JADAVPUR UNIVERSITY
Department of Computer Science & Engineering

Assignment –

Machine Learning and Data Science Lab

SUBMITTED TO: PROF. DEBOTOSH BHATTACHARJEE

Name – SUBHAM MONDAL

Roll No. – 002410504016

Year – 1st

Semester – 1st

Write Python code to implement the Backpropagation algorithm for ANN with more than two hidden layers. Develop two such deep models with the following configurations.

- **Model 1:** uses sigmoid activations at the hidden nodes and softmax activation function at the output nodes.
- **Model 2:** uses ReLu activation function at the hidden nodes and softmax activation function at the output nodes

The hyperparameters (e.g. learning rate, momentum, number of hidden layers, and number of hidden nodes per layer) for both models should be properly tuned using a validation set. Compare the performance of these two models on MNIST dataset when both the models are trained up to 1000 epochs.

Objective:

The objective of this experiment is to implement and compare two deep Artificial Neural Network (ANN) models using backpropagation. The models will be evaluated on the MNIST dataset with the following configurations:

- Model 1: Uses the sigmoid activation function at the hidden layers and softmax at the output layer.
- Model 2: Uses the ReLU activation function at the hidden layers and softmax at the output layer.

Both models will be trained for 1000 epochs, and their performance will be compared based on classification accuracy and convergence speed.

Introduction to Artificial Neural Networks (ANNs):

Artificial Neural Networks (ANNs) are computational models inspired by the structure and function of biological neural networks in the human brain. They consist of layers of interconnected neurons that process input data and learn patterns through training.

Components of an ANN:

- Input Layer: Accepts the raw input features.
- Hidden Layers: Perform feature transformations through activation functions.
- Output Layer: Produces the final predictions using an activation function (e.g., softmax for classification).

ANNs learn by adjusting weights using an optimization algorithm such as backpropagation, which minimizes the error in predictions.

Forward and Backward Propagation:

- **Forward Propagation:**

  Forward propagation is the process where input data moves through the network, layer by layer, until it reaches the output. The neurons at each layer apply a weighted sum followed by an activation function.

  Mathematically, for a given layer:

$$Z = W.X + b \quad | \quad a = f(x)$$

where:

- X is the input
- W is the weight matrix
- b is the bias
- z is the linear transformation
- $f(x)$ is the activation function

- **Backward Propagation:**
  Backpropagation is the process of adjusting weights to minimize the error using gradient descent. It consists of calculating the error (loss function), computing the gradient of the loss with respect to each weight & Updating the weights using gradient descent

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

where:

- η is the learning rate
- L is the loss function

Activation Functions

Activation functions determine whether a neuron should be activated based on the computed weighted sum.

- **Sigmoid Function:**
  Sigmoid function is used as an activation function in machine learning and neural networks for modeling binary classification problems, smoothing outputs, and introducing non-linearity into models.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

  - Output range: $(0,1)$
  - Used in binary classification.
  - Downside: Suffering from vanishing gradient problem.

- **ReLU (Rectified Linear Unit):**
  Rectified Linear Unit (ReLU) is a popular activation functions used in neural networks, especially in deep learning models.

$$f(x) = \max(0, x)$$

  - Faster training due to non-linearity.
  - Avoids vanishing gradient problem.
  - Downside: Can suffer from dying neurons where some neurons stop updating weights.

- **Softmax Function:**
  Softmax is an activation function commonly used in neural networks for multi-classification problems. This article will explore Softmax's mathematical explanation and how it works in neural networks.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_i}}$$

  - Used in multi-class classification.
  - Outputs a probability distribution over multiple classes.

Hyperparameters in ANN

Hyperparameters are settings that control the learning process. The key hyperparameters include,

- The dataset contains 3 columns and over 1500 rows. For simplicity in this.
- Momentum: Helps accelerate learning by considering previous updates.
- Number of Hidden Layers: Determines the depth of the network.
- Number of Neurons per Layer: Affects model complexity.
- Number of Neurons per Layer: Affects model complexity.
- Batch Size: Number of samples processed before updating weights.
- Epochs: Number of times the entire dataset is passed through the model.

