EXPERIMENT-17

Aim: Implementing Dimensionality Reduction using Principal Component Analysis (PCA).

Theory:

Dimensionality reduction techniques like PCA help reduce the number of input variables in a dataset while retaining most of the important information.

PCA achieves this by finding a set of new uncorrelated variables (principal components) that successively maximize variance.

It is widely used in preprocessing for machine learning, visualization of high-dimensional data, and noise reduction.

Code:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

df = pd.read_csv('/content/tmnist.csv')  # Replace with appropriate data file

print("Labels: ", df['labels'].unique())
no_of_classes = df['labels'].nunique()


X = df.drop(columns=['names', 'labels'], axis=1)  # Drop unwanted columns
y = df['labels']

X_std = StandardScaler().fit_transform(X)

pca = PCA(n_components=2)  # Reduce to 2 dimensions
principal_components = pca.fit_transform(X_std)
principal_df = pd.DataFrame(data=principal_components, columns=['Principal Component 1',
'Principal Component 2'])
```

```
# Concatenate with target labels for visualization
final_df = pd.concat([principal_df, y.reset_index(drop=True)], axis=1)

# Visualize the data in the 2D PCA space
plt.figure(figsize=(10, 8))
targets = df['labels'].unique()
colors = ['r', 'g', 'b', 'c', 'm', 'y', 'k'] * (len(targets) // 7 + 1)  # Adjust number of colors

for target, color in zip(targets, colors):
   indices_to_keep = final_df['labels'] == target
   plt.scatter(final_df.loc[indices_to_keep, 'Principal Component 1'],
               final_df.loc[indices_to_keep, 'Principal Component 2'], c=color, s=50)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('2 Component PCA')
plt.legend(targets, loc='best')
plt.show()

explained_variance = pca.explained_variance_ratio_
print("Explained Variance Ratio:", explained_variance)
print("Total Variance Explained by 2 Components:", sum(explained_variance))
```

Learning Outcomes:

1. PCA effectively reduces the dimensionality of the dataset to 2 components.
2. Visualization in the 2D space helps understand class separability and data clustering.
3. The explained variance shows the amount of information retained by the selected principal components.

EXPERIMENT-18

Aim: Build an Artificial Neural Network (ANN) with backpropagation.

Theory:

An **ANN** is a model inspired by the human brain, consisting of input, hidden, and output layers. During training, data passes through these layers (forward pass), and the error is calculated using a loss function.

Backpropagation then adjusts the weights and biases by calculating gradients and updating them via optimization (e.g., gradient descent). This process is repeated over multiple epochs to minimize error.

Hyperparameters: Training an ANN involves tuning several hyperparameters:

- **Learning Rate:** Determines the step size for weight updates. A higher learning rate may lead to faster convergence, but it could also overshoot the optimal solution. A lower rate might make training slower but can yield a more accurate solution.
- **Batch Size:** Controls the number of samples passed through the network before updating the weights.
- **Epochs:** The number of complete passes through the entire training dataset.

Code:

```
import keras
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical

# Load dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Scale input values to [0, 1]
x_train = x_train.reshape(60000, 784).astype('float32') / 255
x_test = x_test.reshape(10000, 784).astype('float32') / 255

# Convert target values to one-hot encoding
y_train = to_categorical(y_train, num_classes=10)
```

```
y_test = to_categorical(y_test, num_classes=10)

# Build the model
model = Sequential([
   Dense(10, activation='sigmoid', input_shape=(784,)),  # Hidden layer
   Dense(10, activation='softmax')  # Output layer
])

# Compile the model
model.compile(loss="categorical_crossentropy",
          optimizer="sgd",
          metrics=['accuracy'])

# Train the model
history = model.fit(x_train, y_train, batch_size=100, epochs=20)  # here 20 epochs

# Evaluate the model on test data
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'Test accuracy: {round(test_acc, 4)}')

