# Spectral Neural Networks: A Universal Approximation Framework via Spectral Decomposition

Orges Leka

November 1, 2024

**Abstract**

Neural network architectures that draw inspiration from physical interactions have garnered significant interest due to their structured parameterizations and potential computational efficiencies. This paper introduces the concept of *Spectral Neural Networks*, where the weight matrices are parameterized through spectral (eigen-decomposition) representations. We investigate the representational capacity of such networks, particularly focusing on their ability to universally approximate any symmetric weight matrix. By leveraging the spectral theorem, we establish conditions under which spectral neural networks achieve universal approximation. Furthermore, we discuss the implications of dimensionality constraints, propose strategies to enhance the expressiveness of these networks under limited spectral dimensions, and elucidate the learning mechanisms involving gradient descent and backpropagation within this framework.

## 1 Introduction

The universal approximation capability of neural networks underpins their widespread applicability in diverse machine learning tasks. Traditional architectures, such as multilayer perceptrons (MLPs), achieve this by employing dense weight matrices that can represent arbitrary linear transformations given sufficient capacity. However, as the scale of neural networks grows, the parameterization of weight matrices becomes a computational bottleneck, motivating the exploration of more structured approaches.

In this context, we explore a neural network architecture inspired by physical interactions, where each neuron is associated with "mass" and "position" vectors. The weight matrix between neurons is computed using a force-like inverse-square law interaction, analogous to the Coulomb interaction between charged particles. Specifically, for neurons with mass vectors $\mu_i$ and $\mu_j$ located at positions $x_i$ and $x_j$, the weight $w_{ij}$ is defined as:

$$w_{ij} = \frac{\mu_i \cdot \mu_j}{\|x_i - x_j\|^2}$$

This formulation bears structural resemblance to the *Coulomb matrix* used in quantum chemistry to represent atomic interactions in molecules.

The central inquiry of this paper is to determine the conditions under which a general symmetric weight matrix $W$ can be represented in the form of a Coulomb-like matrix. Formally, we seek to establish when there exists a set of vectors $\{\mu_i, x_i\}$ such that:

$$w_{ij} = \frac{\mu_i \cdot \mu_j}{\|x_i - x_j\|^2} \quad \forall i, j$$

Subsequently, we refine the problem to the case where the diagonal of $W$ is zero:

$$w_{ij} = \mu_i \cdot \mu_j \cdot \|x_i - x_j\|^2 \quad \forall i \neq j$$

This simplification aids in the analytical tractability of the representation.

# 2 Spectral Neural Networks

## 2.1 Spectral Parameterization of Weight Matrices

A *Spectral Neural Network* parameterizes its weight matrices via spectral decomposition. Specifically, a symmetric weight matrix $W \in \mathbb{R}^{n \times n}$ is expressed as:

$$W = Q \Lambda Q^\top$$

where:

- $Q \in \mathbb{R}^{n \times d}$ is a matrix whose columns are vectors $q_k$, analogous to eigenvectors.
- $\Lambda \in \mathbb{R}^d$ is a vector containing scaling factors $\lambda_k$, analogous to eigenvalues.
- $d$ is the dimensionality parameter, potentially much smaller than $n$, the number of neurons.

This formulation draws inspiration from the *Spectral Theorem*, which asserts that any real symmetric matrix can be decomposed into its eigenvectors and eigenvalues.

## 2.2 Relation to Coulomb-like Matrices

In the original formulation, weights are defined via interactions resembling Coulomb forces:

$$w_{ij} = \frac{\mu_i \cdot \mu_j}{\|x_i - x_j\|^2}$$

However, for analytical simplicity, we consider the modified representation without division by the distance squared:

$$w_{ij} = \mu_i \cdot \mu_j \cdot \|x_i - x_j\|^2 \quad \forall i \neq j$$

This adjustment facilitates the exploration of the representational capacity without singularities arising from zero distances.

# 3 Universal Approximation with Spectral Neural Networks

## 3.1 Universal Approximation Theorem

**Theorem 1.** *A Spectral Neural Network can universally approximate any symmetric weight matrix $W \in \mathbb{R}^{n \times n}$ if and only if the dimensionality parameter $d$ satisfies $d \geq n$.*

## 3.2 Proof of Theorem 1

**Sufficiency ($d \geq n$):**

By the *Spectral Theorem*, any real symmetric matrix $W \in \mathbb{R}^{n \times n}$ can be decomposed as:

$$W = Q \Lambda Q^\top$$

where $Q$ is an orthogonal matrix ($Q^\top Q = I$) and $\Lambda$ is a diagonal matrix containing the eigenvalues of $W$.

