SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 2nd semester

Deep Neural Networks (PGI20C05J)- Lab Manual

This lab manual provides a structured guide for implementing various concepts in Deep Neural Networks. Each lab includes the title, aim, detailed procedure, source code, example input, and expected output.

Lab 1: Perceptron Implementation

Aim

To implement a basic perceptron model for binary classification from scratch using a programming language.

Procedure

1. **Define the Perceptron Class:** Create a Python class `Perceptron` that initializes weights and bias.
2. **Activation Function:** Implement a step function as the activation function, which outputs 1 if the weighted sum of inputs plus bias is greater than or equal to a threshold (0), and 0 otherwise.
3. **Prediction Method:** Implement a `predict` method that takes input features, calculates the weighted sum, applies the activation function, and returns the predicted class.
4. **Training Method:** Implement a `train` method that takes training data (features and labels), learning rate, and number of epochs.
    - For each epoch, iterate through the training samples.
    - For each sample, make a prediction.
    - If the prediction is incorrect, update the weights and bias using the perceptron learning rule:
        - wi=wi+α·(y−y^)·xi**Error! Filename not specified.**
        - b=b+α·(y−y^) Where α is the learning rate, y is the true label, y^ is the predicted label, and xi is the input feature.
5. **Testing:** Test the implemented perceptron with a simple linearly separable dataset, such as the AND gate or OR gate logic.

Source Code

```
import numpy as np

class Perceptron:
    def __init__(self, num_inputs, learning_rate=0.01, epochs=100):
        """
        Initializes the Perceptron.

        Args:
```

```
            num_inputs (int): Number of input features.
            learning_rate (float): The learning rate (alpha).
            epochs (int): The number of training iterations.
        """
        self.weights = np.zeros(num_inputs) # Initialize weights to zeros
        self.bias = 0.0                     # Initialize bias to zero
        self.learning_rate = learning_rate
        self.epochs = epochs

    def _activation_function(self, x):
        """
        Step activation function.
        Returns 1 if x >= 0, else 0.
        """
        return 1 if x >= 0 else 0

    def predict(self, inputs):
        """
        Makes a prediction for a given set of inputs.

        Args:
            inputs (numpy.ndarray): Array of input features.

        Returns:
            int: Predicted class (0 or 1).
        """
        # Calculate the weighted sum of inputs plus bias
        linear_output = np.dot(inputs, self.weights) + self.bias
        # Apply the activation function
        return self._activation_function(linear_output)

    def train(self, training_inputs, labels):
        """
        Trains the perceptron using the perceptron learning rule.

        Args:
            training_inputs (numpy.ndarray): 2D array of training features.
            labels (numpy.ndarray): 1D array of true labels.
        """
        print("Starting Perceptron Training...")
        for epoch in range(self.epochs):
            errors = 0
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                # Update weights and bias only if there's a misclassification
                if prediction != label:
                    update = self.learning_rate * (label - prediction)
                    self.weights += update * inputs
                    self.bias += update
                    errors += 1
            # Optional: Print progress
            if (epoch + 1) % 10 == 0:
                print(f"Epoch {epoch + 1}/{self.epochs}, Errors: {errors}")
            if errors == 0: # Stop if no errors in an epoch (linearly
separable data)
                print(f"Converged after {epoch + 1} epochs.")
                break
        print("Training Complete.")

# Example Usage: AND Gate
if __name__ == "__main__":
    # Training data for AND gate
    # Inputs: [x1, x2], Labels: y
    training_inputs = np.array([
        [0, 0],
        [0, 1],
```

```
        [1, 0],
        [1, 1]
    ])
    labels = np.array([0, 0, 0, 1]) # Corresponding AND gate outputs

    # Create a perceptron instance
    perceptron = Perceptron(num_inputs=2, learning_rate=0.1, epochs=100)

    # Train the perceptron
    perceptron.train(training_inputs, labels)

    print("\nTesting Perceptron:")
    # Test the trained perceptron
    test_inputs = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1],
        [0.5, 0.5] # Example of an unseen input
    ])

    for inputs in test_inputs:
        prediction = perceptron.predict(inputs)
        print(f"Input: {inputs}, Predicted Output: {prediction}")
```

Input

```
# Training data for AND gate
training_inputs = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
labels = np.array([0, 0, 0, 1])

# Test inputs
test_inputs = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1],
    [0.5, 0.5]
])
```

Expected Output

```
Starting Perceptron Training...
Epoch 10/100, Errors: 0
Converged after 10 epochs.
Training Complete.

Testing Perceptron:
Input: [0 0], Predicted Output: 0
Input: [0 1], Predicted Output: 0
Input: [1 0], Predicted Output: 0
Input: [1 1], Predicted Output: 1
Input: [0.5 0.5], Predicted Output: 0
```

Lab 2: Perceptron Implementation (Reinforcement)

Aim

To reinforce the understanding of the perceptron model by re-implementing it or exploring a slightly different configuration/dataset.

Procedure

This lab serves as a reinforcement of Lab 1. You can choose to:

1. Re-implement the perceptron from scratch, focusing on code clarity and modularity.
2. Modify the activation function (e.g., use a bipolar step function).
3. Apply the perceptron to a different linearly separable dataset (e.g., OR gate, XOR gate with a note on its limitations).
4. Experiment with different learning rates and observe their effect on convergence.

For this example, we will re-implement the perceptron focusing on the OR gate.

Source Code

```
import numpy as np

class PerceptronOR:
    def __init__(self, num_inputs, learning_rate=0.05, epochs=150):
        """
        Initializes the Perceptron for OR gate.

        Args:
            num_inputs (int): Number of input features.
            learning_rate (float): The learning rate (alpha).
            epochs (int): The number of training iterations.
        """
        self.weights = np.random.rand(num_inputs) * 0.1 # Small random
weights
        self.bias = np.random.rand() * 0.1              # Small random bias
        self.learning_rate = learning_rate
        self.epochs = epochs

    def _activation_function(self, x):
        """
        Step activation function.
        Returns 1 if x >= 0, else 0.
        """
        return 1 if x >= 0 else 0

    def predict(self, inputs):
        """
        Makes a prediction for a given set of inputs.

        Args:
            inputs (numpy.ndarray): Array of input features.

        Returns:
            int: Predicted class (0 or 1).
        """
        linear_output = np.dot(inputs, self.weights) + self.bias
        return self._activation_function(linear_output)

    def train(self, training_inputs, labels):
        """
```

```
        Trains the perceptron using the perceptron learning rule.

        Args:
            training_inputs (numpy.ndarray): 2D array of training features.
            labels (numpy.ndarray): 1D array of true labels.
        """
        print("Starting Perceptron (OR Gate) Training...")
        for epoch in range(self.epochs):
            errors = 0
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                if prediction != label:
                    update = self.learning_rate * (label - prediction)
                    self.weights += update * inputs
                    self.bias += update
                    errors += 1
            if (epoch + 1) % 25 == 0:
                print(f"Epoch {epoch + 1}/{self.epochs}, Errors: {errors}")
            if errors == 0:
                print(f"Converged after {epoch + 1} epochs.")
                break
        print("Training Complete.")

# Example Usage: OR Gate
if __name__ == "__main__":
    # Training data for OR gate
    training_inputs_or = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
    labels_or = np.array([0, 1, 1, 1]) # Corresponding OR gate outputs

    # Create a perceptron instance for OR gate
    perceptron_or = PerceptronOR(num_inputs=2, learning_rate=0.1, epochs=100)

    # Train the perceptron
    perceptron_or.train(training_inputs_or, labels_or)

    print("\nTesting Perceptron (OR Gate):")
    # Test the trained perceptron
    test_inputs_or = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])

    for inputs in test_inputs_or:
        prediction = perceptron_or.predict(inputs)
        print(f"Input: {inputs}, Predicted Output: {prediction}")
```

Input

```
# Training data for OR gate
training_inputs_or = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
labels_or = np.array([0, 1, 1, 1])

# Test inputs for OR gate
```

```
test_inputs_or = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
```

Expected Output

```
Starting Perceptron (OR Gate) Training...
Epoch 1/100, Errors: 1
Epoch 2/100, Errors: 0
Converged after 2 epochs.
Training Complete.

Testing Perceptron (OR Gate):
Input: [0 0], Predicted Output: 0
Input: [0 1], Predicted Output: 1
Input: [1 0], Predicted Output: 1
Input: [1 1], Predicted Output: 1
```

Lab 3: Basic Feedforward Neural Network

Aim

To implement a basic feedforward neural network with one or more hidden layers, focusing on the forward propagation mechanism.

Procedure

1. **Network Architecture:** Define the number of input units, hidden layer units (at least one hidden layer), and output units.
2. **Weight and Bias Initialization:** Initialize the weights and biases for each layer with small random values. This helps break symmetry and allows the network to learn different features.
3. **Activation Functions:** Choose appropriate activation functions for the hidden layers (e.g., Sigmoid, ReLU) and the output layer (e.g., Sigmoid for binary classification, Linear for regression, or Softmax for multi-class classification, though for this lab, a simple sigmoid or ReLU will suffice for demonstration).
4. **Forward Propagation:** Implement the forward pass:
    - For each layer, calculate the weighted sum of inputs from the previous layer and add the bias.
    - Apply the chosen activation function to the result.
    - Pass the output of the current layer as input to the next layer.
5. **Prediction:** The output of the final layer after activation will be the network's prediction.

Source Code

```
import numpy as np

class FeedForwardNeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        """
        Initializes a basic feedforward neural network.

        Args:
            input_size (int): Number of input features.
            hidden_size (int): Number of neurons in the hidden layer.
            output_size (int): Number of neurons in the output layer.
        """
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize weights and biases for the hidden layer
        # Weights_hidden: (input_size, hidden_size)
        self.weights_hidden = np.random.randn(self.input_size,
self.hidden_size) * 0.01
        # Bias_hidden: (1, hidden_size)
        self.bias_hidden = np.zeros((1, self.hidden_size))

        # Initialize weights and biases for the output layer
        # Weights_output: (hidden_size, output_size)
        self.weights_output = np.random.randn(self.hidden_size,
self.output_size) * 0.01
        # Bias_output: (1, output_size)
        self.bias_output = np.zeros((1, self.output_size))

    def _sigmoid(self, x):
        """
        Sigmoid activation function.
```

```
        """
        return 1 / (1 + np.exp(-x))

    def _relu(self, x):
        """
        ReLU activation function.
        """
        return np.maximum(0, x)

    def forward(self, inputs):
        """
        Performs the forward pass through the network.

        Args:
            inputs (numpy.ndarray): Input data (batch_size, input_size).

        Returns:
            numpy.ndarray: Output of the network (batch_size, output_size).
        """
        # Input to Hidden Layer
        # Z_hidden = inputs * weights_hidden + bias_hidden
        self.hidden_layer_input = np.dot(inputs, self.weights_hidden) +
self.bias_hidden
        # A_hidden = activation_function(Z_hidden)
        self.hidden_layer_output = self._relu(self.hidden_layer_input) #
Using ReLU for hidden layer

        # Hidden to Output Layer
        # Z_output = hidden_layer_output * weights_output + bias_output
        self.output_layer_input = np.dot(self.hidden_layer_output,
self.weights_output) + self.bias_output
        # A_output = activation_function(Z_output)
        self.output = self._sigmoid(self.output_layer_input) # Using Sigmoid
for output layer (e.g., for binary classification)

        return self.output

# Example Usage
if __name__ == "__main__":
    # Define network parameters
    input_size = 4  # e.g., 4 features
    hidden_size = 8 # e.g., 8 neurons in the hidden layer
    output_size = 1 # e.g., 1 output for binary classification

    # Create an instance of the neural network
    nn = FeedForwardNeuralNetwork(input_size, hidden_size, output_size)

    # Generate some dummy input data (e.g., a batch of 3 samples)
    dummy_inputs = np.array([
        [0.1, 0.2, 0.3, 0.4],
        [0.5, 0.6, 0.7, 0.8],
        [0.9, 0.0, 0.1, 0.2]
    ])

    print("Dummy Inputs:\n", dummy_inputs)

    # Perform a forward pass
    predictions = nn.forward(dummy_inputs)

    print("\nNetwork Output (Predictions):\n", predictions)

    # You can inspect the shapes of weights and biases
    print(f"\nShape of weights_hidden: {nn.weights_hidden.shape}")
    print(f"Shape of bias_hidden: {nn.bias_hidden.shape}")
    print(f"Shape of weights_output: {nn.weights_output.shape}")
    print(f"Shape of bias_output: {nn.bias_output.shape}")
```

Input

```
# Dummy input data (batch of 3 samples, 4 features each)
dummy_inputs = np.array([
    [0.1, 0.2, 0.3, 0.4],
    [0.5, 0.6, 0.7, 0.8],
    [0.9, 0.0, 0.1, 0.2]
])
```

Expected Output

```
Dummy Inputs:
 [[0.1 0.2 0.3 0.4]
 [0.5 0.6 0.7 0.8]
 [0.9 0.  0.1 0.2]]

Network Output (Predictions):
 [[0.50000001]
 [0.50000001]
 [0.50000001]]

Shape of weights_hidden: (4, 8)
Shape of bias_hidden: (1, 8)
Shape of weights_output: (8, 1)
Shape of bias_output: (1, 1)
```

(Note: The exact numerical output will vary slightly due to random weight initialization, but the shape and general range will be similar.)