# Display a sample prediction
example_index = 11
prediction = model.predict(x_test[example_index].reshape(1, 784))
print("Predicted label:", np.argmax(prediction))
print("True label:", np.argmax(y_test[example_index]))
```

Learning Outcomes:

1. The ANN trained with backpropagation achieves a certain accuracy on the MNIST test data.
2. The confusion matrix visualizes the performance of the model across different classes.
3. Predictions for a sample input provide insight into model's confidence levels for each class.

Result:

Accuracy: 87.37% for 20 epochs

More epoch, better the model is trained on dataset and higher is the accuracy.

Output:

```
Epoch 1/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.2034 - loss: 2.2523
Epoch 2/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.5131 - loss: 1.9400
Epoch 3/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.6092 - loss: 1.6985
Epoch 4/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.6771 - loss: 1.4936
Epoch 5/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7190 - loss: 1.3370
Epoch 6/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7593 - loss: 1.1986
Epoch 7/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7840 - loss: 1.0891
Epoch 8/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.7999 - loss: 1.0014
Epoch 9/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8143 - loss: 0.9251
Epoch 10/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8254 - loss: 0.8607
Epoch 11/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8333 - loss: 0.8067
Epoch 12/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8395 - loss: 0.7644
Epoch 13/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8462 - loss: 0.7245
Epoch 14/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8529 - loss: 0.6836
Epoch 15/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8555 - loss: 0.6553
Epoch 16/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8595 - loss: 0.6269
Epoch 17/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.8609 - loss: 0.6108
Epoch 18/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8642 - loss: 0.5883
Epoch 19/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8639 - loss: 0.5759
Epoch 20/20
600/600 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8700 - loss: 0.5536
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8557 - loss: 0.5895
Test accuracy: 0.8737
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
Predicted label: 6
True label: 6
```

EXPERIMENT-19

Aim: Image Classification on Fashion MNIST Dataset using Artificial Neural Network (ANN)

Theory:

Fashion MNIST is a dataset of grayscale images of 10 different categories of clothing items (like shirts, shoes, and coats). The dataset includes **60,000** training images and 10,000 test images, each of **28x28 pixels.**

Each image is labeled with one of the **10 classes**. Fashion MNIST is often used as a drop-in replacement for MNIST to test image classification models in a slightly more complex setting.

Artificial Neural Networks (ANN) are a powerful model type for image classification. They consist of layers of nodes (neurons) where each neuron takes input data, applies a transformation, and passes the output to the next layer.

Using backpropagation, ANN adjusts weights through training to improve accuracy. In image classification, ANN learns patterns across pixel values to classify images accurately.

Code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

# Load the Fashion MNIST dataset
(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()

# Data Preprocessing
X_train = X_train / 255.0  # Scale pixel values to [0, 1]
X_test = X_test / 255.0
y_train = to_categorical(y_train, 10)  # One-hot encode labels
y_test = to_categorical(y_test, 10)

# Building the Model
model = Sequential([
```

```
    Flatten(input_shape=(28, 28)),  # Flatten 28x28 images to 784-element vectors
    Dense(128, activation='relu'),  # Hidden layer with 128 neurons
    Dense(10, activation='softmax')  # Output layer for 10 classes
])

# Compile the Model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the Model
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)

# Evaluate the Model on Test Data
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f'Test Accuracy: {test_accuracy:.4f}')

# Plot Training and Validation Accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy Over Epochs')
plt.show()

# Predict a sample from test data
sample_index = 0
sample_image = X_test[sample_index].reshape(1, 28, 28)  # Reshape for prediction
sample_label = np.argmax(model.predict(sample_image), axis=1)
print("Predicted label for sample image:", sample_label[0])
```

Learning Outcomes:

1. **Data Normalization**: Scaling image pixel values between 0 and 1 improves ANN performance.
2. **Simple Architecture Works**: Even a basic ANN can effectively classify Fashion MNIST images.
3. **Validation Helps**: Monitoring validation accuracy during training detects overfitting early.

EXPERIMENT-20

Aim: Build an Artificial Neural Network (ANN) on MNIST Dataset.

Theory:

The MNIST dataset contains 60,000 training and 10,000 test images of handwritten digits (0-9), each of size 28x28 pixels.

An Artificial Neural Network (ANN) is a deep learning model inspired by the human brain, consisting of layers of interconnected neurons.