When $d \geq n$, $Q \in \mathbb{R}^{n \times n}$ can encompass all the eigenvectors of $W$, and $\Lambda \in \mathbb{R}^n$ can include all corresponding eigenvalues. Thus, the spectral parameterization can exactly represent any symmetric weight matrix $W$.

**Necessity ($d < n$):**

If $d < n$, the matrix $Q \Lambda Q^\top$ has rank at most $d$, since it is the product of an $n \times d$ matrix, a $d$-dimensional diagonal matrix, and a $d \times n$ matrix. Consequently, only symmetric matrices of rank $\leq d$ can be exactly represented. However, there exist symmetric matrices with rank $> d$, which cannot be represented by the spectral parameterization when $d < n$. Therefore, exact universal approximation is unattainable in this regime.

### 3.3 Implications of Theorem 1

The theorem establishes that spectral neural networks achieve universal approximation of symmetric weight matrices when the spectral dimensionality meets or exceeds the number of neurons. However, practical scenarios often require $d \ll n$ for computational efficiency, necessitating approximate representations through low-rank decompositions.

## 4 Learning in Spectral Neural Networks

### 4.1 Network Parameterization

In the Spectral Neural Network framework, the entire network's weight matrices are parameterized through spectral components. Specifically, each weight matrix $W$ in the network is represented as:

$$W^{(l)} = Q^{(l)}\Lambda^{(l)}(Q^{(l)})^\top$$

for each layer $l$, where:

- $Q^{(l)} \in \mathbb{R}^{n_l \times d_l}$ contains the spectral vectors $q_k^{(l)}$ for layer $l$.
- $\Lambda^{(l)} \in \mathbb{R}^{d_l}$ is the scaling vector $\lambda_k^{(l)}$ for layer $l$.
- $d_l$ is the spectral dimensionality for layer $l$.

Each neuron within the network is associated with a vector $q_i^{(l)}$ in $Q^{(l)}$, encapsulating its spectral properties. The scaling factors $\Lambda^{(l)}$ modulate the contributions of each spectral component to the overall weight matrix.

### 4.2 Learning Mechanism

The learning process in Spectral Neural Networks involves optimizing the spectral parameters $Q^{(l)}$ and $\Lambda^{(l)}$ across all layers. This optimization is performed using gradient descent in conjunction with backpropagation to minimize a predefined loss function $L$.

#### 4.2.1 Gradient Descent

Given a loss function $L$, the objective is to minimize $L$ with respect to the spectral parameters. The update rules for gradient descent are as follows:

$$Q^{(l)} \leftarrow Q^{(l)} - \eta \frac{\partial L}{\partial Q^{(l)}}$$

$$\Lambda^{(l)} \leftarrow \Lambda^{(l)} - \eta \frac{\partial L}{\partial \Lambda^{(l)}}$$

where $\eta$ is the learning rate.

#### 4.2.2 Backpropagation

Backpropagation is employed to compute the gradients $\frac{\partial L}{\partial Q^{(l)}}$ and $\frac{\partial L}{\partial \Lambda^{(l)}}$ efficiently. The computation leverages the chain rule to propagate errors from the output layer backward through each spectral parameterized layer.

For each layer $l$, the gradient with respect to $Q^{(l)}$ is derived as:

$$\frac{\partial L}{\partial Q^{(l)}} = \frac{\partial L}{\partial W^{(l)}} \cdot \frac{\partial W^{(l)}}{\partial Q^{(l)}} = \frac{\partial L}{\partial W^{(l)}} \cdot \left(\Lambda^{(l)}(Q^{(l)})^\top + Q^{(l)}(\Lambda^{(l)})^\top\right)$$

Similarly, the gradient with respect to $\Lambda^{(l)}$ is:

$$\frac{\partial L}{\partial \Lambda^{(l)}} = \text{diag}\left(Q^{(l)\top} \cdot \frac{\partial L}{\partial W^{(l)}} \cdot Q^{(l)}\right)$$

where $\text{diag}(\cdot)$ extracts the diagonal elements corresponding to the scaling factors.

### 4.3 Parameter Efficiency

By parameterizing weight matrices through spectral decomposition, Spectral Neural Networks can achieve significant parameter efficiency, especially when $d_l \ll n_l$ for each layer $l$. This reduction from $O(n_l^2)$ to $O(n_l d_l)$ parameters not only conserves memory but also potentially enhances learning dynamics by focusing on the most salient spectral components.