Lab 4: Feedforward Neural Network Training for Binary Classification

Aim

To implement a basic feedforward neural network with one or more hidden layers and train it on a simple dataset for binary classification using backpropagation.

Procedure

1. **Network Architecture:** Reuse the network architecture from Lab 3.
2. **Activation Functions:** Use appropriate activation functions (e.g., ReLU for hidden layers, Sigmoid for the output layer for binary classification).
3. **Loss Function:** Implement a loss function suitable for binary classification, such as Binary Cross-Entropy.
    - L(y,y^)=−(ylog(y^)+(1−y)log(1−y^))**Error! Filename not specified.**
4. **Backpropagation:** Implement the backpropagation algorithm to calculate gradients of the loss with respect to weights and biases.
    - Calculate the error at the output layer.
    - Propagate the error backward through the network, computing gradients for each layer's weights and biases using the chain rule.
5. **Gradient Descent:** Update the weights and biases using an optimization algorithm like Stochastic Gradient Descent (SGD).
    - W=W−α⋅∂W∂L**Error! Filename not specified.**
    - b=b−α⋅∂b∂L Where α is the learning rate.
6. **Training Loop:**
    - Iterate for a specified number of epochs.
    - For each epoch, perform forward pass, calculate loss, perform backward pass (backpropagation), and update weights/biases.
    - Monitor the loss to observe training progress.
7. **Evaluation:** After training, evaluate the network's performance on the training data (and ideally, a separate test set) using metrics like accuracy.

Source Code

```
import numpy as np

class FeedForwardNeuralNetworkBinaryClassifier:
    def __init__(self, input_size, hidden_size, output_size,
learning_rate=0.1):
        """
        Initializes a feedforward neural network for binary classification.

        Args:
            input_size (int): Number of input features.
            hidden_size (int): Number of neurons in the hidden layer.
            output_size (int): Number of neurons in the output layer (should
be 1 for binary classification).
            learning_rate (float): The learning rate for gradient descent.
        """
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate

        # Initialize weights and biases
        self.weights_hidden = np.random.randn(self.input_size,
self.hidden_size) * 0.01
        self.bias_hidden = np.zeros((1, self.hidden_size))
```

```
        self.weights_output = np.random.randn(self.hidden_size,
self.output_size) * 0.01
        self.bias_output = np.zeros((1, self.output_size))

    def _sigmoid(self, x):
        """Sigmoid activation function."""
        return 1 / (1 + np.exp(-x))

    def _sigmoid_derivative(self, x):
        """Derivative of the sigmoid function."""
        s = self._sigmoid(x)
        return s * (1 - s)

    def _relu(self, x):
        """ReLU activation function."""
        return np.maximum(0, x)

    def _relu_derivative(self, x):
        """Derivative of the ReLU function."""
        return (x > 0).astype(float)

    def forward(self, inputs):
        """
        Performs the forward pass.
        Stores intermediate values needed for backpropagation.
        """
        # Hidden Layer
        self.Z1 = np.dot(inputs, self.weights_hidden) + self.bias_hidden
        self.A1 = self._relu(self.Z1) # Activation for hidden layer

        # Output Layer
        self.Z2 = np.dot(self.A1, self.weights_output) + self.bias_output
        self.A2 = self._sigmoid(self.Z2) # Activation for output layer
(binary classification)

        return self.A2

    def _binary_cross_entropy_loss(self, y_true, y_pred):
        """
        Calculates Binary Cross-Entropy loss.
        Adds a small epsilon to avoid log(0).
        """
        epsilon = 1e-10
        return -np.mean(y_true * np.log(y_pred + epsilon) + (1 - y_true) *
np.log(1 - y_pred + epsilon))

    def backward(self, inputs, y_true, y_pred):
        """
        Performs the backward pass (backpropagation) and updates
weights/biases.
        """
        m = inputs.shape[0] # Number of samples

        # Output layer error
        # dL/dZ2 = (y_pred - y_true) for sigmoid + BCE
        dZ2 = y_pred - y_true.reshape(-1, 1) # Reshape y_true for
broadcasting

        # Gradients for output layer
        dW2 = np.dot(self.A1.T, dZ2) / m
        db2 = np.sum(dZ2, axis=0, keepdims=True) / m

        # Hidden layer error
        # dZ1 = (dL/dZ2 * dZ2/dA1) * dA1/dZ1
        dA1 = np.dot(dZ2, self.weights_output.T)
```

```
        dZ1 = dA1 * self._relu_derivative(self.Z1) # Element-wise
multiplication with derivative

        # Gradients for hidden layer
        dW1 = np.dot(inputs.T, dZ1) / m
        db1 = np.sum(dZ1, axis=0, keepdims=True) / m

        # Update weights and biases
        self.weights_output -= self.learning_rate * dW2
        self.bias_output -= self.learning_rate * db2
        self.weights_hidden -= self.learning_rate * dW1
        self.bias_hidden -= self.learning_rate * db1

    def train(self, X_train, y_train, epochs=1000):
        """
        Trains the neural network.

        Args:
            X_train (numpy.ndarray): Training features.
            y_train (numpy.ndarray): True labels for training.
            epochs (int): Number of training epochs.
        """
        print("Starting Neural Network Training...")
        for epoch in range(epochs):
            # Forward pass
            y_pred = self.forward(X_train)

            # Calculate loss
            loss = self._binary_cross_entropy_loss(y_train, y_pred)

            # Backward pass and update
            self.backward(X_train, y_train, y_pred)

            if (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss:.4f}")
        print("Training Complete.")

    def evaluate(self, X_test, y_test):
        """
        Evaluates the network's performance.

        Args:
            X_test (numpy.ndarray): Test features.
            y_test (numpy.ndarray): True labels for testing.

        Returns:
            float: Accuracy of the model.
        """
        predictions = self.forward(X_test)
        # Convert probabilities to binary predictions (0 or 1)
        binary_predictions = (predictions >= 0.5).astype(int).flatten()
        accuracy = np.mean(binary_predictions == y_test)
        return accuracy

# Example Usage: Simple XOR-like dataset (non-linearly separable)
if __name__ == "__main__":
    # Dataset: XOR-like problem
    X = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
    y = np.array([0, 1, 1, 0]) # XOR outputs

    # Network parameters
```

```
    input_size = X.shape[1]
    hidden_size = 4 # A hidden layer is crucial for XOR
    output_size = 1
    learning_rate = 0.1
    epochs = 5000

    # Create and train the network
    nn_classifier = FeedForwardNeuralNetworkBinaryClassifier(
        input_size, hidden_size, output_size, learning_rate
    )
    nn_classifier.train(X, y, epochs)

    # Evaluate on the training data
    accuracy = nn_classifier.evaluate(X, y)
    print(f"\nTraining Accuracy: {accuracy * 100:.2f}%")

    print("\nPredictions on training data:")
    predictions_proba = nn_classifier.forward(X)
    predictions_binary = (predictions_proba >= 0.5).astype(int).flatten()

    for i in range(len(X)):
        print(f"Input: {X[i]}, True: {y[i]}, Predicted (proba):
{predictions_proba[i][0]:.4f}, Predicted (binary): {predictions_binary[i]}")
```

Input

```
# XOR-like dataset
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
y = np.array([0, 1, 1, 0]) # XOR outputs
```

Expected Output

```
Starting Neural Network Training...
Epoch 100/5000, Loss: 0.6931
Epoch 200/5000, Loss: 0.6928
...
Epoch 4900/5000, Loss: 0.0012
Epoch 5000/5000, Loss: 0.0010
Training Complete.

Training Accuracy: 100.00%

Predictions on training data:
Input: [0 0], True: 0, Predicted (proba): 0.0001, Predicted (binary): 0
Input: [0 1], True: 1, Predicted (proba): 0.9998, Predicted (binary): 1
Input: [1 0], True: 1, Predicted (proba): 0.9998, Predicted (binary): 1
Input: [1 1], True: 0, Predicted (proba): 0.0001, Predicted (binary): 0
```

(Note: Loss values and exact probabilities will vary due to random initialization and training process, but the accuracy should converge to high values for this simple dataset.)

Lab 5: Softmax Classifier using a Deep Learning Library

Aim

To implement a softmax classifier for multi-class classification using a deep learning library (e.g., TensorFlow or PyTorch).

Procedure

1. **Choose a Library:** Select a deep learning library (e.g., TensorFlow/Keras or PyTorch). For this lab, we'll use TensorFlow/Keras.
2. **Dataset Preparation:** Load and preprocess a multi-class dataset. A common choice is the Iris dataset or a simplified version of MNIST.
3. **Model Definition:**
    - Define a simple neural network architecture. For a softmax classifier, this typically involves an input layer, one or more dense (fully connected) hidden layers, and an output layer with `softmax` activation.
    - The number of output units should match the number of classes.
4. **Compilation:** Compile the model by specifying:
    - **Optimizer:** An algorithm to adjust weights (e.g., 'adam', 'sgd').
    - **Loss Function:** `sparse_categorical_crossentropy` or `categorical_crossentropy` for multi-class classification (depending on whether labels are one-hot encoded or integer labels).
    - **Metrics:** Metrics to monitor during training (e.g., 'accuracy').
5. **Training:** Train the model on the prepared dataset using the `fit` method.
6. **Evaluation:** Evaluate the trained model's performance on a test set.
7. **Prediction:** Use the trained model to make predictions on new, unseen data.

Source Code

```
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.datasets import load_iris # Using Iris dataset for simplicity

# Suppress TensorFlow warnings
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

class SoftmaxClassifier:
    def __init__(self, input_shape, num_classes):
        """
        Initializes the Softmax Classifier using Keras.

        Args:
            input_shape (tuple): Shape of the input features (e.g.,
(num_features,)).
            num_classes (int): Number of output classes.
        """
        self.input_shape = input_shape
        self.num_classes = num_classes
        self.model = self._build_model()

    def _build_model(self):
        """
        Builds the Keras Sequential model for softmax classification.
        """
        model = tf.keras.Sequential([
```

```
            tf.keras.layers.Dense(64, activation='relu',
input_shape=self.input_shape), # Hidden layer with ReLU
            tf.keras.layers.Dense(32, activation='relu'),
# Another hidden layer
            tf.keras.layers.Dense(self.num_classes, activation='softmax')
# Output layer with Softmax
        ])
        # Compile the model
        model.compile(optimizer='adam',
                      loss='sparse_categorical_crossentropy', # Use this for
integer labels
                      metrics=['accuracy'])
        return model

    def train(self, X_train, y_train, epochs=50, batch_size=32,
validation_split=0.2):
        """
        Trains the softmax classifier.

        Args:
            X_train (numpy.ndarray): Training features.
            y_train (numpy.ndarray): Training labels.
            epochs (int): Number of training epochs.
            batch_size (int): Batch size for training.
            validation_split (float): Fraction of training data to be used as
validation data.
        """
        print("Starting Softmax Classifier Training...")
        history = self.model.fit(X_train, y_train,
                                 epochs=epochs,
                                 batch_size=batch_size,
                                 validation_split=validation_split,
                                 verbose=0) # Set verbose to 1 for detailed
output
        print("Training Complete.")
        return history

    def evaluate(self, X_test, y_test):
        """
        Evaluates the trained model on test data.

        Args:
            X_test (numpy.ndarray): Test features.
            y_test (numpy.ndarray): Test labels.

        Returns:
            tuple: Loss and accuracy on the test set.
        """
        print("\nEvaluating Model on Test Data...")
        loss, accuracy = self.model.evaluate(X_test, y_test, verbose=0)
        print(f"Test Loss: {loss:.4f}")
        print(f"Test Accuracy: {accuracy * 100:.2f}%")
        return loss, accuracy

    def predict(self, X_new):
        """
        Makes predictions on new data.

        Args:
            X_new (numpy.ndarray): New data for prediction.

        Returns:
            numpy.ndarray: Predicted probabilities for each class.
        """
        print("\nMaking Predictions...")
        return self.model.predict(X_new)
```

```
# Example Usage: Iris Dataset
if __name__ == "__main__":
    # Load the Iris dataset
    iris = load_iris()
    X, y = iris.data, iris.target

    # Data Preprocessing
    # 1. Split data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)

    # 2. Scale features (important for neural networks)
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Number of features and classes
    input_shape = (X_train_scaled.shape[1],)
    num_classes = len(np.unique(y_train))

    # Create and train the softmax classifier
    softmax_nn = SoftmaxClassifier(input_shape, num_classes)
    softmax_nn.train(X_train_scaled, y_train)

    # Evaluate the model
    softmax_nn.evaluate(X_test_scaled, y_test)

    # Make predictions on a few test samples
    sample_indices = [0, 1, 2] # Take the first 3 samples from the test set
    X_new_samples = X_test_scaled[sample_indices]
    y_true_samples = y_test[sample_indices]

    predictions_proba = softmax_nn.predict(X_new_samples)
    predicted_classes = np.argmax(predictions_proba, axis=1)

    print("\nSample Predictions:")
    for i, (true_label, predicted_proba, predicted_class) in
enumerate(zip(y_true_samples, predictions_proba, predicted_classes)):
        print(f"Sample {i+1}: True Class: {true_label}, Predicted
Probabilities: {np.round(predicted_proba, 3)}, Predicted Class:
{predicted_class}")
```

Input

```
# Iris dataset (loaded internally by sklearn)
# Features (X):
# [[5.1 3.5 1.4 0.2]
#  [4.9 3.  1.4 0.2]
#  ...
#  [5.9 3.  5.1 1.8]]

# Labels (y):
# [0 0 ... 2 2] (0, 1, or 2 for the three Iris species)

# Example new data for prediction:
# X_new_samples = X_test_scaled[[0, 1, 2]]
```

Expected Output

```
Starting Softmax Classifier Training...
Training Complete.

Evaluating Model on Test Data...
```

```
Test Loss: 0.0512
Test Accuracy: 96.67%

Making Predictions...
Sample Predictions:
Sample 1: True Class: 1, Predicted Probabilities: [0.    0.999 0.   ],
Predicted Class: 1
Sample 2: True Class: 0, Predicted Probabilities: [0.999 0.    0.   ],
Predicted Class: 0
Sample 3: True Class: 2, Predicted Probabilities: [0.    0.001 0.999],
Predicted Class: 2
```

(Note: Loss and accuracy will vary slightly based on random initialization and dataset split. Predicted probabilities will be close to 0 or 1 for the correct class after training.)

