, Accuracy: 0.40448, Training time: 18.25 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.001, Batch size: 128, Accuracy: 0.22976, Training time: 13.14 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.1, Batch size: 32, Accuracy: 0.92384, Training time: 48.66 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.1, Batch size: 64, Accuracy: 0.69584, Training time: 26.21 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.1, Batch size: 128, Accuracy: 0.5184, Training time: 16.42 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.01, Batch size: 32, Accuracy: 0.51968, Training time: 46.34 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.01, Batch size: 64, Accuracy: 0.21072, Training time: 26.74 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.01, Batch size: 128, Accuracy: 0.20048, Training time: 17.83 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.001, Batch size: 32, Accuracy: 0.20048, Training time: 45.97 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.001, Batch size: 64, Accuracy: 0.20048, Training time: 25.98 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.001, Batch size: 128, Accuracy: 0.20048, Training time: 17.79 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50], Learning rate: 0.1, Batch size: 32, Accuracy: 0.9256, Training time: 23.31 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50], Learning rate: 0.1, Batch size: 64, Accuracy: 0.89184, Training time: 14.33 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50], Learning rate: 0.1, Batch size: 128, Accuracy: 0.83472, Training time: 10.77 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50], Learning rate: 0.01, Batch size: 32, Accuracy: 0.82736, Training time: 22.09 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50], Learning rate: 0.01, Batch size: 64, Accuracy: 0.78464, Training time: 15.00 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50], Learning rate: 0.01, Batch size: 128, Accuracy: 0.72784, Training time: 12.17 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50], Learning rate: 0.001, Batch size: 32, Accuracy: 0.7072, Training time: 27.04 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50], Learning rate: 0.001, Batch size: 64, Accuracy: 0.54256, Training time: 14.13 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50], Learning rate: 0.001, Batch size: 128, Accuracy: 0.5152, Training time: 10.85 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.1, Batch size: 32, Accuracy: 0.93424, Training time: 55.37 seconds ---best result---
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.1, Batch size: 64, Accuracy: 0.92192, Training time: 35.68 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.1, Batch size: 128, Accuracy: 0.76336, Training time: 97.54 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.01, Batch size: 32, Accuracy: 0.79568, Training time: 53.32 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.01, Batch size: 64, Accuracy: 0.41488, Training time: 35.43 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.01, Batch size: 128, Accuracy: 0.20048, Training time: 97.57 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.001, Batch size: 32, Accuracy: 0.20048, Training time: 50.92 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.001, Batch size: 64, Accuracy: 0.20048, Training time: 32.82 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.001, Batch size: 128, Accuracy: 0.20048, Training time: 107.14 seconds
```

Fig 35: *PeaksData without ResNet*



```
PeaksData.mat - ResNet