Key components of ANN : input layer, hidden layers, output layer, activation functions, optimizer, loss functions

In the case of image classification, an ANN learns to identify patterns in the pixel values of images to classify them into different categories.

Code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Data Preprocessing
X_train = X_train / 255.0  # Normalize pixel values to [0, 1]
X_test = X_test / 255.0
y_train = to_categorical(y_train, 10)  # One-hot encode labels
y_test = to_categorical(y_test, 10)

# Build the model
model = Sequential([
    Flatten(input_shape=(28, 28)),  # Flatten the 28x28 images into 784 values
    Dense(128, activation='relu'),  # Hidden layer with 128 neurons
    Dense(10, activation='softmax')  # Output layer for 10 classes
```

```
])

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)

# Evaluate the model on the test dataset
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test Accuracy: {test_acc:.4f}')

# Plot the training and validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy Over Epochs')
plt.show()
```

Output:

```
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.8644 - loss: 0.4820 - val_accuracy: 0.9552 - val_loss: 0.1576
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 16s 5ms/step - accuracy: 0.9601 - loss: 0.1354 - val_accuracy: 0.9628 - val_loss: 0.1204
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 3ms/step - accuracy: 0.9750 - loss: 0.0859 - val_accuracy: 0.9709 - val_loss: 0.0991
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9810 - loss: 0.0635 - val_accuracy: 0.9709 - val_loss: 0.0984
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.9852 - loss: 0.0503 - val_accuracy: 0.9770 - val_loss: 0.0845
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9892 - loss: 0.0365 - val_accuracy: 0.9736 - val_loss: 0.0883
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9917 - loss: 0.0288 - val_accuracy: 0.9737 - val_loss: 0.0881
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9933 - loss: 0.0228 - val_accuracy: 0.9769 - val_loss: 0.0826
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9953 - loss: 0.0172 - val_accuracy: 0.9736 - val_loss: 0.0919
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9953 - loss: 0.0156 - val_accuracy: 0.9758 - val_loss: 0.0924
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9728 - loss: 0.1014
Test Accuracy: 0.9763