## 5 Enhancing Expressiveness with Limited Spectral Dimensionality

While universal approximation is guaranteed for $d \geq n$, practical applications benefit from lower dimensionality. To mitigate the expressiveness limitations when $d < n$, the following strategies are proposed:

### 5.1 Hierarchical Spectral Decomposition

Introduce multiple layers of spectral decompositions within the network architecture. Each layer captures distinct aspects or frequency components of the transformations, thereby compensating for the reduced dimensionality in individual layers.

### 5.2 Sparsity Constraints

Impose sparsity on the matrices $Q^{(l)}$ or the vectors $\Lambda^{(l)}$. Sparse representations focus on the most significant interactions, effectively utilizing limited dimensionality to capture essential patterns within the weight matrices.

### 5.3 Nonlinear Transformations

Incorporate nonlinear activation functions between spectral layers. Nonlinearities enable the network to model complex interactions and compensate for the expressiveness loss due to reduced spectral dimensionality.

### 5.4 Adaptive Dimensionality

Allow the spectral dimensionality parameter $d_l$ to adapt dynamically during training. Starting with a small $d_l$, the network can incrementally increase dimensionality based on the complexity of the task, balancing computational efficiency with representational capacity.

## 6 Conclusion

This paper introduces Spectral Neural Networks, a novel architecture wherein symmetric weight matrices are parameterized via spectral decomposition. We established that such networks achieve universal approximation of symmetric weight matrices provided the spectral dimensionality meets or exceeds the number of neurons. While exact universality necessitates $d \geq n$, practical applications can leverage strategies such as hierarchical decomposition, sparsity, nonlinear

transformations, and adaptive dimensionality to enhance expressiveness under computational constraints.

Furthermore, we elucidated the learning mechanisms within Spectral Neural Networks, highlighting how gradient descent and backpropagation facilitate the optimization of spectral parameters. By structuring the network's parameters through $\lambda$ vectors and $q$ vectors for each neuron, the framework not only ensures parameter efficiency but also integrates seamlessly with established optimization techniques in neural network training.

The spectral parameterization offers a structured and potentially memory-efficient representation of weight matrices, bridging concepts from physics-inspired interactions and linear algebraic decompositions. Future work may explore empirical validations of the proposed theoretical insights and the integration of spectral methods with other architectural innovations to further optimize performance and scalability.

## References


1. Horn, R. A., & Johnson, C. R. (2012). *Matrix Analysis*. Cambridge University Press.

2. Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), 359-366.

3. Saxe, A. M., McClelland, J. L., & Ganguli, S. (2013). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*.

4. Candes, E. J., & Recht, B. (2009). Exact matrix completion via convex optimization. *Foundations of Computational Mathematics*, 9(6), 717-772.

5. Coulomb, C. A. (1785). *Recherches sur la théorie des phénomènes électro-dynamiques*.


## A Appendix: Python Implementation of Spectral Neural Networks

Below is the Python code used for implementing and experimenting with Spectral Neural Networks. The code includes the definition of activation functions, loss functions, the spectral layer, the spectral neural network class, and example usage on XOR, classification, and regression tasks.

```
import numpy as np
from sklearn.datasets import load_breast_cancer, load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    accuracy_score,
    mean_squared_error,
    matthews_corrcoef,
    r2_score,
    confusion_matrix,
    roc_curve,
    auc
)
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import LogisticRegression, LinearRegression
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
import matplotlib.pyplot as plt
import seaborn as sns

```

```python
# Activation functions and their derivatives
def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return (x > 0).astype(float)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    s = sigmoid(x)
    return s * (1 - s)

# Loss functions and their derivatives
def mse_loss(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

def mse_loss_derivative(y_true, y_pred):
    return 2 * (y_pred - y_true) / y_true.size

def binary_cross_entropy(y_true, y_pred):
    # Adding epsilon for numerical stability
    epsilon = 1e-12
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred)
    )

def binary_cross_entropy_derivative(y_true, y_pred):
    # Adding epsilon for numerical stability
    epsilon = 1e-12
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return (-(y_true / y_pred) + (1 - y_true) / (1 - y_pred)) / y_true.size

# Spectral Layer Class for Rectangular Weight Matrices
class SpectralLayer:
    def __init__(self, input_dim, output_dim, spectral_dim, activation='relu'):
        """
        Initializes the Spectral Layer.

        Parameters:
        - input_dim: Number of input neurons.
        - output_dim: Number of output neurons.
        - spectral_dim: Spectral dimensionality 'd'.
        - activation: Activation function ('relu' or 'sigmoid').
        """
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.spectral_dim = spectral_dim

        # Initialize Q and P with He initialization for better convergence
        # Q: (output_dim, d)
        self.Q = np.random.randn(output_dim, spectral_dim) * np.sqrt(2. / (
    output_dim + spectral_dim))
        # P: (input_dim, d)
        self.P = np.random.randn(input_dim, spectral_dim) * np.sqrt(2. / (
    input_dim + spectral_dim))
        # Lambda: (d,)
        self.Lambda = np.random.randn(spectral_dim) * 0.1

        # Initialize biases
        self.b = np.zeros((output_dim,))