Lab 6: Feedforward Neural Network for Regression Tasks

Aim

To implement a basic feedforward neural network with one or more hidden layers and train it on a simple dataset for regression tasks.

Procedure

1. **Network Architecture:** Define the network architecture similar to Lab 3, but with a crucial difference in the output layer.
2. **Output Layer Activation:** For regression, the output layer typically uses a **linear activation function** (or no activation function, as it's implicitly linear). The number of output units should match the number of target variables.
3. **Loss Function:** Implement a loss function suitable for regression, such as Mean Squared Error (MSE).
    - MSE=N1∑i=1N(yi−y^i)2**Error! Filename not specified.**
4. **Backpropagation:** Implement backpropagation to compute gradients for weights and biases with respect to the MSE loss.
5. **Gradient Descent:** Update weights and biases using an optimizer (e.g., SGD, Adam).
6. **Training Loop:** Iterate for a specified number of epochs, performing forward pass, loss calculation, backward pass, and parameter updates.
7. **Evaluation:** Evaluate the model's performance on a test set using regression metrics like MSE or Root Mean Squared Error (RMSE).

Source Code

```
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_regression # For generating a synthetic
regression dataset

# Suppress TensorFlow warnings
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

class FeedForwardNeuralNetworkRegressor:
    def __init__(self, input_size, hidden_size, output_size,
learning_rate=0.01):
        """
        Initializes a feedforward neural network for regression.

        Args:
            input_size (int): Number of input features.
            hidden_size (int): Number of neurons in the hidden layer.
            output_size (int): Number of neurons in the output layer (e.g., 1
for single target).
            learning_rate (float): The learning rate for gradient descent.
        """
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate

        # Initialize weights and biases
        self.weights_hidden = np.random.randn(self.input_size,
self.hidden_size) * 0.01
        self.bias_hidden = np.zeros((1, self.hidden_size))
```

```
        self.weights_output = np.random.randn(self.hidden_size,
self.output_size) * 0.01
        self.bias_output = np.zeros((1, self.output_size))

    def _relu(self, x):
        """ReLU activation function."""
        return np.maximum(0, x)

    def _relu_derivative(self, x):
        """Derivative of the ReLU function."""
        return (x > 0).astype(float)

    def forward(self, inputs):
        """
        Performs the forward pass.
        Stores intermediate values needed for backpropagation.
        """
        # Hidden Layer
        self.Z1 = np.dot(inputs, self.weights_hidden) + self.bias_hidden
        self.A1 = self._relu(self.Z1)

        # Output Layer (Linear activation for regression)
        self.Z2 = np.dot(self.A1, self.weights_output) + self.bias_output
        self.A2 = self.Z2 # No activation function, or linear activation

        return self.A2

    def _mean_squared_error_loss(self, y_true, y_pred):
        """
        Calculates Mean Squared Error (MSE) loss.
        """
        return np.mean((y_true.reshape(-1, 1) - y_pred)**2)

    def backward(self, inputs, y_true, y_pred):
        """
        Performs the backward pass (backpropagation) and updates
weights/biases.
        """
        m = inputs.shape[0]

        # Output layer error (derivative of MSE with respect to y_pred is
2*(y_pred - y_true)/m)
        # For linear output, dL/dZ2 = dL/dA2 * dA2/dZ2 = (y_pred - y_true) *
1
        dZ2 = (y_pred - y_true.reshape(-1, 1)) / m

        # Gradients for output layer
        dW2 = np.dot(self.A1.T, dZ2)
        db2 = np.sum(dZ2, axis=0, keepdims=True)

        # Hidden layer error
        dA1 = np.dot(dZ2, self.weights_output.T)
        dZ1 = dA1 * self._relu_derivative(self.Z1)

        # Gradients for hidden layer
        dW1 = np.dot(inputs.T, dZ1)
        db1 = np.sum(dZ1, axis=0, keepdims=True)

        # Update weights and biases
        self.weights_output -= self.learning_rate * dW2
        self.bias_output -= self.learning_rate * db2
        self.weights_hidden -= self.learning_rate * dW1
        self.bias_hidden -= self.learning_rate * db1

    def train(self, X_train, y_train, epochs=5000):
        """
```

```
        Trains the neural network for regression.

        Args:
            X_train (numpy.ndarray): Training features.
            y_train (numpy.ndarray): True target values for training.
            epochs (int): Number of training epochs.
        """
        print("Starting Neural Network Regression Training...")
        for epoch in range(epochs):
            # Forward pass
            y_pred = self.forward(X_train)

            # Calculate loss
            loss = self._mean_squared_error_loss(y_train, y_pred)

            # Backward pass and update
            self.backward(X_train, y_train, y_pred)

            if (epoch + 1) % 500 == 0:
                print(f"Epoch {epoch + 1}/{epochs}, MSE Loss: {loss:.4f}")
        print("Training Complete.")

    def evaluate(self, X_test, y_test):
        """
        Evaluates the network's performance on test data.

        Args:
            X_test (numpy.ndarray): Test features.
            y_test (numpy.ndarray): True target values for testing.

        Returns:
            float: MSE on the test set.
        """
        predictions = self.forward(X_test)
        mse = self._mean_squared_error_loss(y_test, predictions)
        return mse

# Example Usage: Synthetic Regression Dataset
if __name__ == "__main__":
    # Generate a synthetic regression dataset
    X, y = make_regression(n_samples=100, n_features=2, noise=10,
random_state=42)

    # Data Preprocessing
    # 1. Split data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

    # 2. Scale features and target (important for regression)
    scaler_X = StandardScaler()
    X_train_scaled = scaler_X.fit_transform(X_train)
    X_test_scaled = scaler_X.transform(X_test)

    scaler_y = StandardScaler()
    y_train_scaled = scaler_y.fit_transform(y_train.reshape(-1, 1)).flatten()
# Flatten back to 1D
    y_test_scaled = scaler_y.transform(y_test.reshape(-1, 1)).flatten()

    # Network parameters
    input_size = X_train_scaled.shape[1]
    hidden_size = 10
    output_size = 1 # For single regression target
    learning_rate = 0.01
    epochs = 5000

    # Create and train the network
```

```
    nn_regressor = FeedForwardNeuralNetworkRegressor(
        input_size, hidden_size, output_size, learning_rate
    )
    nn_regressor.train(X_train_scaled, y_train_scaled, epochs)

    # Evaluate on test data
    test_mse_scaled = nn_regressor.evaluate(X_test_scaled, y_test_scaled)
    print(f"\nTest MSE (scaled data): {test_mse_scaled:.4f}")

    # Make predictions and inverse transform to original scale
    predictions_scaled = nn_regressor.forward(X_test_scaled)
    predictions_original = scaler_y.inverse_transform(predictions_scaled)

    print("\nSample Predictions (Original Scale):")
    for i in range(5): # Print first 5 test samples
        print(f"True: {y_test[i]:.2f}, Predicted:
{predictions_original[i][0]:.2f}")
```

Input

```
# Synthetic regression dataset (generated by make_regression)
# Example features (X):
# [[ 0.49  0.92]
#  [-0.15 -0.1 ]
#  ...
#  [-0.1  -0.12]]

# Example targets (y):
# [ 48.06  -1.39 ...  -1.04]
```

Expected Output

```
Starting Neural Network Regression Training...
Epoch 500/5000, MSE Loss: 0.1234
Epoch 1000/5000, MSE Loss: 0.0567
...
Epoch 4500/5000, MSE Loss: 0.0123
Epoch 5000/5000, MSE Loss: 0.0105
Training Complete.

Test MSE (scaled data): 0.0110

Sample Predictions (Original Scale):
True: 37.94, Predicted: 38.50
True: 10.66, Predicted: 10.15
True: -7.03, Predicted: -7.20
True: -11.08, Predicted: -10.95
True: -10.11, Predicted: -10.30
```

(Note: MSE values and predictions will vary due to random initialization and dataset generation, but the MSE should decrease significantly during training, indicating learning.)

Lab 7: Simple Autoencoder Neural Network

Aim

To implement a simple autoencoder neural network for unsupervised learning tasks such as dimensionality reduction or image denoising.

Procedure

1. **Autoencoder Concept:** Understand that an autoencoder consists of two parts: an **encoder** that compresses the input into a lower-dimensional representation (latent space) and a **decoder** that reconstructs the input from this latent space. The goal is for the reconstructed output to be as close as possible to the original input.
2. **Network Architecture:**
    - **Encoder:** A feedforward network that maps the input to a hidden layer (the latent space) with fewer neurons than the input.
    - **Decoder:** Another feedforward network that maps the latent space back to an output layer with the same number of neurons as the input.
    - The output layer should typically use a linear activation for reconstruction of continuous data, or sigmoid for data normalized between 0 and 1 (like image pixels).
3. **Loss Function:** Use a reconstruction loss, such as Mean Squared Error (MSE), to measure the difference between the input and the reconstructed output.
4. **Training:** Train the autoencoder by feeding it input data and optimizing the network to minimize the reconstruction loss. Backpropagation is used to update weights and biases.
5. **Application (Dimensionality Reduction):** After training, the encoder part of the network can be used to obtain a lower-dimensional representation of new data.
6. **Application (Image Denoising - Conceptual):** For denoising, you would train the autoencoder with noisy images as input and clean images as the target output.

Source Code

```
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.datasets import make_blobs # For generating a simple dataset

# Suppress TensorFlow warnings
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

class Autoencoder:
    def __init__(self, input_dim, encoding_dim):
        """
        Initializes a simple autoencoder.

        Args:
            input_dim (int): Dimension of the input data.
            encoding_dim (int): Dimension of the latent space (should be less
than input_dim).
        """
        self.input_dim = input_dim
        self.encoding_dim = encoding_dim
        self.model = self._build_model()

    def _build_model(self):
        """
        Builds the Keras Sequential model for the autoencoder.
        """
```

```
        # Encoder
        encoder_input = tf.keras.layers.Input(shape=(self.input_dim,))
        encoded = tf.keras.layers.Dense(self.encoding_dim,
activation='relu')(encoder_input)

        # Decoder
        decoded = tf.keras.layers.Dense(self.input_dim,
activation='sigmoid')(encoded) # Sigmoid for output between 0 and 1

        # Autoencoder model
        autoencoder = tf.keras.Model(inputs=encoder_input, outputs=decoded)

        # Compile the autoencoder
        autoencoder.compile(optimizer='adam', loss='mse') # Mean Squared
Error for reconstruction loss
        return autoencoder

    def train(self, X_train, epochs=50, batch_size=32, validation_split=0.2):
        """
        Trains the autoencoder.

        Args:
            X_train (numpy.ndarray): Training data.
            epochs (int): Number of training epochs.
            batch_size (int): Batch size for training.
            validation_split (float): Fraction of training data to be used as
validation data.
        """
        print("Starting Autoencoder Training...")
        history = self.model.fit(X_train, X_train, # Input and target are the
same for autoencoders
                                 epochs=epochs,
                                 batch_size=batch_size,
                                 validation_split=validation_split,
                                 verbose=0)
        print("Training Complete.")
        return history

    def get_encoder(self):
        """
        Returns the encoder part of the autoencoder.
        """
        # The encoder is the first layer of the autoencoder model
        encoder_input = self.model.input
        encoded_output = self.model.layers[0].output # Output of the first
Dense layer (encoded representation)
        encoder = tf.keras.Model(inputs=encoder_input,
outputs=encoded_output)
        return encoder

    def get_decoder(self):
        """
        Returns the decoder part of the autoencoder.
        """
        # Create a new input layer for the decoder, matching the encoding_dim
        encoded_input = tf.keras.layers.Input(shape=(self.encoding_dim,))
        # The decoder part starts from the second Dense layer in the original
model
        decoded_output = self.model.layers[1](encoded_input) # This assumes
the decoder is the second Dense layer
        decoder = tf.keras.Model(inputs=encoded_input,
outputs=decoded_output)
        return decoder

# Example Usage: Dimensionality Reduction on a synthetic dataset
if __name__ == "__main__":
```

```
    # Generate a synthetic dataset with 10 features
    X, _ = make_blobs(n_samples=1000, n_features=10, centers=3,
random_state=42)

    # Data Preprocessing: Scale data to [0, 1] for sigmoid output
    scaler = MinMaxScaler()
    X_scaled = scaler.fit_transform(X)

    # Split data
    X_train, X_test = train_test_split(X_scaled, test_size=0.2,
random_state=42)

    # Define autoencoder parameters
    input_dim = X_train.shape[1]
    encoding_dim = 2 # Reduce to 2 dimensions for visualization potential

    # Create and train the autoencoder
    autoencoder = Autoencoder(input_dim, encoding_dim)
    autoencoder.train(X_train, epochs=100)

    # Get the encoder model
    encoder = autoencoder.get_encoder()

    # Get the decoder model
    decoder = autoencoder.get_decoder()

    # Use the encoder to reduce dimensionality of test data
    encoded_test_data = encoder.predict(X_test)
    print(f"\nOriginal test data shape: {X_test.shape}")
    print(f"Encoded test data shape (dimensionality reduced):
{encoded_test_data.shape}")

    # Use the decoder to reconstruct data from the encoded representation
    reconstructed_test_data = decoder.predict(encoded_test_data)
    print(f"Reconstructed test data shape: {reconstructed_test_data.shape}")

    # Calculate reconstruction error on test set
    reconstruction_error = np.mean((X_test - reconstructed_test_data)**2)
    print(f"Reconstruction MSE on test set: {reconstruction_error:.4f}")

    print("\nOriginal vs Reconstructed (first 5 samples):")
    for i in range(5):
        print(f"Original: {np.round(X_test[i], 2)}")
        print(f"Reconstructed: {np.round(reconstructed_test_data[i], 2)}")
        print("-" * 30)
```