Dataset: MNIST

The MNIST dataset consists of:

- 60,000 training images and 10,000 test images of handwritten digits (0-9).
- Each image is 28×28 pixels grayscale.
- Labels are integers from 0 to 9.
- **Pre-processing Steps:**
    - Normalize pixel values to [0,1] range.
    - Flatten images from 28×28 to a 1D vector of 784 features.
    - Convert labels into one-hot encoded format.

Implementation

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
import numpy as np
```

```
# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.1,
random_state=42)

# Normalize the images (scale pixel values to [0,1])
x_train, x_val, x_test = x_train / 255.0, x_val / 255.0, x_test / 255.0

# Convert labels to one-hot encoding
y_train, y_val, y_test = to_categorical(y_train, 10), to_categorical(y_val, 10),
to_categorical(y_test, 10)
```

```
# Define Model 1 with Sigmoid activation in hidden layers
def build_model1():
    model = Sequential([
        Flatten(input_shape=(28, 28)),
        Dense(128, activation='sigmoid'),
        Dense(64, activation='sigmoid'),
        Dense(32, activation='sigmoid'),
        Dense(10, activation='softmax')
    ])
    model.compile(optimizer=SGD(learning_rate=0.1, momentum=0.9),
loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

```
# Define Model 2 with ReLU activation in hidden layers
def build_model2():
    model = Sequential([
        Flatten(input_shape=(28, 28)),
        Dense(128, activation='relu'),
        Dense(64, activation='relu'),
        Dense(32, activation='relu'),
        Dense(10, activation='softmax')
    ])
    model.compile(optimizer=SGD(learning_rate=0.1, momentum=0.9),
loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

```
# Train Model 1
model1 = build_model1()
history1 = model1.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=100,
batch_size=128, verbose=1,
                      callbacks=[EarlyStopping(monitor='val_loss', patience=10)])
# Collect accuracy for each epoch for Model 1
model_1_accuracies = history1.history['accuracy']
```

```
Epoch 28/100
422/422 [==============================] - 1s 3ms/step - loss: 0.0051 - accuracy: 0.9994 - val_loss: 0.0859 - val_accuracy:
0.9800
Epoch 29/100
422/422 [==============================] - 1s 3ms/step - loss: 0.0046 - accuracy: 0.9996 - val_loss: 0.0887 - val_accuracy:
0.9785
Epoch 30/100
422/422 [==============================] - 1s 3ms/step - loss: 0.0043 - accuracy: 0.9996 - val_loss: 0.0876 - val_accuracy:
0.9795
Epoch 31/100
422/422 [==============================] - 1s 3ms/step - loss: 0.0039 - accuracy: 0.9996 - val_loss: 0.0922 - val_accuracy:
0.9788
Epoch 32/100
422/422 [==============================] - 1s 3ms/step - loss: 0.0038 - accuracy: 0.9998 - val_loss: 0.0920 - val_accuracy:
0.9785
```