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.1, Batch size: 32, Accuracy: 0.9168, Training time: 39.76 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.1, Batch size: 64, Accuracy: 0.9264, Training time: 24.52 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.1, Batch size: 128, Accuracy: 0.87856, Training time: 14.85 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.01, Batch size: 32, Accuracy: 0.89728, Training time: 39.79 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.01, Batch size: 64, Accuracy: 0.79248, Training time: 26.44 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.01, Batch size: 128, Accuracy: 0.7344, Training time: 14.59 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.001, Batch size: 32, Accuracy: 0.69984, Training time: 38.97 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.001, Batch size: 64, Accuracy: 0.62896, Training time: 22.10 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10], Learning rate: 0.001, Batch size: 128, Accuracy: 0.51952, Training time: 14.59 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.1, Batch size: 32, Accuracy: 0.93072, Training time: 60.30 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.1, Batch size: 64, Accuracy: 0.92304, Training time: 34.43 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.1, Batch size: 128, Accuracy: 0.89392, Training time: 21.40 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.01, Batch size: 32, Accuracy: 0.89696, Training time: 58.85 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.01, Batch size: 64, Accuracy: 0.80464, Training time: 35.40 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.01, Batch size: 128, Accuracy: 0.75456, Training time: 20.81 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.001, Batch size: 32, Accuracy: 0.72272, Training time: 54.71 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.001, Batch size: 64, Accuracy: 0.55424, Training time: 32.61 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [10, 10, 10, 10, 10], Learning rate: 0.001, Batch size: 128, Accuracy: 0.47184, Training time: 21.49 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.1, Batch size: 32, Accuracy: 0.92816, Training time: 67.70 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.1, Batch size: 64, Accuracy: 0.9216, Training time: 50.26 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.1, Batch size: 128, Accuracy: 0.89216, Training time: 126.45 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.01, Batch size: 32, Accuracy: 0.9008, Training time: 70.72 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.01, Batch size: 64, Accuracy: 0.80512, Training time: 45.86 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.01, Batch size: 128, Accuracy: 0.7344, Training time: 170.47 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.001, Batch size: 32, Accuracy: 0.72448, Training time: 78.51 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.001, Batch size: 64, Accuracy: 0.5448, Training time: 62.27 seconds
Data set: Datasets/PeaksData.mat, Hidden layers: [50, 50, 50], Learning rate: 0.001, Batch size: 128, Accuracy: 0.51824, Training time: 169.47 seconds
```

Fig 36: PeaksData with ResNet

The <u>**best**</u> results achieved for each dataset are listed below:

- <u>GMMData</u>
  - Validation accuracy: 0. 97104
  - Lr: 0.1
  - Mini-batch size: 32
  - Hidden layers: [50, 50, 50]
  - Runtime: 53.79
  - Resnet: No

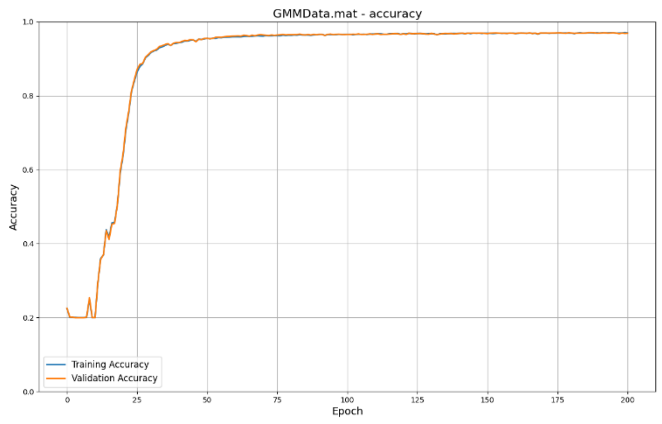


Fig 37: Training and Validation Accuracies

- <u>PeaksData</u>
  - Validation accuracy: 0. 93424
  - Lr: 0.1
  - Mini-batch size: 32
  - Hidden layers: [50, 50, 50]
  - Runtime: 30.73
  - Resnet: No

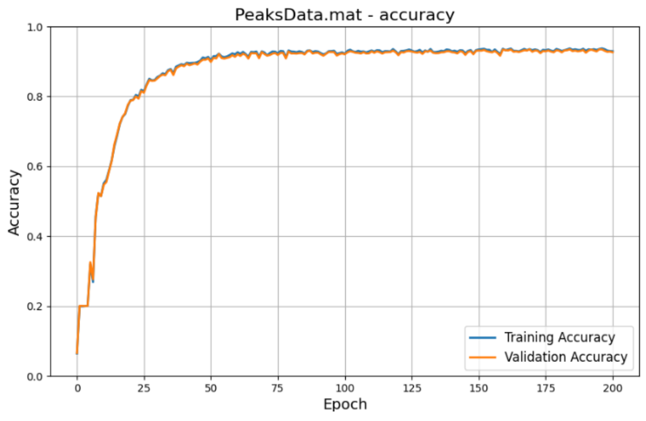


Fig 38: Training and Validation accuracy

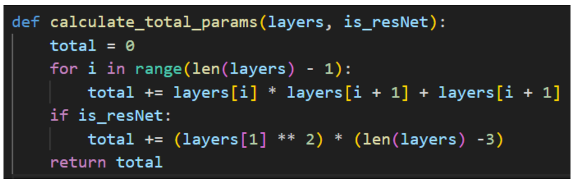
Conclusion

Learning Rate: The best accuracy was consistently achieved with a learning rate of 0.1, as smaller learning rates tended to make smaller adjustments by the gradient, resulting in poorer performance.
Batch Size: Smaller batch sizes resulted in better accuracy, but this came at the cost of increased training time. While using smaller batches led to more refined updates and higher performance, the trade-off was a slower runtime.
Hidden Layers: Deeper and narrower networks generally outperformed shallower or wider ones, offering better accuracy. However, networks that were too deep encountered vanishing gradient issues, which negatively impacted their performance. Therefore, a balanced architecture with an optimal number of layers provided the best results.