Input

```
# Synthetic dataset (generated by make_blobs)
# Example features (X_scaled, normalized to [0,1]):
# [[0.45 0.5  0.4  0.55 0.6  0.48 0.52 0.49 0.51 0.47]
#  [0.9  0.95 0.88 0.92 0.91 0.93 0.94 0.89 0.9  0.96]
#  ...
#  [0.1  0.15 0.08 0.12 0.11 0.13 0.14 0.09 0.1  0.16]]
```

Expected Output

```
Starting Autoencoder Training...
Training Complete.

Original test data shape: (200, 10)
Encoded test data shape (dimensionality reduced): (200, 2)
Reconstructed test data shape: (200, 10)
Reconstruction MSE on test set: 0.0015
```

```
Original vs Reconstructed (first 5 samples):
Original: [0.38 0.43 0.35 0.4  0.41 0.39 0.42 0.37 0.4  0.4 ]
Reconstructed: [0.38 0.43 0.35 0.4  0.41 0.39 0.42 0.37 0.4  0.4 ]
------------------------------
Original: [0.93 0.98 0.91 0.96 0.95 0.97 0.98 0.92 0.95 0.99]
Reconstructed: [0.93 0.98 0.91 0.96 0.95 0.97 0.98 0.92 0.95 0.99]
------------------------------
Original: [0.08 0.13 0.05 0.1  0.09 0.11 0.12 0.07 0.1  0.13]
Reconstructed: [0.08 0.13 0.05 0.1  0.09 0.11 0.12 0.07 0.1  0.13]
------------------------------
Original: [0.4  0.45 0.37 0.42 0.43 0.41 0.44 0.39 0.42 0.42]
Reconstructed: [0.4  0.45 0.37 0.42 0.43 0.41 0.44 0.39 0.42 0.42]
------------------------------
Original: [0.89 0.94 0.86 0.91 0.9  0.92 0.93 0.88 0.91 0.95]
Reconstructed: [0.89 0.94 0.86 0.91 0.9  0.92 0.93 0.88 0.91 0.95]
------------------------------
```

(Note: The reconstruction error should be very low, indicating that the autoencoder can effectively reconstruct the input. The `np.round` is used for display purposes; the actual values will be floats.)

Lab 8: Autoencoder Training on MNIST

Aim

To implement and train a simple autoencoder neural network for unsupervised learning tasks such as dimensionality reduction or image denoising, specifically training the autoencoder on datasets like the MNIST handwritten digit dataset.

Procedure

1. **Dataset Loading and Preprocessing:**
    - Load the MNIST handwritten digit dataset.
    - Normalize pixel values to a range suitable for the autoencoder's output activation (e.g., [0, 1] for sigmoid output).
    - Flatten the 2D image data into 1D vectors, as a simple feedforward autoencoder expects 1D input.
2. **Autoencoder Architecture:**
    - Define an encoder-decoder architecture. For MNIST (28x28 images = 784 pixels), the input and output layers will have 784 units.
    - The hidden (latent) layer should have significantly fewer units (e.g., 32, 64, or 128) for dimensionality reduction.
    - Use appropriate activation functions (e.g., ReLU for hidden layers, Sigmoid for the output layer to reconstruct pixel values between 0 and 1).
3. **Loss Function and Optimizer:** Use Mean Squared Error (MSE) as the reconstruction loss and an optimizer like Adam.
4. **Training:** Train the autoencoder using the `fit` method, providing the same data for both input and target (since it's an unsupervised reconstruction task).
5. **Evaluation and Visualization:**
    - After training, use the autoencoder to reconstruct some test images.
    - Visually compare the original images with their reconstructed counterparts to assess the autoencoder's performance.
    - Optionally, use the encoder part to visualize the latent space (e.g., using t-SNE if the latent dimension is 2 or 3).

Source Code

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# Suppress TensorFlow warnings
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

class MNISTAutoencoder:
    def __init__(self, original_dim, encoding_dim):
        """
        Initializes an autoencoder for MNIST.

        Args:
            original_dim (int): Dimension of the flattened input image (e.g.,
28*28 = 784).
            encoding_dim (int): Dimension of the latent space.
        """
        self.original_dim = original_dim
        self.encoding_dim = encoding_dim
        self.model = self._build_model()

    def _build_model(self):
```

```
        """
        Builds the Keras Sequential model for the autoencoder.
        """
        # Encoder
        encoder_input = tf.keras.layers.Input(shape=(self.original_dim,))
        encoded = tf.keras.layers.Dense(self.encoding_dim,
activation='relu')(encoder_input)

        # Decoder
        decoded = tf.keras.layers.Dense(self.original_dim,
activation='sigmoid')(encoded) # Sigmoid for pixel values [0, 1]

        # Autoencoder model
        autoencoder = tf.keras.Model(inputs=encoder_input, outputs=decoded)

        # Compile the autoencoder
        autoencoder.compile(optimizer='adam', loss='mse') # Mean Squared
Error for reconstruction loss
        return autoencoder

    def train(self, X_train, epochs=50, batch_size=256,
validation_split=0.1):
        """
        Trains the autoencoder on MNIST data.

        Args:
            X_train (numpy.ndarray): Training data (flattened and normalized
MNIST images).
            epochs (int): Number of training epochs.
            batch_size (int): Batch size for training.
            validation_split (float): Fraction of training data to be used as
validation data.
        """
        print("Starting MNIST Autoencoder Training...")
        history = self.model.fit(X_train, X_train, # Input and target are the
same
                                 epochs=epochs,
                                 batch_size=batch_size,
                                 shuffle=True, # Shuffle training data
                                 validation_split=validation_split,
                                 verbose=0)
        print("Training Complete.")
        return history

    def reconstruct_images(self, X_test):
        """
        Reconstructs images from the test set using the trained autoencoder.

        Args:
            X_test (numpy.ndarray): Test data (flattened and normalized MNIST
images).

        Returns:
            numpy.ndarray: Reconstructed images.
        """
        return self.model.predict(X_test)

    def get_encoder(self):
        """
        Returns the encoder part of the autoencoder.
        """
        encoder_input = self.model.input
        encoded_output = self.model.layers[0].output
        encoder = tf.keras.Model(inputs=encoder_input,
outputs=encoded_output)
        return encoder
```

```
# Example Usage: MNIST Dataset
if __name__ == "__main__":
    # Load MNIST dataset
    (X_train, _), (X_test, _) = tf.keras.datasets.mnist.load_data()

    # Data Preprocessing
    # Normalize pixel values to [0, 1]
    X_train = X_train.astype('float32') / 255.
    X_test = X_test.astype('float32') / 255.

    # Flatten images (28x28 to 784)
    X_train_flat = X_train.reshape((len(X_train),
np.prod(X_train.shape[1:])))
    X_test_flat = X_test.reshape((len(X_test), np.prod(X_test.shape[1:])))

    # Define autoencoder parameters
    original_dim = X_train_flat.shape[1] # 784
    encoding_dim = 32 # Latent space dimension

    # Create and train the autoencoder
    mnist_ae = MNISTAutoencoder(original_dim, encoding_dim)
    mnist_ae.train(X_train_flat, epochs=50)

    # Reconstruct some test images
    num_images_to_show = 10
    reconstructed_images =
mnist_ae.reconstruct_images(X_test_flat[:num_images_to_show])

    print(f"\nDisplaying {num_images_to_show} original vs. reconstructed
images:")
    plt.figure(figsize=(20, 4))
    for i in range(num_images_to_show):
        # Original Image
        ax = plt.subplot(2, num_images_to_show, i + 1)
        plt.imshow(X_test_flat[i].reshape(28, 28))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        if i == 0:
            ax.set_title("Original")

        # Reconstructed Image
        ax = plt.subplot(2, num_images_to_show, i + 1 + num_images_to_show)
        plt.imshow(reconstructed_images[i].reshape(28, 28))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        if i == 0:
            ax.set_title("Reconstructed")
    plt.show()

    # Example of using the encoder for dimensionality reduction
    encoder_model = mnist_ae.get_encoder()
    encoded_sample = encoder_model.predict(X_test_flat[0:1])
    print(f"\nShape of original image (flattened): {X_test_flat[0:1].shape}")
    print(f"Shape of encoded representation: {encoded_sample.shape}")
    print(f"Encoded representation for first test
image:\n{np.round(encoded_sample, 3)}")
```

Input

```
# MNIST dataset (loaded internally by tf.keras.datasets.mnist.load_data())
# X_train, X_test are arrays of 28x28 grayscale images.
```

```
# After preprocessing, they become flattened 1D arrays of 784 pixel values,
normalized to [0, 1].
```

Expected Output

```
Starting MNIST Autoencoder Training...
Training Complete.

Displaying 10 original vs. reconstructed images:
(A matplotlib plot showing 10 original MNIST digits in the top row and their
corresponding reconstructed digits in the bottom row. The reconstructed
digits should closely resemble the originals, though with some loss of
detail.)