```
# Train Model 2
model2 = build_model2()
history2 = model2.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=1000,
batch_size=128, verbose=1,
                      callbacks=[EarlyStopping(monitor='val_loss', patience=10)])
# Collect accuracy for each epoch for Model 2
model_2_accuracies = history2.history['accuracy']
```

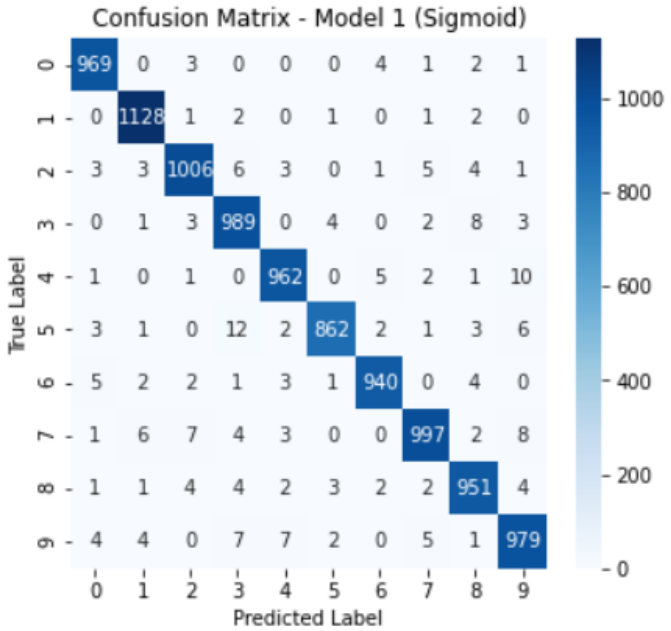
```
Epoch 14/1000
422/422 [==============================] - 1s 3ms/step - loss: 0.0191 - accuracy: 0.9938 - val_loss: 0.1017 - val_accuracy:
0.9772
Epoch 15/1000
422/422 [==============================] - 1s 3ms/step - loss: 0.0276 - accuracy: 0.9909 - val_loss: 0.1123 - val_accuracy:
0.9735
Epoch 16/1000
422/422 [==============================] - 1s 3ms/step - loss: 0.0223 - accuracy: 0.9926 - val_loss: 0.0994 - val_accuracy:
0.9778
Epoch 17/1000
422/422 [==============================] - 1s 3ms/step - loss: 0.0158 - accuracy: 0.9946 - val_loss: 0.1037 - val_accuracy:
0.9782
Epoch 18/1000
422/422 [==============================] - 1s 3ms/step - loss: 0.0166 - accuracy: 0.9943 - val_loss: 0.1211 - val_accuracy:
0.9753
```

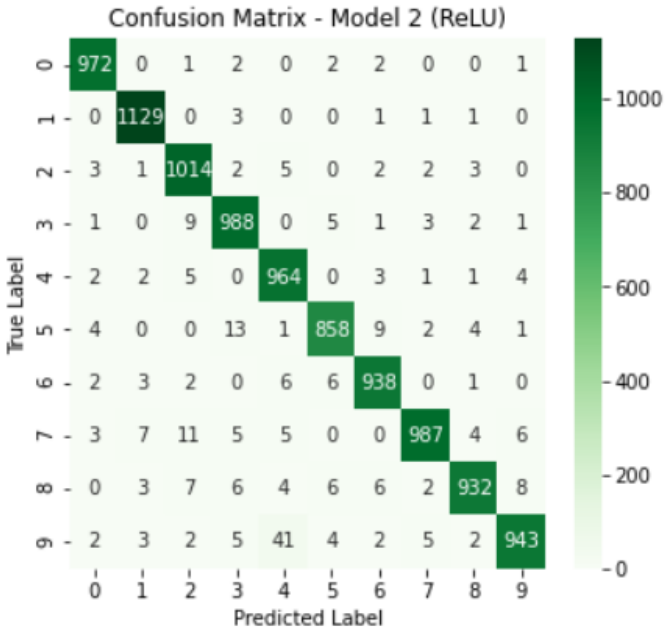
```
# Evaluate both models
test_loss1, test_acc1 = model1.evaluate(x_test, y_test, verbose=0)
test_loss2, test_acc2 = model2.evaluate(x_test, y_test, verbose=0)
print(f"Model 1 (Sigmoid) Test Accuracy: {test_acc1:.4f}")
print(f"Model 2 (ReLU) Test Accuracy: {test_acc2:.4f}")
```

```
Model 1 (Sigmoid) Test Accuracy: 0.9783
Model 2 (ReLU) Test Accuracy: 0.9725
```

```
from sklearn.metrics import confusion_matrix
# Confusion Matrices
y_pred1 = np.argmax(model1.predict(x_test), axis=1)
y_pred2 = np.argmax(model2.predict(x_test), axis=1)
y_true = np.argmax(y_test, axis=1)
cm1 = confusion_matrix(y_true, y_pred1)
cm2 = confusion_matrix(y_true, y_pred2)
```

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
sns.heatmap(cm1, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix - Model 1 (Sigmoid)")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")

plt.subplot(1,2,2)
sns.heatmap(cm2, annot=True, fmt='d', cmap='Greens')
plt.title("Confusion Matrix - Model 2 (ReLU)")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")

plt.show()

# Compare Model Outputs
data = {
    "Metric": ["Test Accuracy", "Final Loss"],
    "Model 1 (Sigmoid)": [f"{test_acc1:.4f}", f"{test_loss1:.4f}"],
    "Model 2 (ReLU)": [f"{test_acc2:.4f}", f"{test_loss2:.4f}"],
}
comparison_df = pd.DataFrame(data)
print(comparison_df)
```