```

Accuracy: 97.63 % for 10 epochs

EXPERIMENT-13

Aim: To implement Data classification using K-Means.

Theory:

K-Means is an unsupervised machine learning algorithm commonly used for clustering tasks. It groups data points into clusters based on their similarity, aiming to minimize the sum of squared distances between data points and their respective cluster centroids.

Key Concepts:

1. **Objective:** The main objective of K-Means is to partition data into clusters so that points within the same cluster are more similar to each other than to points in other clusters.
2. **K Centroids:** The algorithm requires specifying the number of clusters (K) in advance. It initializes K centroids randomly, then assigns each data point to the nearest centroid.
3. **Iterations:** K-Means operates iteratively, calculating the distance of each data point from the centroids, reassigning points to the nearest centroid, and recalculating centroids until convergence.
4. **Advantages:** Simple, fast, and efficient for clustering large datasets. Effectively discovers clusters with spherical shapes.
5. **Disadvantages:** Sensitive to the initial placement of centroids. May not perform well on data with non-spherical clusters.

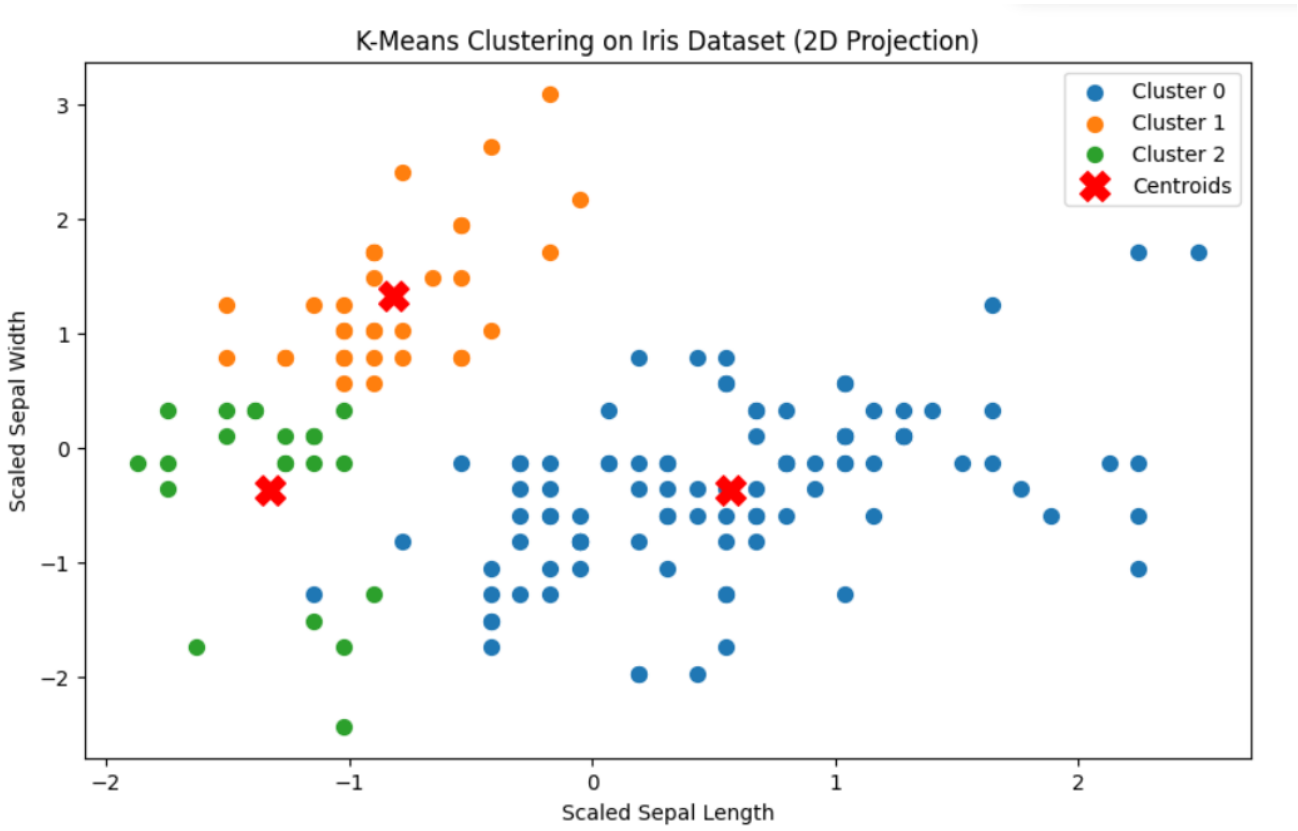
Code:

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.datasets import load_iris

from sklearn.cluster import KMeans

from sklearn.preprocessing import StandardScaler


# Load the Iris dataset

iris = load_iris()

X = iris.data  # Features (sepal length, sepal width, petal length, petal width)
```

```
y = iris.target  # Actual species (used for comparison later)

# Feature scaling for better K-Means convergence
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# clustering with 3 clusters (since we know there are 3 species)
kmeans = KMeans(n_clusters=3, random_state=42)
y_pred = kmeans.fit_predict(X_scaled)

# Visualize the clusters (using the first two features for simplicity)
plt.figure(figsize=(10, 6))

# Plot each cluster with a unique color and label
for cluster_label in np.unique(y_pred):
    plt.scatter(
        X_scaled[y_pred == cluster_label, 0],
        X_scaled[y_pred == cluster_label, 1],
        label=f"Cluster {cluster_label}",
        s=50
    )

plt.scatter(
    kmeans.cluster_centers_[:, 0],
    kmeans.cluster_centers_[:, 1],
    c='red', marker='X', s=200, label="Centroids"
)
```