Shape of original image (flattened): (1, 784)
Shape of encoded representation: (1, 32)
Encoded representation for first test image:
[[0.    0.    0.    ... 0.    0.    0.   ]]
```

(Note: The exact pixel values in the encoded representation will vary. The plot will be displayed in a separate window or inline if in a Jupyter environment.)

Lab 9: Feedforward Neural Network with Backpropagation for Binary Classification

Aim

To implement a feedforward neural network trained using backpropagation for a binary classification task, reinforcing the concepts from Lab 4 with a focus on a more robust implementation.

Procedure

This lab is a more detailed and self-contained version of Lab 4, ensuring all components of a backpropagation-trained FFNN for binary classification are clearly defined.

1. **Network Class:** Create a class for the neural network that encapsulates initialization, forward pass, backward pass, and training.
2. **Initialization:** Initialize weights and biases for all layers with small random values.
3. **Activation Functions:** Implement Sigmoid for the output layer and ReLU (or another non-linear activation) for hidden layers, along with their derivatives.
4. **Forward Pass:** Calculate outputs for each layer sequentially. Store intermediate activations and weighted sums for use in backpropagation.
5. **Loss Function:** Implement Binary Cross-Entropy loss.
6. **Backward Pass (Backpropagation):**
    - Calculate the error at the output layer.
    - Propagate the error backward, calculating gradients for weights and biases in each layer.
    - Use the chain rule for derivatives of activation functions.
7. **Weight Update:** Update weights and biases using gradient descent with a specified learning rate.
8. **Training Loop:** Iterate through epochs, performing forward pass, loss calculation, backward pass, and weight updates. Monitor loss and accuracy.
9. **Prediction and Evaluation:** Implement methods to make predictions and evaluate the model's accuracy on unseen data.

Source Code

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_classification # For generating a synthetic
binary classification dataset

class NeuralNetworkBinaryClassifier:
    def __init__(self, input_size, hidden_layer_sizes, output_size,
learning_rate=0.01):
        """
        Initializes a multi-layer feedforward neural network for binary
classification.

        Args:
            input_size (int): Number of input features.
            hidden_layer_sizes (list): A list where each element is the
number of neurons
                                       in a hidden layer. E.g., [64, 32] for
two hidden layers.
            output_size (int): Number of neurons in the output layer (should
be 1).
```

```
            learning_rate (float): The learning rate for gradient descent.
        """
        self.input_size = input_size
        self.hidden_layer_sizes = hidden_layer_sizes
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.num_layers = len(hidden_layer_sizes) + 1 # +1 for output layer

        self.weights = []
        self.biases = []
        self.activations = [] # Stores activations (A) for each layer during
forward pass
        self.zs = []          # Stores weighted sums (Z) for each layer
during forward pass

        # Initialize weights and biases for all layers
        layer_sizes = [input_size] + hidden_layer_sizes + [output_size]
        for i in range(len(layer_sizes) - 1):
            # Weights: (previous_layer_size, current_layer_size)
            self.weights.append(np.random.randn(layer_sizes[i],
layer_sizes[i+1]) * 0.01)
            # Biases: (1, current_layer_size)
            self.biases.append(np.zeros((1, layer_sizes[i+1])))

    def _sigmoid(self, x):
        """Sigmoid activation function."""
        return 1 / (1 + np.exp(-x))

    def _sigmoid_derivative(self, x):
        """Derivative of the sigmoid function."""
        s = self._sigmoid(x)
        return s * (1 - s)

    def _relu(self, x):
        """ReLU activation function."""
        return np.maximum(0, x)

    def _relu_derivative(self, x):
        """Derivative of the ReLU function."""
        return (x > 0).astype(float)

    def forward(self, inputs):
        """
        Performs the forward pass through the network.
        Stores intermediate values for backpropagation.
        """
        self.activations = [inputs] # A0 = inputs
        self.zs = []

        # Hidden Layers
        for i in range(self.num_layers - 1): # Iterate through hidden layers
            z = np.dot(self.activations[-1], self.weights[i]) +
self.biases[i]
            self.zs.append(z)
            a = self._relu(z) # Using ReLU for hidden layers
            self.activations.append(a)

        # Output Layer
        z_output = np.dot(self.activations[-1], self.weights[-1]) +
self.biases[-1]
        self.zs.append(z_output)
        a_output = self._sigmoid(z_output) # Sigmoid for binary
classification output
        self.activations.append(a_output)

        return a_output
```

```
    def _binary_cross_entropy_loss(self, y_true, y_pred):
        """
        Calculates Binary Cross-Entropy loss.
        Adds a small epsilon to avoid log(0).
        """
        epsilon = 1e-10
        return -np.mean(y_true * np.log(y_pred + epsilon) + (1 - y_true) *
np.log(1 - y_pred + epsilon))

    def backward(self, y_true, y_pred):
        """
        Performs the backward pass (backpropagation) and calculates
gradients.
        """
        m = y_true.shape[0] # Number of samples
        y_true = y_true.reshape(-1, 1) # Ensure y_true is column vector

        # Output layer error (dL/dZ_output)
        dZ = y_pred - y_true

        # Gradients for output layer
        dW_output = np.dot(self.activations[-2].T, dZ) / m
        db_output = np.sum(dZ, axis=0, keepdims=True) / m

        # Store gradients
        dWs = [dW_output]
        dbs = [db_output]

        # Backpropagate through hidden layers
        for i in reversed(range(self.num_layers - 1)): # Iterate from last
hidden to first hidden
            # dL/dA_current = dL/dZ_next * dZ_next/dA_current
            dA_current = np.dot(dZ, self.weights[i+1].T)
            # dZ_current = dL/dA_current * dA_current/dZ_current
            dZ = dA_current * self._relu_derivative(self.zs[i])

            # Gradients for current hidden layer
            dW_current = np.dot(self.activations[i].T, dZ) / m
            db_current = np.sum(dZ, axis=0, keepdims=True) / m

            dWs.insert(0, dW_current) # Insert at beginning to match layer
order
            dbs.insert(0, db_current)

        # Update weights and biases
        for i in range(len(self.weights)):
            self.weights[i] -= self.learning_rate * dWs[i]
            self.biases[i] -= self.learning_rate * dbs[i]

    def train(self, X_train, y_train, epochs=1000):
        """
        Trains the neural network.

        Args:
            X_train (numpy.ndarray): Training features.
            y_train (numpy.ndarray): True labels for training.
            epochs (int): Number of training epochs.
        """
        print("Starting Neural Network Training with Backpropagation...")
        for epoch in range(epochs):
            # Forward pass
            y_pred = self.forward(X_train)

            # Calculate loss
            loss = self._binary_cross_entropy_loss(y_train, y_pred)
```

```
            # Backward pass and update
            self.backward(y_train, y_pred)

            if (epoch + 1) % 100 == 0:
                # Calculate accuracy for monitoring
                binary_predictions = (y_pred >= 0.5).astype(int).flatten()
                accuracy = np.mean(binary_predictions == y_train)
                print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss:.4f},
Accuracy: {accuracy:.4f}")
        print("Training Complete.")

    def predict(self, X_test):
        """
        Makes predictions on new data.

        Args:
            X_test (numpy.ndarray): Input features for prediction.

        Returns:
            numpy.ndarray: Binary predictions (0 or 1).
        """
        probabilities = self.forward(X_test)
        return (probabilities >= 0.5).astype(int).flatten()

    def evaluate(self, X_test, y_test):
        """
        Evaluates the network's performance.

        Args:
            X_test (numpy.ndarray): Test features.
            y_test (numpy.ndarray): True labels for testing.

        Returns:
            float: Accuracy of the model.
        """
        predictions = self.predict(X_test)
        accuracy = np.mean(predictions == y_test)
        return accuracy

# Example Usage: Synthetic Binary Classification Dataset
if __name__ == "__main__":
    # Generate a synthetic binary classification dataset
    X, y = make_classification(n_samples=200, n_features=4, n_informative=2,
                               n_redundant=0, n_clusters_per_class=1,
random_state=42)

    # Data Preprocessing
    # 1. Split data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)

    # 2. Scale features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Network parameters
    input_size = X_train_scaled.shape[1]
    hidden_layer_sizes = [16, 8] # Two hidden layers with 16 and 8 neurons
    output_size = 1
    learning_rate = 0.05
    epochs = 2000

    # Create and train the neural network
    nn_classifier = NeuralNetworkBinaryClassifier(
```

```
        input_size, hidden_layer_sizes, output_size, learning_rate
    )
    nn_classifier.train(X_train_scaled, y_train, epochs)

    # Evaluate on test data
    test_accuracy = nn_classifier.evaluate(X_test_scaled, y_test)
    print(f"\nTest Accuracy: {test_accuracy * 100:.2f}%")

    print("\nSample Predictions on Test Data:")
    sample_indices = [0, 1, 2, 3, 4]
    X_sample = X_test_scaled[sample_indices]
    y_true_sample = y_test[sample_indices]
    y_pred_sample = nn_classifier.predict(X_sample)

    for i in range(len(sample_indices)):
        print(f"Input: {np.round(X_sample[i], 2)}, True: {y_true_sample[i]},
Predicted: {y_pred_sample[i]}")
```

Input

```
# Synthetic binary classification dataset (generated by make_classification)
# Example features (X_scaled):
# [[-0.27 -0.73  0.08  0.64]
#  [-0.64 -0.58 -0.06 -0.19]
#  ...
#  [ 0.77  0.47  0.03 -0.66]]

# Example labels (y):
# [0 1 ... 1 0]
```

Expected Output

```
Starting Neural Network Training with Backpropagation...
Epoch 100/2000, Loss: 0.2567, Accuracy: 0.9000
Epoch 200/2000, Loss: 0.1502, Accuracy: 0.9500
...
Epoch 1900/2000, Loss: 0.0051, Accuracy: 1.0000
Epoch 2000/2000, Loss: 0.0048, Accuracy: 1.0000
Training Complete.

Test Accuracy: 97.50%

Sample Predictions on Test Data:
Input: [-0.27 -0.73  0.08  0.64], True: 0, Predicted: 0
Input: [-0.64 -0.58 -0.06 -0.19], True: 1, Predicted: 1
Input: [ 0.22 -0.12 -0.21  0.51], True: 0, Predicted: 0
Input: [ 0.97  0.94  0.09 -0.66], True: 1, Predicted: 1
Input: [-0.17  0.22 -0.16  0.4 ], True: 0, Predicted: 0
```

(Note: Loss and accuracy values will vary based on random initialization and dataset generation, but accuracy should converge to high values.)

Lab 10: Gradient Descent for Simple Linear Regression

Aim

To implement the gradient descent algorithm for a simple linear regression problem in Python from scratch.

Procedure

1. **Linear Regression Model:** Understand the simple linear regression equation: y=mx+b, where m is the slope and b is the y-intercept.
2. **Loss Function:** Define the Mean Squared Error (MSE) as the loss function, which measures the difference between the predicted values (y^) and the true values (y).
    - MSE=N1∑i=1N(yi−(mxi+b))2**Error! Filename not specified.**
3. **Gradients:** Calculate the partial derivatives of the MSE loss function with respect to m and b. These derivatives represent the gradients.
    - ∂m∂MSE=−N2∑i=1Nxi(yi−y^i)**Error! Filename not specified.**
    - ∂b∂MSE=−N2∑i=1N(yi−y^i)**Error! Filename not specified.**
4. **Gradient Descent Update Rule:** Update m and b iteratively using the gradients and a learning rate (α).
    - m=m−α·∂m∂MSE**Error! Filename not specified.**
    - b=b−α·∂b∂MSE**Error! Filename not specified.**
5. **Training Loop:**
    - Initialize m and b randomly or to zero.
    - Iterate for a specified number of epochs.
    - In each epoch, calculate predictions, compute gradients, and update m and b.
    - Monitor the MSE loss to observe convergence.
6. **Prediction:** After training, use the learned m and b to make predictions on new data.

Source Code

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

class SimpleLinearRegressor:
    def __init__(self, learning_rate=0.01, epochs=1000):
        """
        Initializes the Simple Linear Regressor with Gradient Descent.

        Args:
            learning_rate (float): The learning rate for gradient descent.
            epochs (int): The number of training iterations.
        """
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.m = 0.0 # Initialize slope
        self.b = 0.0 # Initialize y-intercept
        self.loss_history = [] # To store loss at each epoch

    def predict(self, X):
        """
        Makes predictions using the current m and b.

        Args:
            X (numpy.ndarray): Input features.

        Returns:
            numpy.ndarray: Predicted target values.
        """
```

```
        return self.m * X + self.b

    def _mean_squared_error(self, y_true, y_pred):
        """
        Calculates Mean Squared Error (MSE).
        """
        return np.mean((y_true - y_pred)**2)

    def train(self, X, y):
        """
        Trains the linear regression model using gradient descent.

        Args:
            X (numpy.ndarray): Training features.
            y (numpy.ndarray): True target values.
        """
        n = len(X) # Number of samples
        print("Starting Linear Regression Training with Gradient Descent...")
        for epoch in range(self.epochs):
            y_pred = self.predict(X)

            # Calculate gradients
            # dL/dm = -2/N * sum(x_i * (y_i - y_pred_i))
            dm = (-2 / n) * np.sum(X * (y - y_pred))
            # dL/db = -2/N * sum(y_i - y_pred_i)
            db = (-2 / n) * np.sum(y - y_pred)

            # Update m and b
            self.m -= self.learning_rate * dm
            self.b -= self.learning_rate * db

            # Calculate and store loss
            loss = self._mean_squared_error(y, y_pred)
            self.loss_history.append(loss)

            if (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch + 1}/{self.epochs}, MSE Loss:
{loss:.4f}, m: {self.m:.4f}, b: {self.b:.4f}")
        print("Training Complete.")

# Example Usage
if __name__ == "__main__":
    # Generate some synthetic data for linear regression
    np.random.seed(42)
    X = 2 * np.random.rand(100, 1) # 100 random values between 0 and 2
    y = 4 + 3 * X + np.random.randn(100, 1) * 2 # y = 4 + 3x + noise

    # Flatten X and y to 1D arrays for simpler handling
    X = X.flatten()
    y = y.flatten()

    # Create and train the regressor
    regressor = SimpleLinearRegressor(learning_rate=0.01, epochs=1000)
    regressor.train(X, y)

    print(f"\nFinal learned parameters: m = {regressor.m:.4f}, b =
{regressor.b:.4f}")

    # Make predictions on the training data
    y_pred = regressor.predict(X)

    # Plotting the results
    plt.figure(figsize=(10, 6))
    plt.scatter(X, y, label='Original Data', alpha=0.7)
    plt.plot(X, y_pred, color='red', label=f'Regression Line (y =
{regressor.m:.2f}x + {regressor.b:.2f})')
```

```
    plt.xlabel('X')
    plt.ylabel('y')
    plt.title('Linear Regression with Gradient Descent')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Plotting the loss history
    plt.figure(figsize=(10, 4))
    plt.plot(regressor.loss_history, color='blue')
    plt.xlabel('Epoch')
    plt.ylabel('MSE Loss')
    plt.title('MSE Loss during Training')
    plt.grid(True)
    plt.show()
```

Input

```
# Synthetic data generated by the script:
# X (features): 100 random values between 0 and 2
# y (targets): y = 4 + 3 * X + noise
```

Expected Output

```
Starting Linear Regression Training with Gradient Descent...
Epoch 100/1000, MSE Loss: 4.8765, m: 2.7654, b: 4.3123
Epoch 200/1000, MSE Loss: 4.1234, m: 2.8976, b: 4.1567
...
Epoch 900/1000, MSE Loss: 3.8456, m: 3.0123, b: 3.9987
Epoch 1000/1000, MSE Loss: 3.8450, m: 3.0130, b: 3.9970
Training Complete.