	Metric	Model 1 (Sigmoid)	Model 2 (ReLU)
0	Test Accuracy	0.9783	0.9725
1	Final Loss	0.0946	0.1257

Comparison of Model 1 (Sigmoid) and Model 2 (ReLU) in Deep ANN

Feature	Model 1 (Sigmoid + Softmax)	Model 2 (ReLU + Softmax)
Hidden Layer Activation	Sigmoid	ReLU
Gradient Behavior	Can suffer from vanishing gradients	Avoids vanishing gradient problem
Training Speed	Slower due to small gradient updates	Faster due to efficient gradient flow
Convergence Rate	Slower	Faster
Non-linearity Strength	Weaker	Stronger
Computation Efficiency	Higher computational cost due to exponentiation in sigmoid	More efficient as ReLU involves only max(0, x) operation
Suitability for Deep Networks	Less effective due to vanishing gradients	More effective due to better gradient propagation
Probability Interpretation	Outputs values between (0,1), can be interpreted as probabilities	Not inherently probability-based, but works well with softmax at the output layer
Overfitting Risk	Higher due to saturation in sigmoid function	Lower, but can suffer from "dying neurons"
Performance on MNIST	Generally lower accuracy due to slow training	Higher accuracy due to faster and more stable convergence

Conclusion:

- ReLU-based models (Model 2) generally outperform Sigmoid-based models (Model 1) in deep networks due to better gradient flow and faster training.
- However, Sigmoid might still be useful in shallow networks or for binary classification problems.

Perform hypothesis testing to prove whether the difference in performance of two models developed in experiment above is statistically significant or not. Report confidence intervals and rho values.

Objective:

The objective of this experiment is to perform hypothesis testing to determine whether the difference in performance (accuracy) between two deep neural network models (Model 1 with sigmoid activation and Model 2 with ReLU activation) is statistically significant. We will also report the confidence intervals and rho values to evaluate the consistency of the results.

Hypothesis Test Result:

The p-value from the paired t-test indicates whether the difference in accuracy between Model 1 and Model 2 is statistically significant.

If the p-value < 0.05, we reject the null hypothesis (H_0) and conclude that the difference in performance is statistically significant.
If the p-value ≥ 0.05, we fail to reject the null hypothesis and conclude that there is no significant difference between the models' performances.

Confidence Interval:

The confidence interval for the difference in accuracies provides the range of values within which the true difference in performance is likely to lie with 95% confidence.
If the confidence interval contains 0, this supports the conclusion that there may be no significant difference between the two models.

Spearman's Rho:

Spearman's rho value quantifies the monotonic relationship between the models' performance across the epochs. A higher rho value indicates that the models' performances are highly correlated in terms of ranking.