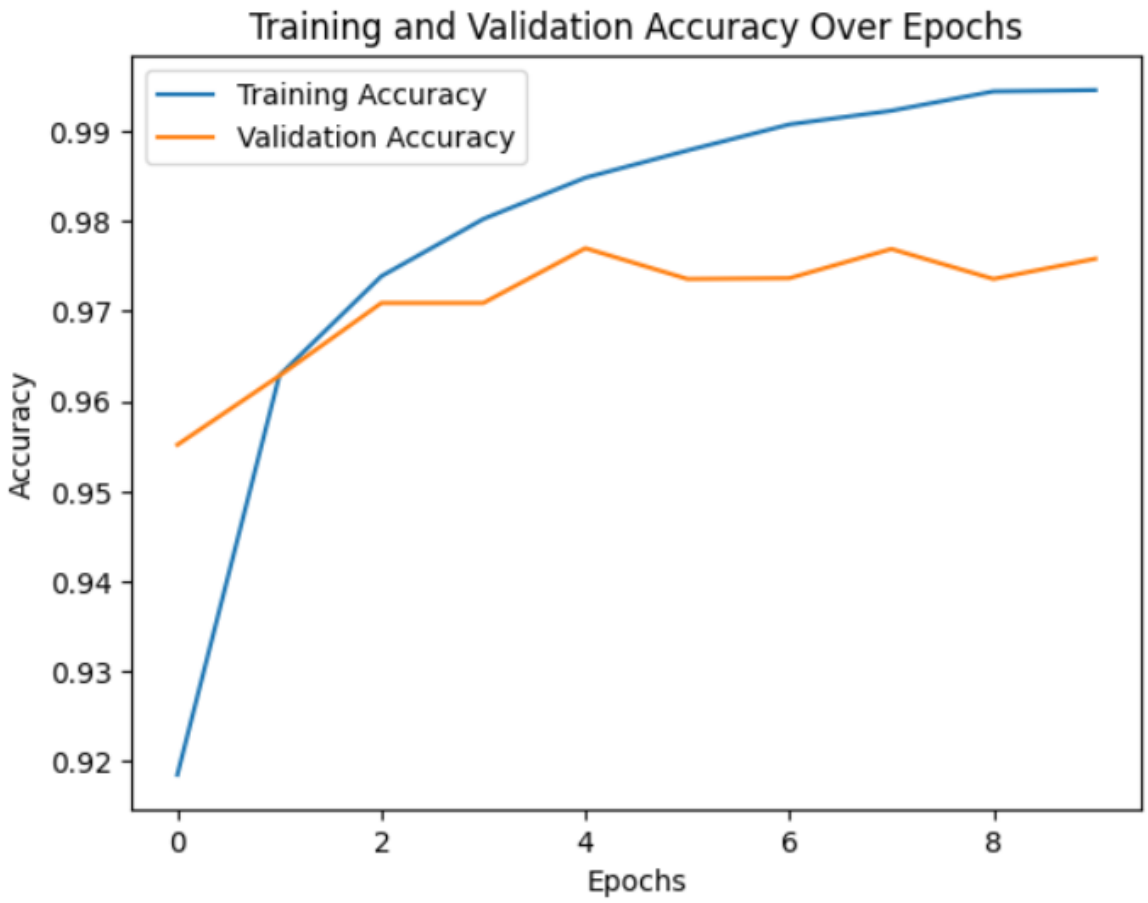
```
plt.xlabel('Scaled Sepal Length')
plt.ylabel('Scaled Sepal Width')
plt.title('K-Means Clustering on Iris Dataset (2D Projection)')
plt.legend()
plt.show()
```

Output:



Learning Outcomes:

1. Gained an understanding of K-Means clustering, including its iterative process to achieve cluster convergence.
2. Recognized the simplicity and efficiency of K-Means for clustering tasks, especially in large datasets.
3. Identified the limitations of K-Means, such as sensitivity to centroid initialization and challenges with non-spherical clusters.