Final learned parameters: m = 3.0130, b = 3.9970

(A scatter plot showing the original data points and a red line representing
the learned regression line, which should fit the data well.)
(A line plot showing the MSE loss decreasing over epochs, indicating
convergence.)
```

(Note: Exact numerical values for m, b, and loss will vary slightly due to random noise in data generation and initialization, but they should converge close to the true values (m=3, b=4) and the loss should minimize.)

Lab 11: Basic Architecture of an RBF Network

Aim

To implement the basic architecture of a Radial Basis Function (RBF) network, consisting of an input layer, a hidden layer with radial basis functions, and an output layer.

Procedure

1. **RBF Network Structure:** Understand that an RBF network typically has three layers:
    - **Input Layer:** Passes the input features directly to the hidden layer.
    - **Hidden Layer (RBF Layer):** Contains RBF neurons, each characterized by a **center** (μ) and a **width** (σ). Each RBF neuron computes its activation based on the distance between the input vector and its center, scaled by its width. A common RBF is the Gaussian function:
        - ϕ(x)=exp(−2σ2|x−μ|2)**Error! Filename not specified.**
    - **Output Layer:** A linear combination of the activations from the R RBF neurons, multiplied by output weights (wj).
        - y^=∑j=1Rwjϕj(x)+b**Error! Filename not specified.**
2. **Initialization:**
    - Initialize the centers (μ) and widths (σ) for the RBF neurons. For this lab, we can randomly pick centers from the training data and use a fixed width.
    - Initialize the output layer weights (w) and bias (b) randomly.
3. **Forward Pass:** Implement the forward propagation:
    - For each input sample, calculate the activation of each RBF neuron in the hidden layer.
    - Compute the weighted sum of RBF activations and add the bias to produce the final output.

Source Code

```
import numpy as np
from sklearn.datasets import make_regression # For generating a synthetic
dataset

class RBFNetwork:
    def __init__(self, input_dim, num_rbf_neurons, output_dim):
        """
        Initializes the basic architecture of an RBF Network.

        Args:
            input_dim (int): Dimension of the input features.
            num_rbf_neurons (int): Number of RBF neurons in the hidden layer.
            output_dim (int): Dimension of the output (e.g., 1 for
regression).
        """
        self.input_dim = input_dim
        self.num_rbf_neurons = num_rbf_neurons
        self.output_dim = output_dim

        # Centers and widths for RBF neurons
        # These will typically be learned or determined by clustering in
training
        # For this architecture-only lab, we'll initialize them randomly or
conceptually.
        # Here, we'll use random initialization within a plausible range.
        self.centers = np.random.rand(self.num_rbf_neurons, self.input_dim) *
5 - 2.5 # Random centers
```

```
        self.widths = np.ones(self.num_rbf_neurons) * 1.0 # Initial fixed
width for all RBFs

        # Output layer weights and bias
        self.output_weights = np.random.randn(self.num_rbf_neurons,
self.output_dim) * 0.01
        self.output_bias = np.zeros((1, self.output_dim))

    def _gaussian_rbf(self, x, center, width):
        """
        Gaussian Radial Basis Function.

        Args:
            x (numpy.ndarray): Input vector.
            center (numpy.ndarray): Center of the RBF.
            width (float): Width (sigma) of the RBF.

        Returns:
            float: Activation of the RBF neuron.
        """
        # Calculate Euclidean distance squared
        distance_sq = np.sum((x - center)**2)
        # Apply Gaussian formula
        return np.exp(-distance_sq / (2 * width**2))

    def forward(self, inputs):
        """
        Performs the forward pass through the RBF network.

        Args:
            inputs (numpy.ndarray): Input data (batch_size, input_dim).

        Returns:
            numpy.ndarray: Output of the network (batch_size, output_dim).
        """
        batch_size = inputs.shape[0]
        rbf_activations = np.zeros((batch_size, self.num_rbf_neurons))

        # Calculate activations for each RBF neuron for each input sample
        for i in range(batch_size):
            for j in range(self.num_rbf_neurons):
                rbf_activations[i, j] = self._gaussian_rbf(inputs[i],
self.centers[j], self.widths[j])

        # Output layer calculation: linear combination of RBF activations
        output = np.dot(rbf_activations, self.output_weights) +
self.output_bias
        return output

# Example Usage
if __name__ == "__main__":
    # Generate some synthetic data (e.g., for regression)
    X, y = make_regression(n_samples=50, n_features=2, noise=5,
random_state=42)

    # Define RBF network parameters
    input_dim = X.shape[1]      # Number of features
    num_rbf_neurons = 10        # Number of RBF neurons in the hidden layer
    output_dim = 1              # Single output for regression

    # Create an instance of the RBF network
    rbf_net = RBFNetwork(input_dim, num_rbf_neurons, output_dim)

    print(f"RBF Network initialized with {input_dim} input dimensions, "
          f"{num_rbf_neurons} RBF neurons, and {output_dim} output
dimension.")
```

```
    # Display initial random centers and widths (for demonstration)
    print("\nInitial RBF Centers (first 3):\n", np.round(rbf_net.centers[:3],
2))
    print("Initial RBF Widths (first 3):", np.round(rbf_net.widths[:3], 2))
    print(f"Shape of output weights: {rbf_net.output_weights.shape}")
    print(f"Shape of output bias: {rbf_net.output_bias.shape}")

    # Perform a forward pass with some dummy input
    dummy_inputs = np.array([
        [0.5, 1.2],
        [-0.8, 0.3],
        [2.1, -1.5]
    ])
    print("\nDummy Inputs:\n", np.round(dummy_inputs, 2))

    predictions = rbf_net.forward(dummy_inputs)
    print("\nNetwork Output (Predictions):\n", np.round(predictions, 2))
```

Input

```
# Synthetic data generated by make_regression (e.g., for X):
# [[ 0.94 -0.19]
#  [-0.14  0.08]
#  ...
#  [ 0.7  -0.09]]

# Dummy inputs for forward pass:
dummy_inputs = np.array([
    [0.5, 1.2],
    [-0.8, 0.3],
    [2.1, -1.5]
])
```

Expected Output

```
RBF Network initialized with 2 input dimensions, 10 RBF neurons, and 1 output
dimension.

Initial RBF Centers (first 3):
 [[-0.54 -1.33]
 [ 1.95 -1.21]
 [-0.34  0.64]]
Initial RBF Widths (first 3): [1. 1. 1.]
Shape of output weights: (10, 1)
Shape of output bias: (1, 1)

Dummy Inputs:
 [[ 0.5  1.2]
 [-0.8  0.3]
 [ 2.1 -1.5]]

Network Output (Predictions):
 [[ 0.01]
 [-0.01]
 [ 0.01]]
```

(Note: The exact numerical values for centers, weights, and predictions will vary due to random initialization, but the shapes and general structure will be consistent.)

Lab 12: Training Algorithm for RBF Network

Aim

To implement the training algorithm for the RBF network, such as the k-means clustering algorithm for determining the centers of the radial basis functions, and the least squares method for computing the output layer weights.

Procedure

1. **RBF Network Recap:** Recall the RBF network architecture from Lab 11. The key challenge is to determine the RBF centers, widths, and output weights.
2. **K-Means for Centers:**
    - Apply the k-means clustering algorithm to the training input data.
    - The centroids found by k-means will serve as the **centers (μ)** for the RBF neurons. The number of clusters will be equal to the number of RBF neurons.
3. **Widths (σ) Determination:**
    - A common heuristic for widths is to set them based on the maximum distance between the chosen centers, or the average distance to the nearest neighbor center. A simpler approach for this lab is to use a fixed width, or calculate it as the average distance from each center to the data points assigned to it.
    - Another approach is to set σj=2·NRBFdmax, where dmax is the maximum distance between any two cluster centers, and NRBF is the number of RBF neurons.
4. **Feature Transformation (RBF Layer Output):** For each input sample, calculate the activation of each RBF neuron using the determined centers and widths. This transforms the input data into a new feature space (the RBF activations).
5. **Least Squares for Output Weights:**
    - Once the RBF activations are computed for all training inputs, the problem becomes a linear regression problem in this new feature space.
    - Use the **least squares method** (e.g., using `numpy.linalg.lstsq` or by solving the normal equations: W=(HTH)−1HTY) to find the optimal output weights (W) and bias (b). Here, H is the matrix of RBF activations (plus a column of ones for the bias term).

Source Code

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_regression
import matplotlib.pyplot as plt

class RBFNetworkTrainer:
    def __init__(self, num_rbf_neurons):
        """
        Initializes the RBF Network Trainer.

        Args:
            num_rbf_neurons (int): Number of RBF neurons in the hidden layer.
        """
        self.num_rbf_neurons = num_rbf_neurons
        self.centers = None
        self.widths = None
        self.output_weights = None
        self.output_bias = None
```

```
    def _gaussian_rbf(self, x, center, width):
        """Gaussian Radial Basis Function."""
        distance_sq = np.sum((x - center)**2)
        # Add a small epsilon to width_sq to prevent division by zero if
width is very small
        return np.exp(-distance_sq / (2 * (width**2 + 1e-8)))

    def _calculate_rbf_activations(self, X):
        """
        Calculates the activations of all RBF neurons for a given input
dataset.
        """
        num_samples = X.shape[0]
        rbf_activations = np.zeros((num_samples, self.num_rbf_neurons))

        for i in range(num_samples):
            for j in range(self.num_rbf_neurons):
                rbf_activations[i, j] = self._gaussian_rbf(X[i],
self.centers[j], self.widths[j])
        return rbf_activations

    def train(self, X_train, y_train):
        """
        Trains the RBF network using K-Means for centers and Least Squares
for output weights.

        Args:
            X_train (numpy.ndarray): Training features.
            y_train (numpy.ndarray): True target values for training.
        """
        print("Starting RBF Network Training...")

        # Step 1: Determine RBF Centers using K-Means
        print(f"Running K-Means with {self.num_rbf_neurons} clusters...")
        kmeans = KMeans(n_clusters=self.num_rbf_neurons, random_state=42,
n_init=10)
        kmeans.fit(X_train)
        self.centers = kmeans.cluster_centers_
        print("K-Means complete. Centers determined.")

        # Step 2: Determine RBF Widths
        # Heuristic: Average distance from each center to the data points in
its cluster
        # Or, a simpler global width based on max distance between centers
        # For simplicity, let's use a global width based on max distance
between centers
        max_dist = 0
        for i in range(self.num_rbf_neurons):
            for j in range(i + 1, self.num_rbf_neurons):
                dist = np.linalg.norm(self.centers[i] - self.centers[j])
                if dist > max_dist:
                    max_dist = dist
        # A common heuristic: width = max_dist / sqrt(2 * num_rbf_neurons)
        self.widths = np.ones(self.num_rbf_neurons) * (max_dist / np.sqrt(2 *
self.num_rbf_neurons))
        print(f"RBF Widths set. Max inter-center distance: {max_dist:.4f},
Calculated width: {self.widths[0]:.4f}")

        # Step 3: Calculate RBF Activations for training data
        print("Calculating RBF activations...")
        H = self._calculate_rbf_activations(X_train)

        # Step 4: Solve for Output Weights using Least Squares
        # Add a bias term to the H matrix (column of ones)
        H_with_bias = np.concatenate((H, np.ones((H.shape[0], 1))), axis=1)
```

```
        # Solve H_with_bias * W_extended = y_train for W_extended
        # W_extended will contain [output_weights, output_bias]
        print("Solving for output weights using Least Squares...")
        # Use lstsq for robustness
        W_extended, residuals, rank, s = np.linalg.lstsq(H_with_bias,
y_train.reshape(-1, 1), rcond=None)

        self.output_weights = W_extended[:-1, :] # All but the last row are
weights
        self.output_bias = W_extended[-1, :] # The last row is the bias

        print("Training Complete. Output weights and bias determined.")

    def predict(self, X):
        """
        Makes predictions using the trained RBF network.

        Args:
            X (numpy.ndarray): Input features for prediction.

        Returns:
            numpy.ndarray: Predicted target values.
        """
        if self.centers is None or self.output_weights is None:
            raise Exception("RBF Network has not been trained yet. Call
.train() first.")

        H = self._calculate_rbf_activations(X)
        predictions = np.dot(H, self.output_weights) + self.output_bias
        return predictions.flatten() # Flatten to 1D array

# Example Usage: Regression Problem
if __name__ == "__main__":
    # Generate a synthetic regression dataset
    X, y = make_regression(n_samples=150, n_features=1, noise=15,
random_state=42) # Simple 1D regression

    # Data Preprocessing: Scale features
    scaler_X = StandardScaler()
    X_scaled = scaler_X.fit_transform(X)
    scaler_y = StandardScaler()
    y_scaled = scaler_y.fit_transform(y.reshape(-1, 1)).flatten()

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_scaled,
test_size=0.2, random_state=42)

    # Define number of RBF neurons
    num_rbf_neurons = 10

    # Create and train the RBF network
    rbf_trainer = RBFNetworkTrainer(num_rbf_neurons)
    rbf_trainer.train(X_train, y_train)

    print(f"\nLearned RBF Centers (first
3):\n{np.round(rbf_trainer.centers[:3], 2)}")
    print(f"Learned RBF Widths (first 3): {np.round(rbf_trainer.widths[:3],
2)}")
    print(f"Learned Output Weights (first
3):\n{np.round(rbf_trainer.output_weights[:3], 2)}")
    print(f"Learned Output Bias: {np.round(rbf_trainer.output_bias, 2)}")

    # Make predictions on test data
    y_pred_scaled = rbf_trainer.predict(X_test)
```

```
    # Inverse transform predictions and true values to original scale for
evaluation
    y_pred_original = scaler_y.inverse_transform(y_pred_scaled.reshape(-1,
1))
    y_test_original = scaler_y.inverse_transform(y_test.reshape(-1, 1))

    # Calculate Mean Squared Error on original scale
    mse = np.mean((y_test_original - y_pred_original)**2)
    print(f"\nTest MSE (original scale): {mse:.4f}")

    # Plotting the results (for 1D input)
    if X.shape[1] == 1:
        plt.figure(figsize=(10, 6))
        plt.scatter(scaler_X.inverse_transform(X_test), y_test_original,
label='True Test Data', alpha=0.7)
        plt.scatter(scaler_X.inverse_transform(X_test), y_pred_original,
color='red', marker='x', label='RBF Network Predictions')
        # Plot the learned function over a range
        X_range = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)
        X_range_scaled = scaler_X.transform(X_range)
        y_range_pred_scaled = rbf_trainer.predict(X_range_scaled)
        y_range_pred_original =
scaler_y.inverse_transform(y_range_pred_scaled.reshape(-1, 1))
        plt.plot(X_range, y_range_pred_original, color='green', linestyle='--
', label='Learned RBF Function')

        plt.xlabel('X')
        plt.ylabel('y')
        plt.title('RBF Network Regression')
        plt.legend()
        plt.grid(True)
        plt.show()
```

Input

```
# Synthetic regression dataset (generated by make_regression)
# X (features): 150 random values (1D for this example)
# y (targets): corresponding target values with noise
```

Expected Output

```
Starting RBF Network Training...
Running K-Means with 10 clusters...
K-Means complete. Centers determined.
RBF Widths set. Max inter-center distance: 5.1234, Calculated width: 1.1456
Calculating RBF activations...
Solving for output weights using Least Squares...
Training Complete. Output weights and bias determined.

Learned RBF Centers (first 3):
[[-1.85]
 [-0.3 ]
 [ 1.5 ]]
Learned RBF Widths (first 3): [1.15 1.15 1.15]
Learned Output Weights (first 3):
[[-0.12]
 [ 0.98]
 [ 0.56]]
Learned Output Bias: [0.01]

Test MSE (original scale): 200.5678
```

(A scatter plot showing true test data points and predicted points. A green dashed line representing the learned RBF function should roughly follow the trend of the data.)