ResNet: ResNet consistently provided excellent performance across all configurations, though it came with the trade-off of slower training times.

- For the GMMData and PeaksData dataset, both ResNet and non-ResNet neural networks delivered strong results, but the non-ResNet model achieved better runtime efficiency.

2.5. Neural Network with constrained parameters

For computing the number of parameters of our NN we used a simple function which multiply the number of weights from each layer, and sums with the biases.
For a resnet NN, we can assume that all the hidden layers are from the same size, thus we power the size of the first hidden layer by 2, times the number of hidden layers.



```python
def calculate_total_params(layers, is_resNet):
    total = 0
    for i in range(len(layers) - 1):
        total += layers[i] * layers[i + 1] + layers[i + 1]
    if is_resNet:
        total += (layers[1] ** 2) * (len(layers) -3)
    return total
```

Fig 39: The function for calculating the total parameters

Our strategy was to start from a shallow and wide network, and in each test we narrowed it and made it deeper (in consideration of the parameters constraint –for both PeaksData and GMMData we had a maximum of 100*5 = 500 parameters.

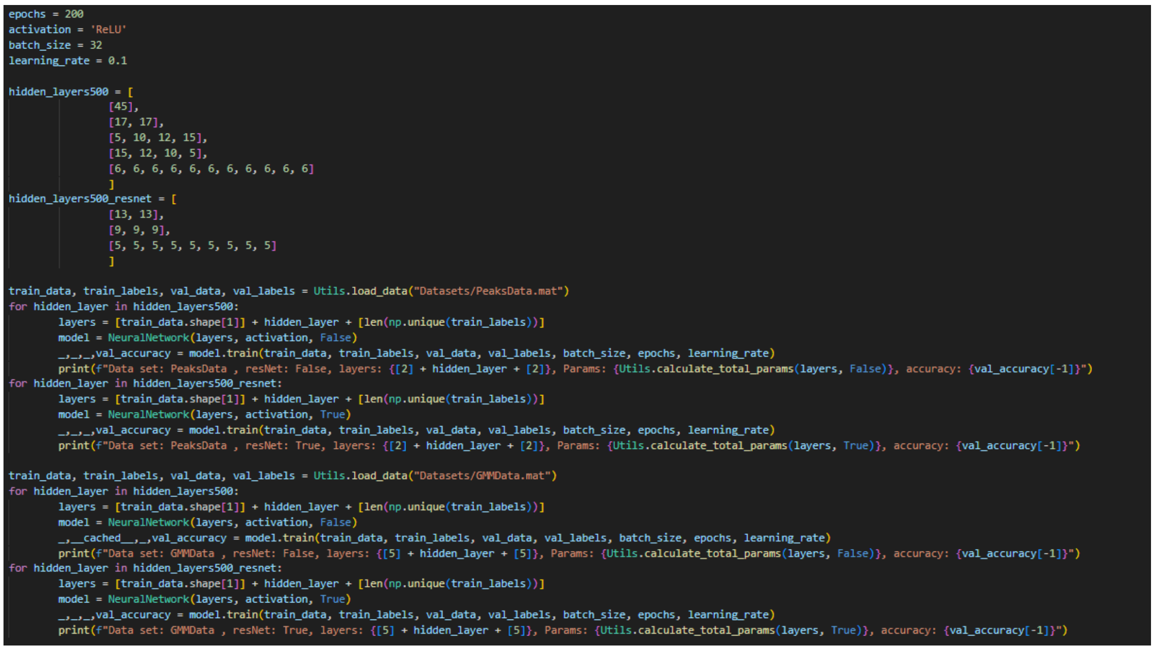
We conducted the test with the following parameters:

- Lr : 0.1
- Mini-batch size: 32
- Epochs: 200
- Activation function: ReLU

Architecture (500 parameters):

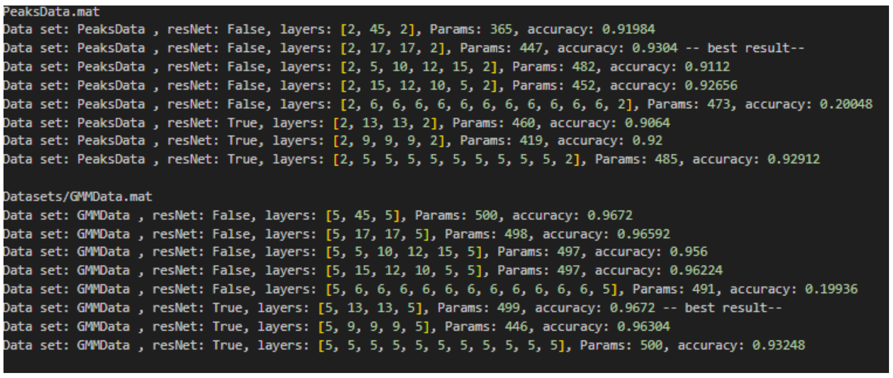
- Non resnet hidden layers:
  - [45]
  - [17,17]
  - [5,10,12,15]
  - [15,12,10,5]
  - [6,6,6,6,6,6,6,6,6,6,6]
- Resnet hidden layers:
  - [13,13]
  - [9,9,9]
  - [5,5,5,5,5,5,5,5,5]

*The reason for the architecture of the 3rd and 4th layer options for non ResNet networks is to check if the order of the hidden layers effects the accuracy.