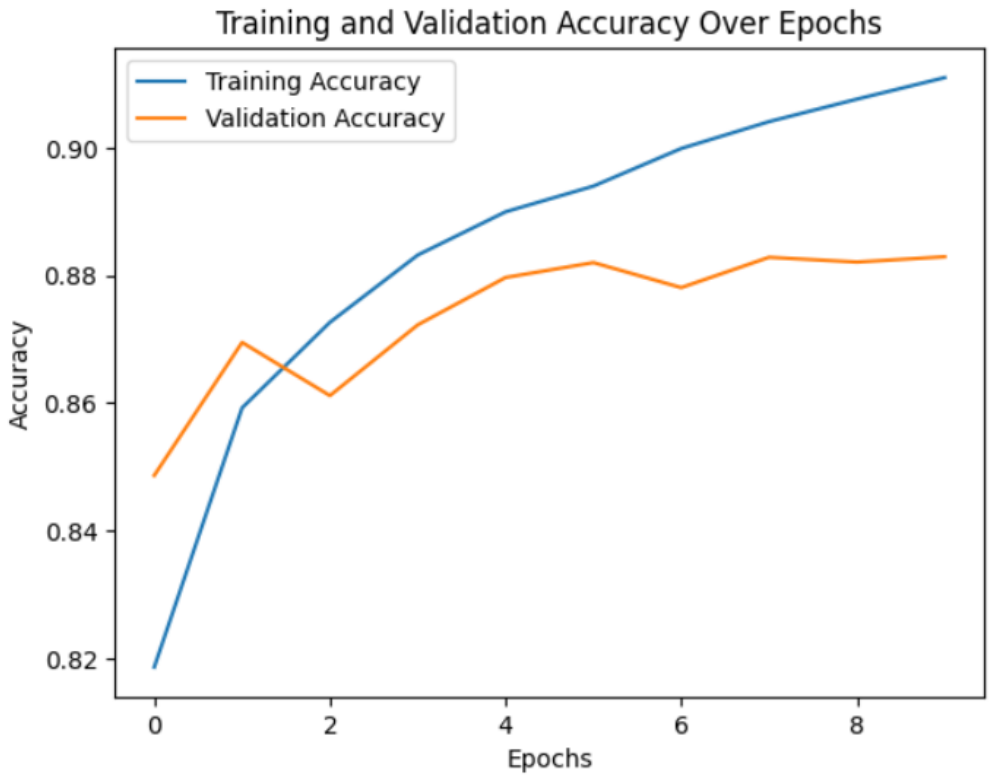
Learning Outcomes:

1. **Data Preprocessing**: Normalizing image pixel values improves the ANN's ability to learn effectively.
2. **Activation Functions**: Using functions like ReLU and softmax helps the model learn complex patterns and output probabilities for classification.
3. **Model Evaluation**: Tracking accuracy and loss during training helps ensure the model is learning correctly and prevents overfitting.

Output:

```
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 22s 12ms/step - accuracy: 0.7714 - loss: 0.6513 - val_accuracy: 0.8487 - val_loss: 0.4335
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.8537 - loss: 0.3975 - val_accuracy: 0.8695 - val_loss: 0.3678
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 4ms/step - accuracy: 0.8695 - loss: 0.3566 - val_accuracy: 0.8612 - val_loss: 0.3871
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.8809 - loss: 0.3271 - val_accuracy: 0.8723 - val_loss: 0.3499
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8907 - loss: 0.2972 - val_accuracy: 0.8797 - val_loss: 0.3393
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8939 - loss: 0.2855 - val_accuracy: 0.8820 - val_loss: 0.3274
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9006 - loss: 0.2710 - val_accuracy: 0.8781 - val_loss: 0.3447
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9049 - loss: 0.2557 - val_accuracy: 0.8828 - val_loss: 0.3245
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.9093 - loss: 0.2442 - val_accuracy: 0.8821 - val_loss: 0.3295
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9116 - loss: 0.2386 - val_accuracy: 0.8829 - val_loss: 0.3324
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8740 - loss: 0.3558
Test Accuracy: 0.8736
```