(Note: Numerical values will vary. The MSE should be reasonably low, indicating that the network has learned the underlying relationship.)

Lab 13: Recurrent Neural Network (RNN) for Sequential Data

Aim

To implement an RNN for sequential data tasks such as time series prediction or text generation. Train the RNN on datasets like the IMDB movie review dataset or stock price data.

Procedure

1. **RNN Concept:** Understand that RNNs are designed to process sequential data by maintaining an internal "memory" (hidden state) that captures information from previous steps in the sequence.
2. **Dataset Preparation (Time Series):**
    - Load a time series dataset (e.g., synthetic sine wave data for simplicity, or actual stock price data).
    - **Sequence Creation:** Transform the time series into sequences of input-output pairs. For example, to predict the next value, an input sequence might be `[t, t+1, t+2]` and the target `t+3`.
    - **Normalization:** Normalize the data.
3. **RNN Architecture (Simple RNN Layer):**
    - Use a `SimpleRNN` layer (or `LSTM`/`GRU` for more complex tasks, but `SimpleRNN` is sufficient for basic understanding).
    - The input to an RNN layer typically has a shape of `(batch_size, timesteps, features)`.
    - The output layer depends on the task: a `Dense` layer with linear activation for regression (time series prediction) or `softmax` for classification/text generation (predicting next character/word).
4. **Compilation:** Compile the model with an appropriate optimizer and loss function (e.g., MSE for time series prediction).
5. **Training:** Train the RNN on the prepared sequential data.
6. **Prediction/Generation:** Use the trained RNN to predict future values in a time series or generate new sequences of text.

Source Code

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

# Suppress TensorFlow warnings
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

class SimpleRNNPredictor:
    def __init__(self, input_features, timesteps, rnn_units, output_dim):
        """
        Initializes a simple RNN for time series prediction.

        Args:
            input_features (int): Number of features per timestep (e.g., 1
for univariate time series).
            timesteps (int): Length of the input sequence.
            rnn_units (int): Number of units in the SimpleRNN layer.
            output_dim (int): Dimension of the output (e.g., 1 for predicting
next value).
        """
        self.input_features = input_features
```

```
        self.timesteps = timesteps
        self.rnn_units = rnn_units
        self.output_dim = output_dim
        self.model = self._build_model()

    def _build_model(self):
        """
        Builds the Keras Sequential model with a SimpleRNN layer.
        """
        model = tf.keras.Sequential([
            tf.keras.layers.SimpleRNN(self.rnn_units, activation='relu',
input_shape=(self.timesteps, self.input_features)),
            tf.keras.layers.Dense(self.output_dim) # Linear activation for
regression
        ])
        model.compile(optimizer='adam', loss='mse') # Mean Squared Error for
time series prediction
        return model

    def train(self, X_train, y_train, epochs=50, batch_size=32,
validation_split=0.2):
        """
        Trains the RNN model.

        Args:
            X_train (numpy.ndarray): Training input sequences.
            y_train (numpy.ndarray): Training target values.
            epochs (int): Number of training epochs.
            batch_size (int): Batch size for training.
            validation_split (float): Fraction of training data for
validation.
        """
        print("Starting Simple RNN Training for Time Series Prediction...")
        history = self.model.fit(X_train, y_train,
                                 epochs=epochs,
                                 batch_size=batch_size,
                                 validation_split=validation_split,
                                 verbose=0)
        print("Training Complete.")
        return history

    def predict(self, X_new):
        """
        Makes predictions on new sequences.

        Args:
            X_new (numpy.ndarray): New input sequences for prediction.

        Returns:
            numpy.ndarray: Predicted values.
        """
        return self.model.predict(X_new)

# Helper function to create sequences for time series prediction
def create_sequences(data, timesteps):
    X, y = [], []
    for i in range(len(data) - timesteps):
        X.append(data[i:(i + timesteps)])
        y.append(data[i + timesteps])
    return np.array(X), np.array(y)

# Example Usage: Synthetic Sine Wave Time Series Prediction
if __name__ == "__main__":
    # Generate synthetic sine wave data
    time = np.arange(0, 100, 0.1)
```

```
    data = np.sin(time) + np.random.normal(0, 0.1, len(time)) # Sine wave
with some noise

    # Reshape data for scaling (needs to be 2D)
    data = data.reshape(-1, 1)

    # Normalize data
    scaler = MinMaxScaler(feature_range=(0, 1))
    data_scaled = scaler.fit_transform(data)

    # Define sequence parameters
    timesteps = 10 # Predict the next value based on the previous 10 values
    input_features = 1 # Univariate time series

    # Create sequences
    X, y = create_sequences(data_scaled, timesteps)

    # Split data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

    # Define RNN parameters
    rnn_units = 50
    output_dim = 1

    # Create and train the RNN
    rnn_predictor = SimpleRNNPredictor(input_features, timesteps, rnn_units,
output_dim)
    rnn_predictor.train(X_train, y_train, epochs=100)

    # Make predictions on test data
    y_pred_scaled = rnn_predictor.predict(X_test)

    # Inverse transform predictions and true values to original scale
    y_pred_original = scaler.inverse_transform(y_pred_scaled)
    y_test_original = scaler.inverse_transform(y_test)

    # Calculate MSE
    mse = np.mean((y_test_original - y_pred_original)**2)
    print(f"\nTest MSE (original scale): {mse:.4f}")

    # Plotting actual vs. predicted values
    plt.figure(figsize=(12, 6))
    plt.plot(y_test_original, label='True Values', color='blue')
    plt.plot(y_pred_original, label='Predicted Values', color='red',
linestyle='--')
    plt.title('Time Series Prediction with Simple RNN')
    plt.xlabel('Time Step (Test Set)')
    plt.ylabel('Value')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Demonstrate prediction for a future sequence
    # Take the last 'timesteps' values from the original data
    last_sequence_scaled = data_scaled[-timesteps:].reshape(1, timesteps,
input_features)
    future_prediction_scaled = rnn_predictor.predict(last_sequence_scaled)
    future_prediction_original =
scaler.inverse_transform(future_prediction_scaled)
    print(f"\nPrediction for the next value after the dataset:
{future_prediction_original[0][0]:.4f}")
```

Input

```
# Synthetic sine wave data with noise (generated by the script)
# Example (after scaling and sequence creation, X is 3D, y is 2D):
# X_train (sequences):
# [[[0.51]
#   [0.55]
#   ...
#   [0.61]]]
# y_train (next values):
# [[0.65]
#  [0.68]
#  ...
#  [0.72]]
```

Expected Output

```
Starting Simple RNN Training for Time Series Prediction...
Training Complete.

Test MSE (original scale): 0.0123

(A plot showing the true time series values from the test set and the RNN's
predicted values, which should closely follow the true values.)

Prediction for the next value after the dataset: 0.7567
```

(Note: MSE and prediction values will vary. The plot should show a good fit between true and predicted values.)

Lab 14: Convolutional Neural Network (CNN) for Image Classification

Aim

To implement a CNN for image classification tasks using libraries like TensorFlow or PyTorch, and train the CNN on datasets like MNIST.

Procedure

1. **CNN Concept:** Understand that CNNs are particularly effective for image data due to their ability to automatically learn hierarchical features using convolutional layers, pooling layers, and fully connected layers.
2. **Dataset Loading and Preprocessing:**
    - Load the MNIST handwritten digit dataset.
    - **Normalization:** Normalize pixel values to a range suitable for neural networks (e.g., [0, 1]).
    - **Reshaping:** Reshape the images to include a channel dimension (e.g., `(batch_size, height, width, channels)` for TensorFlow/Keras). For grayscale MNIST, channels = 1.
    - **One-Hot Encoding (for labels):** Convert integer labels to one-hot encoded vectors if using `categorical_crossentropy` loss. If using `sparse_categorical_crossentropy`, this step is not needed.
3. **CNN Architecture:**
    - **Convolutional Layers (`Conv2D`):** Apply filters to learn spatial features.
    - **Activation Functions:** Use non-linear activation functions (e.g., ReLU) after convolutional layers.
    - **Pooling Layers (`MaxPooling2D`):** Downsample feature maps, reducing dimensionality and making the model more robust to small shifts.
    - **Flatten Layer:** Convert the 2D feature maps from convolutional/pooling layers into a 1D vector before feeding into fully connected layers.
    - **Dense (Fully Connected) Layers:** Standard neural network layers for classification.
    - **Output Layer:** A `Dense` layer with `softmax` activation for multi-class classification, with the number of units equal to the number of classes.
4. **Compilation:** Compile the model with an optimizer (e.g., 'adam'), a loss function (e.g., `sparse_categorical_crossentropy`), and metrics (e.g., 'accuracy').
5. **Training:** Train the CNN on the prepared image data.
6. **Evaluation:** Evaluate the trained model's performance on a test set.
7. **Prediction:** Use the trained model to classify new images.

Source Code

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# Suppress TensorFlow warnings
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

class MNISTCNNClassifier:
    def __init__(self, input_shape, num_classes):
        """
        Initializes a CNN for MNIST image classification.

        Args:
```

```
            input_shape (tuple): Shape of the input images (height, width,
channels).
            num_classes (int): Number of output classes (0-9 for MNIST).
        """
        self.input_shape = input_shape
        self.num_classes = num_classes
        self.model = self._build_model()

    def _build_model(self):
        """
        Builds the Keras Sequential CNN model.
        """
        model = tf.keras.Sequential([
            # Convolutional Layer 1
            tf.keras.layers.Conv2D(32, (3, 3), activation='relu',
input_shape=self.input_shape),
            tf.keras.layers.MaxPooling2D((2, 2)),

            # Convolutional Layer 2
            tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
            tf.keras.layers.MaxPooling2D((2, 2)),

            # Flatten the output for the fully connected layers
            tf.keras.layers.Flatten(),

            # Dense Hidden Layer
            tf.keras.layers.Dense(128, activation='relu'),
            tf.keras.layers.Dropout(0.5), # Dropout for regularization

            # Output Layer
            tf.keras.layers.Dense(self.num_classes, activation='softmax')
        ])

        # Compile the model
        model.compile(optimizer='adam',
                      loss='sparse_categorical_crossentropy', # Use for
integer labels
                      metrics=['accuracy'])
        return model

    def train(self, X_train, y_train, epochs=10, batch_size=64,
validation_split=0.1):
        """
        Trains the CNN model on MNIST data.

        Args:
            X_train (numpy.ndarray): Training images.
            y_train (numpy.ndarray): Training labels.
            epochs (int): Number of training epochs.
            batch_size (int): Batch size for training.
            validation_split (float): Fraction of training data for
validation.
        """
        print("Starting MNIST CNN Training...")
        history = self.model.fit(X_train, y_train,
                                 epochs=epochs,
                                 batch_size=batch_size,
                                 validation_split=validation_split,
                                 verbose=1) # Set verbose to 1 to see
progress
        print("Training Complete.")
        return history

    def evaluate(self, X_test, y_test):
        """
        Evaluates the trained model on test data.
```

```
        Args:
            X_test (numpy.ndarray): Test images.
            y_test (numpy.ndarray): Test labels.

        Returns:
            tuple: Loss and accuracy on the test set.
        """
        print("\nEvaluating Model on Test Data...")
        loss, accuracy = self.model.evaluate(X_test, y_test, verbose=0)
        print(f"Test Loss: {loss:.4f}")
        print(f"Test Accuracy: {accuracy * 100:.2f}%")
        return loss, accuracy

    def predict(self, X_new):
        """
        Makes predictions on new images.

        Args:
            X_new (numpy.ndarray): New images for prediction.

        Returns:
            numpy.ndarray: Predicted class probabilities.
        """
        return self.model.predict(X_new)

# Example Usage: MNIST Dataset
if __name__ == "__main__":
    # Load MNIST dataset
    (X_train, y_train), (X_test, y_test) =
tf.keras.datasets.mnist.load_data()

    # Data Preprocessing
    # Normalize pixel values to [0, 1]
    X_train = X_train.astype('float32') / 255.0
    X_test = X_test.astype('float32') / 255.0

    # Reshape images to include channel dimension (for grayscale, channels=1)
    # Keras expects (batch_size, height, width, channels)
    X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
    X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)

    # Define CNN parameters
    input_shape = (28, 28, 1)
    num_classes = 10 # Digits 0-9

    # Create and train the CNN
    mnist_cnn = MNISTCNNClassifier(input_shape, num_classes)
    mnist_cnn.train(X_train, y_train, epochs=5) # Train for fewer epochs for
quick demo

    # Evaluate the model
    mnist_cnn.evaluate(X_test, y_test)

    # Make predictions on a few test samples
    num_samples_to_predict = 5
    sample_indices = np.random.choice(len(X_test), num_samples_to_predict,
replace=False)
    X_new_samples = X_test[sample_indices]
    y_true_samples = y_test[sample_indices]

    predictions_proba = mnist_cnn.predict(X_new_samples)
    predicted_classes = np.argmax(predictions_proba, axis=1)

    print("\nSample Predictions:")
    plt.figure(figsize=(10, 4))
```

```
    for i in range(num_samples_to_predict):
        plt.subplot(1, num_samples_to_predict, i + 1)
        plt.imshow(X_new_samples[i].reshape(28, 28), cmap='gray')
        plt.title(f"True: {y_true_samples[i]}\nPred: {predicted_classes[i]}")
        plt.axis('off')
    plt.tight_layout()
    plt.show()
```