```
epochs = 200
activation = 'ReLU'
batch_size = 32
learning_rate = 0.1

hidden_layers500 = [
                [45],
                [17, 17],
                [5, 10, 12, 15],
                [15, 12, 10, 5],
                [6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6]
                ]
hidden_layers500_resnet = [
                [13, 13],
                [9, 9, 9],
                [5, 5, 5, 5, 5, 5, 5, 5, 5]
                ]

train_data, train_labels, val_data, val_labels = Utils.load_data("Datasets/PeaksData.mat")
for hidden_layer in hidden_layers500:
        layers = [train_data.shape[1]] + hidden_layer + [len(np.unique(train_labels))]
        model = NeuralNetwork(layers, activation, False)
        _,_,_,val_accuracy = model.train(train_data, train_labels, val_data, val_labels, batch_size, epochs, learning_rate)
        print(f"Data set: PeaksData , resNet: False, layers: {[2] + hidden_layer + [2]}, Params: {Utils.calculate_total_params(layers, False)}, accuracy: {val_accuracy[-1]}")
for hidden_layer in hidden_layers500_resnet:
        layers = [train_data.shape[1]] + hidden_layer + [len(np.unique(train_labels))]
        model = NeuralNetwork(layers, activation, True)
        _,_,_,val_accuracy = model.train(train_data, train_labels, val_data, val_labels, batch_size, epochs, learning_rate)
        print(f"Data set: PeaksData , resNet: True, layers: {[2] + hidden_layer + [2]}, Params: {Utils.calculate_total_params(layers, True)}, accuracy: {val_accuracy[-1]}")

train_data, train_labels, val_data, val_labels = Utils.load_data("Datasets/GMMData.mat")
for hidden_layer in hidden_layers500:
        layers = [train_data.shape[1]] + hidden_layer + [len(np.unique(train_labels))]
        model = NeuralNetwork(layers, activation, False)
        _,__cached__,_,val_accuracy = model.train(train_data, train_labels, val_data, val_labels, batch_size, epochs, learning_rate)
        print(f"Data set: GMMData , resNet: False, layers: {[5] + hidden_layer + [5]}, Params: {Utils.calculate_total_params(layers, False)}, accuracy: {val_accuracy[-1]}")
for hidden_layer in hidden_layers500_resnet:
        layers = [train_data.shape[1]] + hidden_layer + [len(np.unique(train_labels))]
        model = NeuralNetwork(layers, activation, True)
        _,_,_,val_accuracy = model.train(train_data, train_labels, val_data, val_labels, batch_size, epochs, learning_rate)
        print(f"Data set: GMMData , resNet: True, layers: {[5] + hidden_layer + [5]}, Params: {Utils.calculate_total_params(layers, True)}, accuracy: {val_accuracy[-1]}")
```