```

```
        # Activation function
        if activation == 'relu':
            self.activation = relu
            self.activation_derivative = relu_derivative
        elif activation == 'sigmoid':
            self.activation = sigmoid
            self.activation_derivative = sigmoid_derivative
        else:
            raise ValueError("Unsupported activation function")

        # Placeholders for forward and backward pass
        self.x = None
        self.z = None
        self.a = None
        self.dQ = None
        self.dP = None
        self.dLambda = None
        self.db = None

    def forward(self, x):
        """
        Forward pass through the spectral layer.

        Parameters:
        - x: Input data of shape (batch_size, input_dim)

        Returns:
        - a: Activated output of shape (batch_size, output_dim)
        """
        self.x = x  # (batch_size, input_dim)

        # Compute W = Q * diag(Lambda) * P.T
        # Q_Lambda: (output_dim, d)
        Q_Lambda = self.Q * self.Lambda  # Broadcasting (output_dim, d)
        # Compute W = Q_Lambda @ P.T -> (output_dim, input_dim)
        self.W = Q_Lambda @ self.P.T  # (output_dim, input_dim)

        # Compute linear transformation
        self.z = self.W @ x.T + self.b[:, np.newaxis]  # (output_dim,
    batch_size)

        # Apply activation
        self.a = self.activation(self.z)  # (output_dim, batch_size)

        return self.a.T  # (batch_size, output_dim)

    def backward(self, delta):
        """
        Backward pass through the spectral layer.

        Parameters:
        - delta: Gradient of loss with respect to activated output (batch_size,
     output_dim)

        Returns:
        - delta_prev: Gradient of loss with respect to input x (batch_size,
    input_dim)
        """
        batch_size = self.x.shape[0]

        # Compute derivative of activation
        dz = delta.T * self.activation_derivative(self.z)  # (output_dim,
    batch_size)
```

```
        # Compute gradients w.r. to biases
        self.db = np.sum(dz, axis=1) / batch_size  # (output_dim,)

        # Compute gradients w.r. to W
        # W = Q * diag(Lambda) * P.T
        # dL/dW = dz @ x / batch_size
        dW = dz @ self.x / batch_size  # (output_dim, input_dim)

        # Initialize gradients
        self.dQ = np.zeros_like(self.Q)  # (output_dim, d)
        self.dP = np.zeros_like(self.P)  # (input_dim, d)
        self.dLambda = np.zeros_like(self.Lambda)  # (d,)

        # Compute gradients w.r. to Q, P, and Lambda
        for k in range(self.spectral_dim):
            Q_k = self.Q[:, k].reshape(-1, 1)  # (output_dim, 1)
            P_k = self.P[:, k].reshape(-1, 1)  # (input_dim, 1)
            Lambda_k = self.Lambda[k]

            # Gradient w.r. Q_k: (output_dim, 1) += Lambda_k * (dW @ P_k)
            self.dQ[:, k:k+1] += Lambda_k * (dW @ P_k)  # (output_dim, 1)

            # Gradient w.r. P_k: (input_dim, 1) += Lambda_k * (dW.T @ Q_k)
            self.dP[:, k:k+1] += Lambda_k * (dW.T @ Q_k)  # (input_dim, 1)

            # Gradient w.r. Lambda_k: sum of element-wise product
            self.dLambda[k] += np.sum(dW * (Q_k @ P_k.T))  # Scalar

        # Compute gradient w.r. to input x
        # dL/dx = W.T @ dz
        delta_prev = self.W.T @ dz  # (input_dim, batch_size)
        return delta_prev.T  # (batch_size, input_dim)

    def update_parameters(self, learning_rate):
        """
        Updates the spectral parameters Q, Lambda, P, and biases using gradient descent.

        Parameters:
        - learning_rate: Learning rate for gradient descent.
        """
        self.Q -= learning_rate * self.dQ
        self.P -= learning_rate * self.dP
        self.Lambda -= learning_rate * self.dLambda
        self.b -= learning_rate * self.db