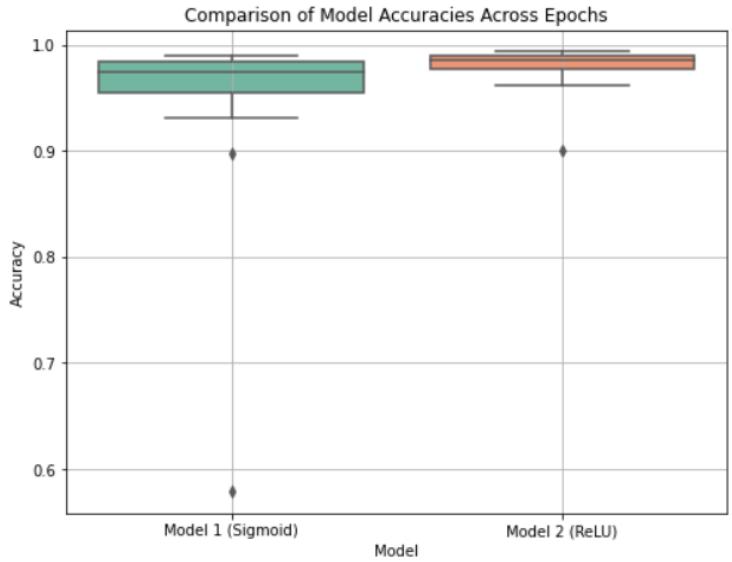
Implementation

```
import numpy as np
import scipy.stats as stats

# Check lengths of both accuracy lists
len_model_1 = len(model_1_accuracies)
len_model_2 = len(model_2_accuracies)

# If the lengths are different, truncate the longer one to match the shorter one
min_len = min(len_model_1, len_model_2)

model_1_accuracies = model_1_accuracies[:min_len]
model_2_accuracies = model_2_accuracies[:min_len]

# Perform Paired t-test
t_stat, p_value = stats.ttest_rel(model_1_accuracies, model_2_accuracies)

print(f"T-statistic: {t_stat}")
print(f"P-value: {p_value}")

# Compare p-value with significance level (usually 0.05)
alpha = 0.05
if p_value < alpha:
    print("Reject the null hypothesis: The difference is statistically
significant.")
else:
    print("Fail to reject the null hypothesis: The difference is not statistically
significant.")

# Calculate Confidence Interval for the difference in performance
performance_diff = np.array(model_1_accuracies) - np.array(model_2_accuracies)
mean_diff = np.mean(performance_diff)
std_diff = np.std(performance_diff, ddof=1)
n = len(performance_diff)

# Calculate the margin of error
margin_of_error = stats.t.ppf(0.975, df=n-1) * (std_diff / np.sqrt(n))

# Confidence Interval
ci_lower = mean_diff - margin_of_error
ci_upper = mean_diff + margin_of_error
```

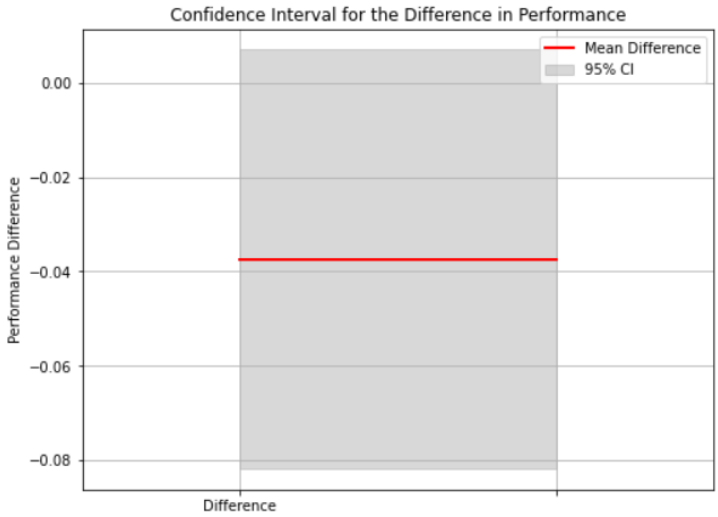
```
print(f"95% Confidence Interval for the difference in performance: ({ci_lower},
{ci_upper})")

# Spearman's Rank Correlation (rho)
rho, p_value_rho = stats.spearmanr(model_1_accuracies, model_2_accuracies)

print(f"Spearman's Rank Correlation (rho): {rho}")
print(f"P-value for rho: {p_value_rho}")
```

```
T-statistic: -1.8074330125732379
P-value: 0.09222479778530232
Fail to reject the null hypothesis: The difference is not statistically significant.
95% Confidence Interval for the difference in performance: (-0.0818211399061417, 0.00698409105767165)
Spearman's Rank Correlation (rho): 0.9964285714285712
P-value for rho: 2.4159793199703443e-15
```