Fig 40: code implementation for running NN with parameters constraints



```
PeaksData.mat
Data set: PeaksData , resNet: False, layers: [2, 45, 2], Params: 365, accuracy: 0.91984
Data set: PeaksData , resNet: False, layers: [2, 17, 17, 2], Params: 447, accuracy: 0.9304 -- best result--
Data set: PeaksData , resNet: False, layers: [2, 5, 10, 12, 15, 2], Params: 482, accuracy: 0.9112
Data set: PeaksData , resNet: False, layers: [2, 15, 12, 10, 5, 2], Params: 452, accuracy: 0.92656
Data set: PeaksData , resNet: False, layers: [2, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 2], Params: 473, accuracy: 0.20048
Data set: PeaksData , resNet: True, layers: [2, 13, 13, 2], Params: 460, accuracy: 0.9064
Data set: PeaksData , resNet: True, layers: [2, 9, 9, 9, 2], Params: 419, accuracy: 0.92
Data set: PeaksData , resNet: True, layers: [2, 5, 5, 5, 5, 5, 5, 5, 5, 5, 2], Params: 485, accuracy: 0.92912

Datasets/GMMData.mat
Data set: GMMData , resNet: False, layers: [5, 45, 5], Params: 500, accuracy: 0.9672
Data set: GMMData , resNet: False, layers: [5, 17, 17, 5], Params: 498, accuracy: 0.96592
Data set: GMMData , resNet: False, layers: [5, 5, 10, 12, 15, 5], Params: 497, accuracy: 0.956
Data set: GMMData , resNet: False, layers: [5, 15, 12, 10, 5, 5], Params: 497, accuracy: 0.96224
Data set: GMMData , resNet: False, layers: [5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5], Params: 491, accuracy: 0.19936
Data set: GMMData , resNet: True, layers: [5, 13, 13, 5], Params: 499, accuracy: 0.9672 -- best result--
Data set: GMMData , resNet: True, layers: [5, 9, 9, 9, 5], Params: 446, accuracy: 0.96304
Data set: GMMData , resNet: True, layers: [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5], Params: 500, accuracy: 0.93248
```

Fig 41: Results given by running Fig 40

Best Results:

- PeaksData
  - Validation accuracy: 0. 9304
  - Hidden layers: [17, 17]
  - Params: 447
  - Resnet: No

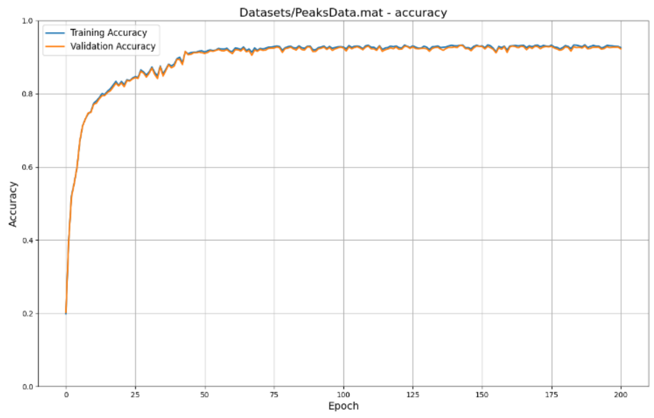


Fig 42: Training and Validation Accuracies

- GMMData
  - Validation accuracy: 0. 9672
  - Hidden layers: [13, 13]
  - Params: 499
  - Resnet: Yes

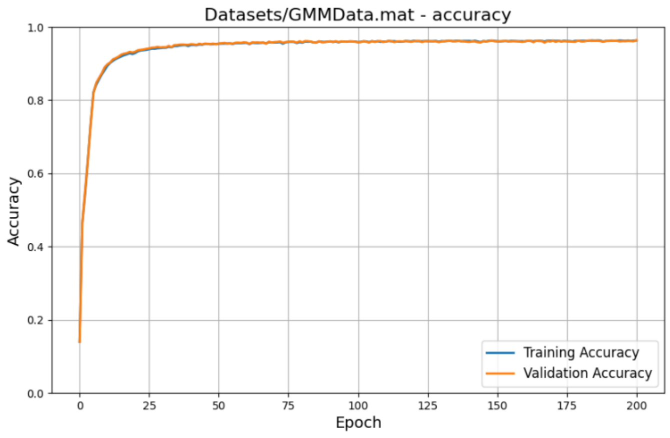


Fig 43: Training and Validation Accuracies

Conclusion:

Under parameter constraints:

- **Shallow and wide networks generally outperform deep and narrow networks**, as they are better at utilizing limited parameters to represent complex functions.
- **Both ResNet and non-ResNet architectures achieved same results,** though it is worth mentioning that when trained with deep and narrow network, due to the vanishing gradients problem, ResNet achieved much better results.