# Spectral Neural Network Class
class SpectralNeuralNetwork:
    def __init__(self, layer_dims, spectral_dims, activations):
        """
        Initializes the Spectral Neural Network.

        Parameters:
        - layer_dims: List of neuron counts for each layer, including input and output layers.
        - spectral_dims: List of spectral dimensions for each layer (excluding input layer).
        - activations: List of activation functions for each layer (excluding input layer).
        """
        assert len(layer_dims) - 1 == len(spectral_dims) == len(activations), "Mismatch in layer specifications."
```

```
        self.layers = []
        for i in range(len(layer_dims) - 1):
            layer = SpectralLayer(
                input_dim=layer_dims[i],
                output_dim=layer_dims[i+1],
                spectral_dim=spectral_dims[i],
                activation=activations[i]
            )
            self.layers.append(layer)
        self.loss_history = []  # To store loss values during training

    def forward(self, x):
        """
        Forward pass through the network.

        Parameters:
        - x: Input data of shape (batch_size, input_dim)

        Returns:
        - Output of the network
        """
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def backward(self, loss_grad):
        """
        Backward pass through the network.

        Parameters:
        - loss_grad: Gradient of the loss with respect to the network's output
        """
        for layer in reversed(self.layers):
            loss_grad = layer.backward(loss_grad)

    def update_parameters(self, learning_rate):
        """
        Updates all spectral layers' parameters.

        Parameters:
        - learning_rate: Learning rate for gradient descent.
        """
        for layer in self.layers:
            layer.update_parameters(learning_rate)

    def train(self, X, y, epochs, learning_rate, loss_function='mse'):
        """
        Trains the network using gradient descent.

        Parameters:
        - X: Training data of shape (num_samples, input_dim)
        - y: Training labels of shape (num_samples, output_dim)
        - epochs: Number of training epochs
        - learning_rate: Learning rate for gradient descent
        - loss_function: 'mse' or 'bce'
        """
        for epoch in range(epochs):
            # Forward pass
            output = self.forward(X)  # (num_samples, output_dim)

            # Compute loss
            if loss_function == 'mse':
                loss = mse_loss(y, output)
```

```
                loss_grad = mse_loss_derivative(y, output)  # (num_samples, output_dim)
            elif loss_function == 'bce':
                loss = binary_cross_entropy(y, output)
                loss_grad = binary_cross_entropy_derivative(y, output)  # (num_samples, output_dim)
            else:
                raise ValueError("Unsupported loss function")

            self.loss_history.append(loss)

            # Backward pass
            self.backward(loss_grad)

            # Update parameters
            self.update_parameters(learning_rate)

            # Print loss every 10% of epochs or first epoch
            if (epoch + 1) % (epochs // 10) == 0 or epoch == 0:
                print(f"Epoch {epoch+1}/{epochs}, Loss: {loss:.4f}")

    def predict(self, X):
        """
        Makes predictions with the network.

        Parameters:
        - X: Input data of shape (num_samples, input_dim)

        Returns:
        - Predictions of shape (num_samples, output_dim)
        """
        return self.forward(X)

# Example Usage: XOR, Classification, and Regression Problems
def create_xor_data():
    """
    Creates XOR dataset.

    Returns:
    - X: Input data of shape (4, 2)