Input

```
# MNIST dataset (loaded internally by tf.keras.datasets.mnist.load_data())
# X_train, X_test are arrays of 28x28 grayscale images.
# After preprocessing, they become 4D arrays (num_samples, 28, 28, 1) with
pixel values [0, 1].
# y_train, y_test are integer labels (0-9).
```

Expected Output

```
Starting MNIST CNN Training...
Epoch 1/5
...
Epoch 5/5
Training Complete.

Evaluating Model on Test Data...
Test Loss: 0.0456
Test Accuracy: 98.50%

Sample Predictions:
(A plot showing 5 random MNIST digits from the test set. Each subplot will
display the image, its true label, and the CNN's predicted label. The
predicted labels should mostly match the true labels.)
```

(Note: Loss and accuracy will vary slightly. The plot will visually confirm the CNN's ability to classify handwritten digits.)

Lab 15: Basic Generative Adversarial Network (GAN)

Aim

To implement a basic Generative Adversarial Network (GAN) for generating synthetic data samples. Train the GAN on datasets like the MNIST dataset for generating handwritten digits.

Procedure

1. **GAN Concept:** Understand that a GAN consists of two neural networks:
    - **Generator (G):** Takes random noise as input and generates synthetic data samples (e.g., images).
    - **Discriminator (D):** Takes a data sample (either real from the dataset or fake from the generator) and tries to classify it as "real" or "fake." The two networks are trained in a competitive (adversarial) manner: the generator tries to fool the discriminator, and the discriminator tries to correctly distinguish real from fake.
2. **Dataset Preparation (MNIST):**
    - Load the MNIST dataset.
    - Normalize pixel values to a range suitable for the generator's output activation (e.g., [-1, 1] for `tanh` output, or [0, 1] for `sigmoid`).
    - Flatten the images if using fully connected layers in the generator/discriminator.
3. **Generator Architecture:**
    - Input: Random noise vector (latent space).
    - Layers: Typically `Dense` layers, followed by `LeakyReLU` activations (or `ReLU`).
    - Output: A `Dense` layer with `tanh` or `sigmoid` activation to match the data's pixel range and shape (e.g., 784 units for flattened MNIST).
4. **Discriminator Architecture:**
    - Input: A data sample (real or fake image).
    - Layers: Typically `Dense` layers, followed by `LeakyReLU` activations.
    - Output: A `Dense` layer with `sigmoid` activation to output a single probability (0 for fake, 1 for real).
5. **Compile Discriminator:** Compile the discriminator as a standalone binary classifier.
6. **Build GAN Model:**
    - Combine the generator and discriminator into a single GAN model for training the generator.
    - **Crucially:** When training the GAN model, set the discriminator's layers to be non-trainable (`discriminator.trainable = False`) so that only the generator's weights are updated based on the discriminator's feedback.
7. **Training Loop:**
    - Iterate for a specified number of epochs.
    - **Train Discriminator:**
        - Generate fake images using the current generator.
        - Get real images from the dataset.
        - Train the discriminator on real images (label 1) and fake images (label 0).
    - **Train Generator (via GAN model):**
        - Generate new random noise.
        - Train the combined GAN model. The discriminator's weights are frozen, so the loss backpropagates only through the generator, trying to make the fake images classified as real (label 1).
    - **Monitor Progress:** Periodically save generated images to observe the generator's learning progress.

Source Code

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# Suppress TensorFlow warnings
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

class GAN:
    def __init__(self, img_rows, img_cols, channels, latent_dim=100):
        """
        Initializes the GAN components (Generator and Discriminator).

        Args:
            img_rows (int): Number of rows in the image (e.g., 28 for MNIST).
            img_cols (int): Number of columns in the image (e.g., 28 for
MNIST).
            channels (int): Number of color channels (e.g., 1 for grayscale
MNIST).
            latent_dim (int): Dimension of the noise vector (input to
generator).
        """
        self.img_rows = img_rows
        self.img_cols = img_cols
        self.channels = channels
        self.img_flat_dim = self.img_rows * self.img_cols * self.channels
        self.latent_dim = latent_dim

        optimizer = tf.keras.optimizers.Adam(0.0002, 0.5) # Adam optimizer
with learning rate and beta_1

        # Build and compile the discriminator
        self.discriminator = self._build_discriminator()
        self.discriminator.compile(loss='binary_crossentropy',
                                   optimizer=optimizer,
                                   metrics=['accuracy'])

        # Build the generator
        self.generator = self._build_generator()

        # The generator takes noise as input and generates images
        z = tf.keras.layers.Input(shape=(self.latent_dim,))
        img = self.generator(z)

        # For the combined model, we will only train the generator
        # The discriminator's weights should be fixed
        self.discriminator.trainable = False

        # The discriminator takes generated images as input and determines
validity
        validity = self.discriminator(img)

        # The combined model (stacked generator and discriminator)
        # Trains the generator to fool the discriminator
        self.combined = tf.keras.Model(z, validity)
        self.combined.compile(loss='binary_crossentropy',
optimizer=optimizer)

    def _build_generator(self):
        """
        Builds the generator model.
        """
        model = tf.keras.Sequential([
            tf.keras.layers.Dense(256, input_dim=self.latent_dim),
            tf.keras.layers.LeakyReLU(alpha=0.2), # LeakyReLU for generator
```

```
            tf.keras.layers.BatchNormalization(momentum=0.8), # Batch
Normalization
            tf.keras.layers.Dense(512),
            tf.keras.layers.LeakyReLU(alpha=0.2),
            tf.keras.layers.BatchNormalization(momentum=0.8),
            tf.keras.layers.Dense(1024),
            tf.keras.layers.LeakyReLU(alpha=0.2),
            tf.keras.layers.BatchNormalization(momentum=0.8),
            tf.keras.layers.Dense(self.img_flat_dim, activation='tanh') #
Tanh for output in [-1, 1]
        ])
        noise = tf.keras.layers.Input(shape=(self.latent_dim,))
        img = model(noise)
        return tf.keras.Model(noise, img)

    def _build_discriminator(self):
        """
        Builds the discriminator model.
        """
        model = tf.keras.Sequential([
            tf.keras.layers.Dense(512, input_dim=self.img_flat_dim),
            tf.keras.layers.LeakyReLU(alpha=0.2),
            tf.keras.layers.Dense(256),
            tf.keras.layers.LeakyReLU(alpha=0.2),
            tf.keras.layers.Dense(1, activation='sigmoid') # Sigmoid for
binary classification (real/fake)
        ])
        img = tf.keras.layers.Input(shape=(self.img_flat_dim,))
        validity = model(img)
        return tf.keras.Model(img, validity)

    def train(self, X_train, epochs, batch_size=128, sample_interval=100):
        """
        Trains the GAN.

        Args:
            X_train (numpy.ndarray): Training images (normalized and
flattened).
            epochs (int): Number of training epochs.
            batch_size (int): Batch size for training.
            sample_interval (int): Interval (in epochs) at which to save
generated images.
        """
        # Adversarial ground truths
        valid = np.ones((batch_size, 1))
        fake = np.zeros((batch_size, 1))

        print("Starting GAN Training...")
        for epoch in range(epochs):
            # ---------------------
            #  Train Discriminator
            # ---------------------

            # Select a random batch of real images
            idx = np.random.randint(0, X_train.shape[0], batch_size)
            real_imgs = X_train[idx]

            # Generate a batch of fake images
            noise = np.random.normal(0, 1, (batch_size, self.latent_dim))
            gen_imgs = self.generator.predict(noise, verbose=0)

            # Train the discriminator
            d_loss_real = self.discriminator.train_on_batch(real_imgs, valid)
            d_loss_fake = self.discriminator.train_on_batch(gen_imgs, fake)
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
```

```
            # ---------------------
            #  Train Generator
            # ---------------------

            # Generate new noise for generator training
            noise = np.random.normal(0, 1, (batch_size, self.latent_dim))

            # Train the generator (discriminator weights are frozen)
            g_loss = self.combined.train_on_batch(noise, valid)

            # Print the progress
            if (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch + 1}/{epochs} [D loss: {d_loss[0]:.4f},
acc.: {100*d_loss[1]:.2f}%] [G loss: {g_loss:.4f}]")

            # If at sample interval, save generated image samples
            if (epoch + 1) % sample_interval == 0:
                self.sample_images(epoch + 1)
        print("Training Complete.")

    def sample_images(self, epoch):
        """
        Generates and saves a grid of synthetic images.
        """
        r, c = 5, 5 # 5x5 grid of images
        noise = np.random.normal(0, 1, (r * c, self.latent_dim))
        gen_imgs = self.generator.predict(noise, verbose=0)

        # Rescale images from [-1, 1] to [0, 1] for plotting
        gen_imgs = 0.5 * gen_imgs + 0.5

        plt.figure(figsize=(10, 10))
        for i in range(r * c):
            plt.subplot(r, c, i + 1)
            plt.imshow(gen_imgs[i].reshape(self.img_rows, self.img_cols),
cmap='gray')
            plt.axis('off')
        plt.suptitle(f"GAN Generated Images - Epoch {epoch}")
        plt.tight_layout(rect=[0, 0.03, 1, 0.95]) # Adjust layout to prevent
title overlap
        plt.show()

# Example Usage: MNIST Dataset
if __name__ == "__main__":
    # Load MNIST dataset
    (X_train, _), (_, _) = tf.keras.datasets.mnist.load_data()

    # Data Preprocessing
    # Rescale images to [-1, 1] (suitable for tanh activation in generator
output)
    X_train = (X_train.astype('float32') - 127.5) / 127.5
    # Flatten images
    X_train = X_train.reshape(X_train.shape[0], 28 * 28 * 1)

    # Define GAN parameters
    img_rows, img_cols, channels = 28, 28, 1
    latent_dim = 100 # Size of the noise vector

    # Create and train the GAN
    gan = GAN(img_rows, img_cols, channels, latent_dim)
    gan.train(X_train, epochs=5000, batch_size=256, sample_interval=1000)

    # After training, you can generate more images
    print("\nGenerating final sample of images:")
    gan.sample_images(epoch="Final")
```

Input

```
# MNIST dataset (loaded internally by tf.keras.datasets.mnist.load_data())
# X_train is an array of 28x28 grayscale images.
# After preprocessing, X_train becomes a 2D array (num_samples, 784) with
pixel values in [-1, 1].
```

Expected Output

```
Starting GAN Training...
Epoch 100/5000 [D loss: 0.6987, acc.: 50.00%] [G loss: 0.6876]
...
Epoch 1000/5000 [D loss: 0.6876, acc.: 55.00%] [G loss: 0.7012]
(A plot showing a 5x5 grid of generated images at Epoch 1000. These images
will likely be very noisy and not clearly resemble digits.)
...
Epoch 2000/5000 [D loss: 0.6754, acc.: 60.00%] [G loss: 0.7234]
(A plot showing a 5x5 grid of generated images at Epoch 2000. Images should
start to show some structure of digits.)
...
Epoch 5000/5000 [D loss: 0.6901, acc.: 52.00%] [G loss: 0.6950]
Training Complete.

Generating final sample of images:
(A final plot showing a 5x5 grid of generated images. These images should
resemble handwritten digits, though they might still be blurry or imperfect.
The discriminator accuracy will hover around 50% as the generator gets better
at fooling it, and generator loss will also be around 0.69.)
```

(Note: GAN training is notoriously unstable. The loss values and image quality will vary significantly depending on random initialization and hyperparameters. The goal is to observe the generator producing increasingly realistic images over time.)