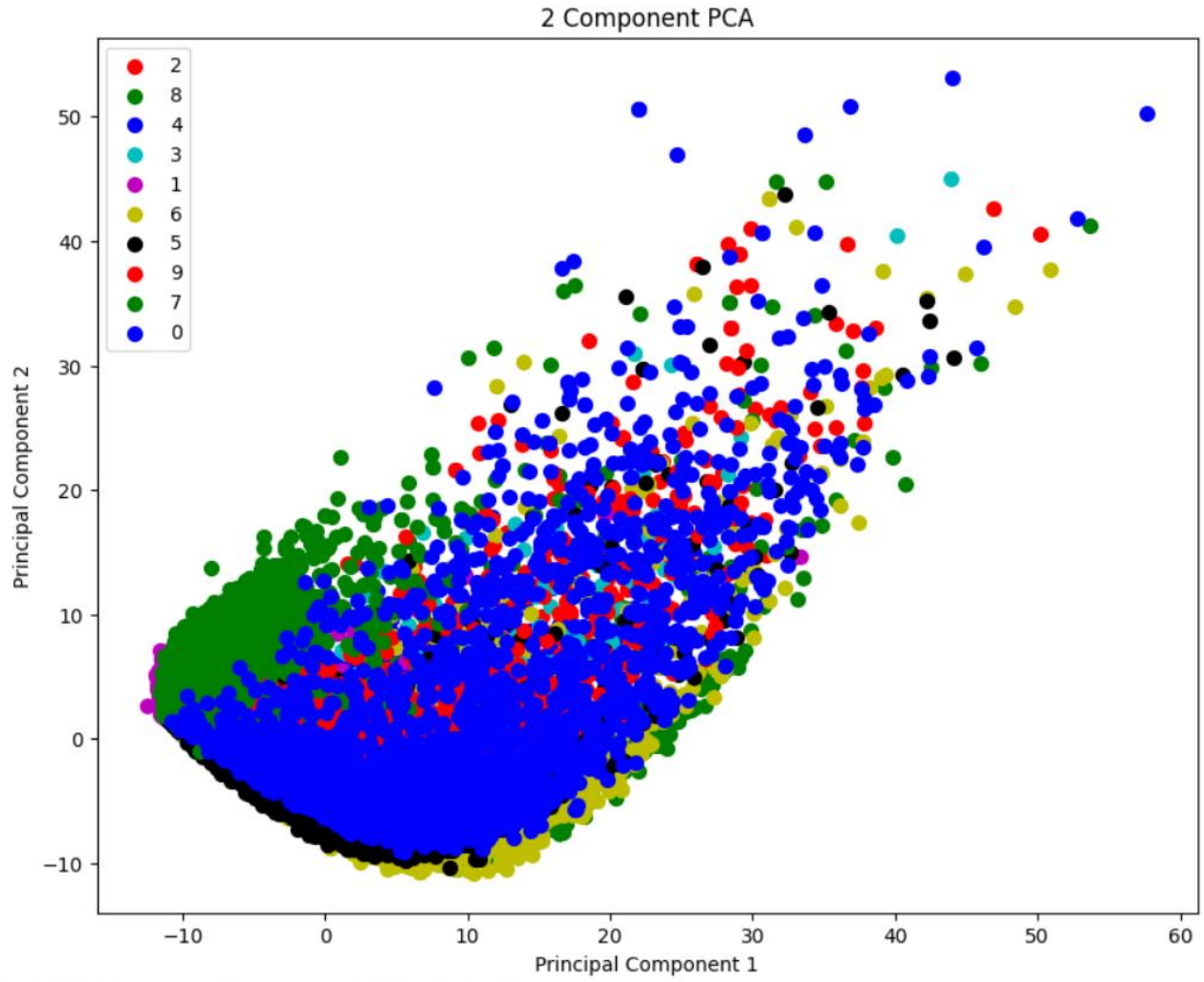
Accuracy: 87.36% for 10 epochs




```
WARNING:tensorflow:5 out of the last 190 calls to <function TensorFlowTrain
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 63ms/step
Predicted label for sample image: 9
```

Output:

```
Labels:  [2 8 4 3 1 6 5 9 7 0]
```




```
Explained Variance Ratio: [0.10312951 0.05227091]
Total Variance Explained by 2 Components: 0.155400428492806
```

DELHI TECHNOLOGICAL UNIVERSITY

DEPARTMENT OF
COMPUTER SCIENCE ENGINEERING

CO327
Machine Learning

SUBMITTED BY:

Ayush Tandon
2K22/CO/133

DELHI TECHNOLOGICAL UNIVERSITY

DEPARTMENT OF
COMPUTER SCIENCE ENGINEERING

CO301
Software Engineering

SUBMITTED TO:

Lavindra Gautam

SUBMITTED BY:

Ayush Tandon
2K22/CO/133