    - y: Labels of shape (4, 1)
    """
    X = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
    y = np.array([
        [0],
        [1],
        [1],
        [0]
    ])
    return X, y

def plot_loss(history, title):
    """
    Plots the loss curve.

    Parameters:
    - history: List of loss values.
    - title: Title of the plot.
```

```
    """
    plt.figure(figsize=(8,6))
    plt.plot(history, label='Loss')
    plt.title(title)
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()

def plot_classification_comparison(mcc_scores, models, title):
    """
    Plots a bar chart comparing MCC scores across models.

    Parameters:
    - mcc_scores: List of MCC scores.
    - models: List of model names.
    - title: Title of the plot.
    """
    plt.figure(figsize=(10,6))
    sns.barplot(x=models, y=mcc_scores, palette='viridis')
    plt.title(title)
    plt.ylabel('Matthews Correlation Coefficient (MCC)')
    plt.ylim(-1,1)
    plt.xticks(rotation=45)
    plt.show()

def plot_regression_comparison(r2_scores, models, title):
    """
    Plots a bar chart comparing R2 scores across models.

    Parameters:
    - r2_scores: List of R2 scores.
    - models: List of model names.
    - title: Title of the plot.
    """
    plt.figure(figsize=(10,6))
    sns.barplot(x=models, y=r2_scores, palette='magma')
    plt.title(title)
    plt.ylabel('R2 Score')
    plt.ylim(0,1)
    plt.xticks(rotation=45)
    plt.show()

def plot_confusion_matrix(cm, classes, title):
    """
    Plots the confusion matrix.

    Parameters:
    - cm: Confusion matrix.
    - classes: List of class names.
    - title: Title of the plot.
    """
    plt.figure(figsize=(6,5))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=classes, yticklabels=classes)
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.title(title)
    plt.show()

def plot_roc_curve(y_true, y_scores, title):
    """
```

```
    Plots the ROC curve.

    Parameters:
    - y_true: True binary labels.
    - y_scores: Scores/probabilities from the classifier.
    - title: Title of the plot.
    """
    fpr, tpr, thresholds = roc_curve(y_true, y_scores)
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(8,6))
    plt.plot(fpr, tpr, color='darkorange',
             lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
    plt.plot([0,1], [0,1], color='navy', lw=2, linestyle='--')
    plt.xlim([-0.01,1.0])
    plt.ylim([0.0,1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(title)
    plt.legend(loc="lower right")
    plt.grid(True)
    plt.show()

def plot_regression_predictions(y_true, y_pred, title):
    """
    Plots predicted vs actual values for regression.

    Parameters:
    - y_true: True target values.
    - y_pred: Predicted target values.
    - title: Title of the plot.
    """
    plt.figure(figsize=(8,6))
    plt.scatter(y_true, y_pred, alpha=0.7, label='Spectral NN')
    plt.plot([y_true.min(), y_true.max()], [y_true.min(), y_true.max()], 'r--',
     lw=2, label='Ideal Fit')
    plt.xlabel('Actual Values')
    plt.ylabel('Predicted Values')
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.show()

def main():
    # ============================
    # Part 1: XOR Problem
    # ============================
    print("Training on XOR Problem")
    X_xor, y_xor = create_xor_data()

    # Define network architecture for XOR
    input_dim = 2
    hidden_dim = 4
    output_dim = 1
    spectral_dim_hidden = 7  # Set spectral_dim >= hidden_dim for better
    expressiveness
    spectral_dim_output = 1  # Corrected spectral_dim_output to match
    output_dim

    layer_dims_xor = [input_dim, hidden_dim, output_dim]
    spectral_dims_xor = [spectral_dim_hidden, spectral_dim_output]
    activations_xor = ['relu', 'sigmoid']