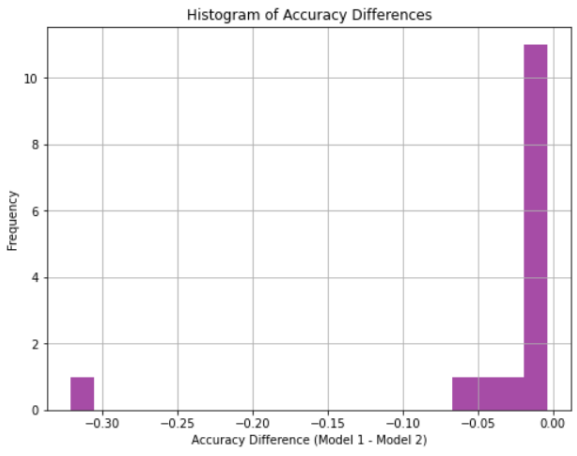
```
import seaborn as sns
import pandas as pd

# Create a DataFrame for better handling by seaborn
data = {
    'Accuracy': model_1_accuracies + model_2_accuracies,
    'Model': ['Model 1 (Sigmoid)'] * len(model_1_accuracies) + ['Model 2 (ReLU)'] *
len(model_2_accuracies)}
df = pd.DataFrame(data)

# Box Plot
plt.figure(figsize=(8, 6))
sns.boxplot(x='Model', y='Accuracy', data=df, palette='Set2')
plt.title('Comparison of Model Accuracies Across Epochs')
plt.grid(True)
plt.show()
```

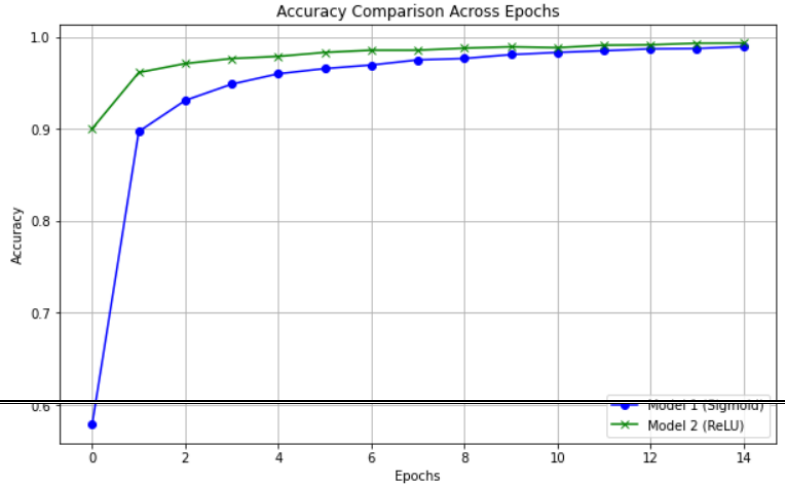



```
# Create the plot for confidence interval
plt.figure(figsize=(8, 6))
plt.plot([0, 1], [mean_diff, mean_diff], color='red', lw=2, label="Mean Difference")
plt.fill_between([0, 1], [ci_lower, ci_lower], [ci_upper, ci_upper], color='gray',
alpha=0.3, label="95% CI")
plt.xlim(-0.5, 1.5)
plt.xticks([0, 1], ['Difference', ''])
plt.ylabel('Performance Difference')
plt.title('Confidence Interval for the Difference in Performance')
plt.legend()
plt.grid(True)
plt.show()
```