```

```
    # Initialize the network
    network_xor = SpectralNeuralNetwork(layer_dims_xor, spectral_dims_xor,
    activations_xor)

    # Train the network on XOR
    epochs = 10000
    learning_rate = 0.1  # Increased learning rate for faster convergence
    network_xor.train(X_xor, y_xor, epochs, learning_rate, loss_function='bce')

    # Plot loss curve for XOR
    plot_loss(network_xor.loss_history, "Spectral Neural Network Loss Curve for
     XOR Problem")

    # Make predictions on XOR
    predictions_xor = network_xor.predict(X_xor)
    predictions_binary_xor = (predictions_xor > 0.5).astype(int)

    print("\nPredictions on XOR after training:")
    for i in range(len(X_xor)):
        print(f"Input: {X_xor[i]}, Predicted: {predictions_binary_xor[i][0]},
    True: {y_xor[i][0]}")

    # ==========================
    # Part 2: Binary Classification on Breast Cancer Dataset
    # ==========================
    print("\n\nTraining on Breast Cancer Classification")

    # Load Breast Cancer dataset
    breast_cancer = load_breast_cancer()
    X_bc = breast_cancer.data
    y_bc = breast_cancer.target.reshape(-1, 1)  # Reshape to (n_samples, 1)

    # Split into train and test
    X_train_bc, X_test_bc, y_train_bc, y_test_bc = train_test_split(
        X_bc, y_bc, test_size=0.92, random_state=42
    )

    # Scale features
    scaler_bc = StandardScaler()
    X_train_bc = scaler_bc.fit_transform(X_train_bc)
    X_test_bc = scaler_bc.transform(X_test_bc)

    # Define network architecture for Breast Cancer
    input_dim_bc = X_train_bc.shape[1]
    hidden_dim_bc = 16
    output_dim_bc = 1
    spectral_dim_hidden_bc = 16  # Adjusted spectral_dim_hidden
    spectral_dim_output_bc = 1  # Corrected spectral_dim_output to match
    output_dim

    layer_dims_bc = [input_dim_bc, hidden_dim_bc, output_dim_bc]
    spectral_dims_bc = [spectral_dim_hidden_bc, spectral_dim_output_bc]
    activations_bc = ['relu', 'sigmoid']

    # Initialize the network
    network_bc = SpectralNeuralNetwork(layer_dims_bc, spectral_dims_bc,
    activations_bc)

    # Train the network on Breast Cancer dataset
    epochs_bc = 500000
    learning_rate_bc = 0.1  # Adjusted learning rate
    network_bc.train(X_train_bc, y_train_bc, epochs_bc, learning_rate_bc,
    loss_function='bce')
```

```
    # Plot loss curve for Breast Cancer
    plot_loss(network_bc.loss_history, "Spectral Neural Network Loss Curve for Breast Cancer Classification")

    # Make predictions on test set
    predictions_bc = network_bc.predict(X_test_bc)
    predictions_binary_bc = (predictions_bc > 0.5).astype(int)

    # Calculate MCC
    mcc_bc = matthews_corrcoef(y_test_bc, predictions_binary_bc)

    # Calculate Accuracy
    accuracy_bc = accuracy_score(y_test_bc, predictions_binary_bc)

    # Confusion Matrix
    cm_bc = confusion_matrix(y_test_bc, predictions_binary_bc)
    plot_confusion_matrix(cm_bc, classes=breast_cancer.target_names, title='Confusion Matrix for Breast Cancer Classification')

    # ROC Curve
    # For ROC, we need probabilities or scores
    # Since our SNN outputs sigmoid activations, we can use them directly
    roc_scores_bc = predictions_bc.ravel()
    plot_roc_curve(y_test_bc, roc_scores_bc, title='ROC Curve for Breast Cancer Classification')

    # ==========================
    # Part 3: Regression on Diabetes Dataset
    # ==========================
    print("\n\nTraining on Diabetes Regression")

    # Load Diabetes dataset
    diabetes = load_diabetes()
    X_diab = diabetes.data
    y_diab = diabetes.target.reshape(-1, 1)  # Reshape to (n_samples, 1)

    # Split into train and test
    X_train_diab, X_test_diab, y_train_diab, y_test_diab = train_test_split(
        X_diab, y_diab, test_size=0.92, random_state=42
    )

    # Scale features
    scaler_diab = StandardScaler()
    X_train_diab = scaler_diab.fit_transform(X_train_diab)
    X_test_diab = scaler_diab.transform(X_test_diab)

    # Define network architecture for Diabetes
    input_dim_diab = X_train_diab.shape[1]
    hidden_dim_diab = 16
    output_dim_diab = 1
    spectral_dim_hidden_diab = 16  # Adjusted spectral_dim_hidden
    spectral_dim_output_diab = 1  # Corrected spectral_dim_output to match output_dim

    layer_dims_diab = [input_dim_diab, hidden_dim_diab, output_dim_diab]
    spectral_dims_diab = [spectral_dim_hidden_diab, spectral_dim_output_diab]
    activations_diab = ['relu', 'relu']  # Using 'relu' for regression output

    # Initialize the network
    network_diab = SpectralNeuralNetwork(layer_dims_diab, spectral_dims_diab, activations_diab)
```

```
    # Train the network on Diabetes dataset
    epochs_diab = 1000000
    learning_rate_diab = 0.005  # Adjusted learning rate for regression
    network_diab.train(X_train_diab, y_train_diab, epochs_diab,