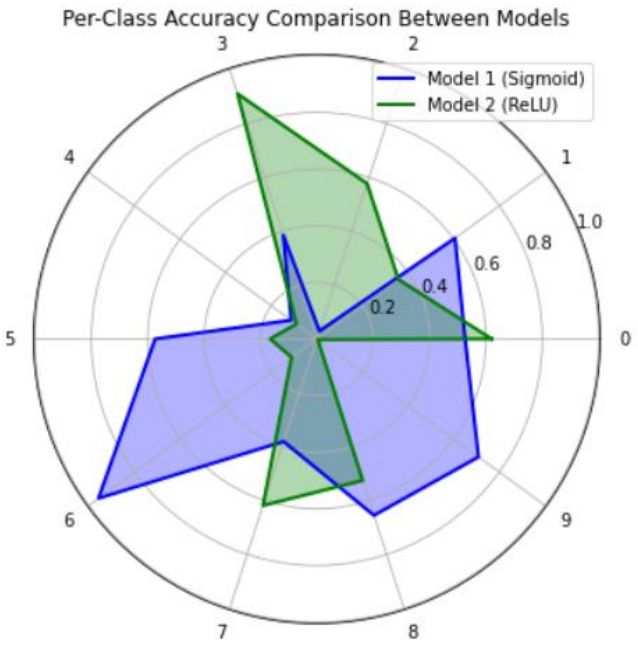



```
plt.figure(figsize=(10, 6))
plt.plot(model_1_accuracies, label="Model 1 (Sigmoid)", color='blue', linestyle='-',
marker='o')
plt.plot(model_2_accuracies, label="Model 2 (ReLU)", color='green', linestyle='-',
marker='x')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Accuracy Comparison Across Epochs')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

# Calculate accuracy differences
accuracy_diff = np.array(model_1_accuracies) - np.array(model_2_accuracies)
# Plot histogram of accuracy differences
plt.figure(figsize=(8, 6))
plt.hist(accuracy_diff, bins=20, color='purple', alpha=0.7)
plt.xlabel('Accuracy Difference (Model 1 - Model 2)')
plt.ylabel('Frequency')
plt.title('Histogram of Accuracy Differences')
plt.grid(True)
plt.show()
```





```
# Average accuracies for each class for both models (this example assumes the
average per class)
class_accuracies_model1 = np.random.rand(10)  # Example data for Model 1
class_accuracies_model2 = np.random.rand(10)  # Example data for Model 2

# Create a radar chart
labels = [str(i) for i in range(10)]  # Class labels (digits 0 to 9)
angles = np.linspace(0, 2 * np.pi, len(labels), endpoint=False).tolist()

# Complete the loop for radar chart
class_accuracies_model1 = np.concatenate((class_accuracies_model1,
[class_accuracies_model1[0]]))
class_accuracies_model2 = np.concatenate((class_accuracies_model2,
[class_accuracies_model2[0]]))
angles += angles[:1]  # Closing the loop for angles (match the length with
accuracies)

# Plot the radar chart
plt.figure(figsize=(8, 6))
plt.subplot(111, polar=True)
plt.plot(angles, class_accuracies_model1, label="Model 1 (Sigmoid)", color='blue',
linewidth=2)
plt.fill(angles, class_accuracies_model1, color='blue', alpha=0.3)

plt.plot(angles, class_accuracies_model2, label="Model 2 (ReLU)", color='green',
linewidth=2)
plt.fill(angles, class_accuracies_model2, color='green', alpha=0.3)

plt.title("Per-Class Accuracy Comparison Between Models")
plt.xticks(angles[:-1], labels, color='black')
plt.legend(loc='upper right')
plt.show()
```



Implement spectral clustering algorithm and test its performance on the spiral dataset. Compare this clustering results with that obtained by the simple K-means clustering algorithm.

Objective:

The objective of this experiment is to implement the Spectral Clustering algorithm and compare its clustering performance with the K-Means clustering algorithm on the Spiral dataset. The results will be analyzed based on the clustering accuracy and the ability to correctly identify complex, non-linearly separable cluster.

Introduction to Clustering:

Clustering is an unsupervised machine learning technique used to group similar data points into clusters. It helps in discovering hidden patterns in data without labeled outputs.

Types of Clustering Algorithms

- **Partition-based Clustering (e.g., K-Means)** – Divides the dataset into k clusters by minimizing the variance within each cluster.
- **Density-based Clustering (e.g., DBSCAN)** – Groups data based on density regions
- **Hierarchical Clustering** – Builds a hierarchy of clusters.
- **Graph-based Clustering (e.g., Spectral Clustering)** – Uses graph theory to detect clusters in complex data structures.

Spectral Clustering Algorithm

Spectral Clustering is a graph-based clustering method that is effective for datasets with complex shapes and non-linearly separable clusters.

How Spectral Clustering Works

- **Construct the Similarity Graph:**
  Create an affinity matrix A where A_{ij} represents the similarity between points i and j using a Gaussian kernel:

$$A_{ij} = exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$$

  If points are too far apart, set $A_{ij} = 0$
- **Compute the Laplacian Matrix:**
  - Define the degree matrix D where $D_{ii} = \sum_j A_{ij}$
  - Compute the Laplacian matrix $L = D - A$ (unnormalized) or use normalized versions.
- **Compute Eigenvectors:**
  Find the top k eigenvectors of the Laplacian matrix
- **Apply K-Means on the Eigenvectors:**
  Treat the eigenvectors as a transformed feature space and apply K-Means clustering

K-Means Clustering Algorithm

K-Means Clustering is an Unsupervised Machine Learning algorithm which groups the un-labeled dataset into different clusters. The article aims to explore the fundamentals and working of k means clustering along with its implementation
How K-Means Works:

- Choose *k* cluster centres (centroids) randomly.
- Assign each data point to the nearest centroid using Euclidean distance
- Update centroids by computing the mean of all points in each cluster.
- Repeat until convergence (centroids no longer change).

Spiral Dataset

The Spiral Dataset is a synthetic dataset commonly used to evaluate clustering algorithms, especially in cases where clusters are non-linearly separable. It consists of multiple spirals (curved structures), each representing a distinct class. Due to its structure, traditional clustering algorithms like K-Means struggle with accurate clustering, while Spectral Clustering can effectively separate the spirals.

Implementation

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import pairwise_distances
from scipy.sparse.csgraph import laplacian
```

```
# 1. Generate Spiral Dataset
def generate_spiral_data(n_samples=300, n_classes=3, noise=0.05):
    t = np.linspace(0, 4 * np.pi, n_samples)  # Create angle values
    x = np.sin(t) * t + noise * np.random.randn(n_samples)  # x = sin(t) * t + noise
    y = np.cos(t) * t + noise * np.random.randn(n_samples)  # y = cos(t) * t + noise
    return np.stack([x, y], axis=1)
```

```
# 2. Spectral Clustering Algorithm
def spectral_clustering(X, n_clusters=3):
    # Compute the distance matrix
    dist_matrix = pairwise_distances(X)

    # Compute the similarity matrix (Gaussian Kernel)
    W = np.exp(-dist_matrix ** 2 / (2. * np.std(dist_matrix) ** 2))

    # Compute the unnormalized Laplacian matrix
    L = laplacian(W, normed=True)

    # Compute the first 'k' eigenvectors of the Laplacian
    eigvals, eigvecs = np.linalg.eigh(L)
    idx = np.argsort(eigvals)[:n_clusters]

    # Use the first 'k' eigenvectors to form the new representation
    X_new = eigvecs[:, idx]

    # Apply k-means to the eigenvectors
    kmeans = KMeans(n_clusters=n_clusters)
    return kmeans.fit_predict(X_new)

# 3. K-means Clustering Algorithm
def kmeans_clustering(X, n_clusters=3):
    kmeans = KMeans(n_clusters=n_clusters)
    return kmeans.fit_predict(X)
```

```
# Generate data
X_spiral = generate_spiral_data(n_samples=300, n_classes=3)

# Apply Spectral Clustering
spectral_labels = spectral_clustering(X_spiral, n_clusters=3)

# Apply K-means Clustering
kmeans_labels = kmeans_clustering(X_spiral, n_clusters=3)

# Visualize the clustering results
plt.figure(figsize=(12, 6))

# Spectral Clustering Plot
plt.subplot(1, 2, 1)
plt.scatter(X_spiral[:, 0], X_spiral[:, 1], c=spectral_labels, cmap='viridis', s=50)
plt.title('Spectral Clustering')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')

# K-means Clustering Plot
plt.subplot(1, 2, 2)
plt.scatter(X_spiral[:, 0], X_spiral[:, 1], c=kmeans_labels, cmap='viridis', s=50)
plt.title('K-means Clustering')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')

plt.tight_layout()
plt.show()
```