    learning_rate_diab, loss_function='mse')

    # Plot loss curve for Diabetes
    plot_loss(network_diab.loss_history, "Spectral Neural Network Loss Curve
    for Diabetes Regression")

    # Make predictions on test set
    predictions_diab = network_diab.predict(X_test_diab)

    # Calculate R2 Score
    r2_diab = r2_score(y_test_diab, predictions_diab)

    # Scatter Plot for Regression
    plot_regression_predictions(y_test_diab, predictions_diab, "Predicted vs
    Actual Values for Diabetes Regression")

    # ============================
    # Part 4: Comparing with Traditional ML Models
    # ============================

    # ============================
    # Classification Comparison
    # ============================
    print("\n\nComparing Classification Models on Breast Cancer Dataset")

    # Initialize other classification models
    svm_bc = SVC(probability=True, random_state=42)
    mlp_bc = MLPClassifier(hidden_layer_sizes=(16,), max_iter=1000,
    random_state=42)
    logistic_bc = LogisticRegression(max_iter=1000, random_state=42)
    rf_bc = RandomForestClassifier(n_estimators=100, random_state=42)

    # Train models
    svm_bc.fit(X_train_bc, y_train_bc.ravel())
    mlp_bc.fit(X_train_bc, y_train_bc.ravel())
    logistic_bc.fit(X_train_bc, y_train_bc.ravel())
    rf_bc.fit(X_train_bc, y_train_bc.ravel())

    # Make predictions
    predictions_svm_bc = svm_bc.predict(X_test_bc)
    predictions_mlp_bc = mlp_bc.predict(X_test_bc)
    predictions_logistic_bc = logistic_bc.predict(X_test_bc)
    predictions_rf_bc = rf_bc.predict(X_test_bc)

    # Compute MCC for all models
    mcc_svm_bc = matthews_corrcoef(y_test_bc, predictions_svm_bc)
    mcc_mlp_bc = matthews_corrcoef(y_test_bc, predictions_mlp_bc)
    mcc_logistic_bc = matthews_corrcoef(y_test_bc, predictions_logistic_bc)
    mcc_rf_bc = matthews_corrcoef(y_test_bc, predictions_rf_bc)

    # Spectral Neural Network MCC already computed as mcc_bc

    # Prepare data for comparison
    models_classification = ['Spectral NN', 'SVM', 'MLP', 'Logistic Regression'
    , 'Random Forest']
    mcc_scores_classification = [mcc_bc, mcc_svm_bc, mcc_mlp_bc,
    mcc_logistic_bc, mcc_rf_bc]

    # Plot MCC comparison
```

```
    plot_classification_comparison(
        mcc_scores_classification,
        models_classification,
        'MCC Comparison on Breast Cancer Classification'
    )

    # ==========================
    # Regression Comparison
    # ==========================
    print("\n\nComparing Regression Models on Diabetes Dataset")

    # Initialize other regression models
    lr_diab = LinearRegression()
    rf_diab = RandomForestRegressor(n_estimators=100, random_state=42)

    # Train models
    lr_diab.fit(X_train_diab, y_train_diab.ravel())
    rf_diab.fit(X_train_diab, y_train_diab.ravel())

    # Make predictions
    predictions_lr_diab = lr_diab.predict(X_test_diab).reshape(-1,1)
    predictions_rf_diab = rf_diab.predict(X_test_diab).reshape(-1,1)

    # Compute R2 Score for all models
    r2_lr_diab = r2_score(y_test_diab, predictions_lr_diab)
    r2_rf_diab = r2_score(y_test_diab, predictions_rf_diab)
    # Spectral Neural Network R2 already computed as r2_diab

    # Prepare data for comparison
    models_regression = ['Spectral NN', 'Linear Regression', 'Random Forest']
    r2_scores_regression = [r2_diab, r2_lr_diab, r2_rf_diab]

    # Plot R2 comparison
    plot_regression_comparison(
        r2_scores_regression,
        models_regression,
        'R2 Score Comparison on Diabetes Regression'
    )

    # Scatter Plot Comparisons
    plt.figure(figsize=(8,6))
    plt.scatter(y_test_diab, predictions_diab, alpha=0.5, label='Spectral NN')
    plt.scatter(y_test_diab, predictions_lr_diab, alpha=0.5, label='Linear
    Regression')
    plt.scatter(y_test_diab, predictions_rf_diab, alpha=0.5, label='Random
    Forest')
    plt.plot([y_test_diab.min(), y_test_diab.max()], [y_test_diab.min(),
    y_test_diab.max()], 'k--', lw=2, label='Ideal Fit')
    plt.xlabel('Actual Values')
    plt.ylabel('Predicted Values')
    plt.title('Predicted vs Actual Values for Diabetes Regression')
    plt.legend()
    plt.grid(True)
    plt.show()

    # ==========================
    # Summary of Results
    # ==========================
    print("\n\nSummary of Classification Results on Breast Cancer Dataset:")
    for model, mcc in zip(models_classification, mcc_scores_classification):
        print(f"{model}: MCC = {mcc:.4f}")

    print("\nSummary of Regression Results on Diabetes Dataset:")
```

```
    for model, r2 in zip(models_regression, r2_scores_regression):
        print(f"{model}: R2 Score = {r2:.4f}")

if __name__ == "__main__":
    main()
```

Listing 1: Spectral Neural Networks Implementation