Ex.No: 01 **IMPLEMENTING PERCEPTRON** **Date:**

Aim:

To implement the Perceptron Learning Algorithm for an AND gate using Python (Jupyter Notebook), and plot the decision boundary after training.

Algorithm

1. **Start**
2. Initialize:
   - ➢ Set weights and bias to 0.
   - ➢ Choose a learning rate α (e.g., 0.1).
3. Define input data X and target output T for the AND gate.
4. For a fixed number of epochs or until convergence:
   - ➢ For each training sample:
     - Compute the weighted sum:

       z = w1*x1 + w2*x2 + bias
     - Apply the activation function:

       output = 1 if z ≥ 0 else 0
     - If output ≠ target:
       - ✓ Update weights:

         w = w + α * (target-output) * x
       - ✓ Update bias:

         bias = bias + α * (target-output)
5. Repeat until all outputs match targets or max epochs reached.
6. Plot the decision boundary:
   - ➢ Line equation:

     w1*x + w2*y + b = 0
7. Display final weights, bias, and predictions.
8. **Stop**

Program:

```
import numpy as np

import matplotlib.pyplot as plt

# Input data (AND gate)

X = np.array([
```

```
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
# Target output
T = np.array([0, 0, 0, 1])
# Initialize Weights and Bias
weights = np.array([0.3,-0.1])  # 2 inputs
bias = 0.2
alpha = 0.1
epochs = 10
# Training Loop
for epoch in range(epochs):
    print(f"\nEpoch {epoch+1}")
    error_occurred = False
    for i in range(len(X)):
        x_i = X[i]
        t = T[i]
        z = np.dot(weights, x_i) + bias
        y = 1 if z >= 0 else 0
# Update only if Prediction is Wrong
        if y != t:
            weights += alpha * (t-y) * x_i
            bias += alpha * (t-y)
            error_occurred = True
        print(f"x: {x_i}, target: {t}, output: {y}, weights: {weights}, bias: {bias}")
    if not error_occurred:
        print("\nTraining converged. Stopping early.")
        break

print(f"Final weights: {weights}, Final bias: {bias}")
```

 # Predict on Trainig Data

```
predictions=[]
for x_i in X:
    z = np.dot(weights, x_i) + bias
    y = 1 if z >= 0 else 0
    #predictions.append(y)
    print(f"x: {x_i},Predictions:{y}")
```

 # Plotting Decision Boundary

```
plt.scatter(X[:,0],X[:,1],c=T,cmap='bwr',edgecolors='k')
x_1=np.min(X[:,0])
x_2=np.max(X[:,0])
x_3=-(weights[0] * x_1 + bias)/weights[1]
x_4=-(weights[0] * x_2 + bias)/weights[1]

plt.plot([x_1,x_2],[x_3,x_4],'g-')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Perceptron Decision Boundary')
plt.show()
```

Outputs:

Epoch 1

x: [0 0], target: 0, output: 1, weights: [0.3 -0.1], bias: 0.1

x: [0 1], target: 0, output: 1, weights: [0.3 -0.2], bias: 0.0

x: [1 0], target: 0, output: 1, weights: [0.2 -0.2], bias: -0.1

x: [1 1], target: 1, output: 0, weights: [0.3 -0.1], bias: 0.0

Epoch 2

x: [0 0], target: 0, output: 1, weights: [0.3 -0.1], bias: -0.1

x: [0 1], target: 0, output: 0, weights: [0.3 -0.1], bias: -0.1

x: [1 0], target: 0, output: 1, weights: [0.2 -0.1], bias: -0.2

x: [1 1], target: 1, output: 0, weights: [0.3 0.], bias: -0.1

Epoch 3

x: [0 0], target: 0, output: 0, weights: [0.3 0.], bias: -0.1

x: [0 1], target: 0, output: 0, weights: [0.3 0.], bias: -0.1

x: [1 0], target: 0, output: 1, weights: [0.2 0.], bias: -0.2

x: [1 1], target: 1, output: 0, weights: [0.3 0.1], bias: -0.1

Epoch 4

x: [0 0], target: 0, output: 0, weights: [0.3 0.1], bias: -0.1

x: [0 1], target: 0, output: 1, weights: [0.3 0.], bias: -0.2

x: [1 0], target: 0, output: 1, weights: [0.2 0.], bias: -0.30000000000000004

x: [1 1], target: 1, output: 0, weights: [0.3 0.1], bias: -0.20000000000000004

Epoch 5

x: [0 0], target: 0, output: 0, weights: [0.3 0.1], bias: -0.20000000000000004

x: [0 1], target: 0, output: 0, weights: [0.3 0.1], bias: -0.20000000000000004

x: [1 0], target: 0, output: 1, weights: [0.2 0.1], bias: -0.30000000000000004

x: [1 1], target: 1, output: 0, weights: [0.3 0.2], bias: -0.20000000000000004

Epoch 6

x: [0 0], target: 0, output: 0, weights: [0.3 0.2], bias: -0.20000000000000004

x: [0 1], target: 0, output: 0, weights: [0.3 0.2], bias: -0.20000000000000004

x: [1 0], target: 0, output: 1, weights: [0.2 0.2], bias: -0.30000000000000004

x: [1 1], target: 1, output: 1, weights: [0.2 0.2], bias: -0.30000000000000004

Epoch 7

x: [0 0], target: 0, output: 0, weights: [0.2 0.2], bias: -0.30000000000000004

x: [0 1], target: 0, output: 0, weights: [0.2 0.2], bias: -0.30000000000000004

x: [1 0], target: 0, output: 0, weights: [0.2 0.2], bias: -0.30000000000000004

x: [1 1], target: 1, output: 1, weights: [0.2 0.2], bias: -0.30000000000000004

Training converged. Stopping early.

Final weights: [0.2 0.2], Final bias: -0.30000000000000004

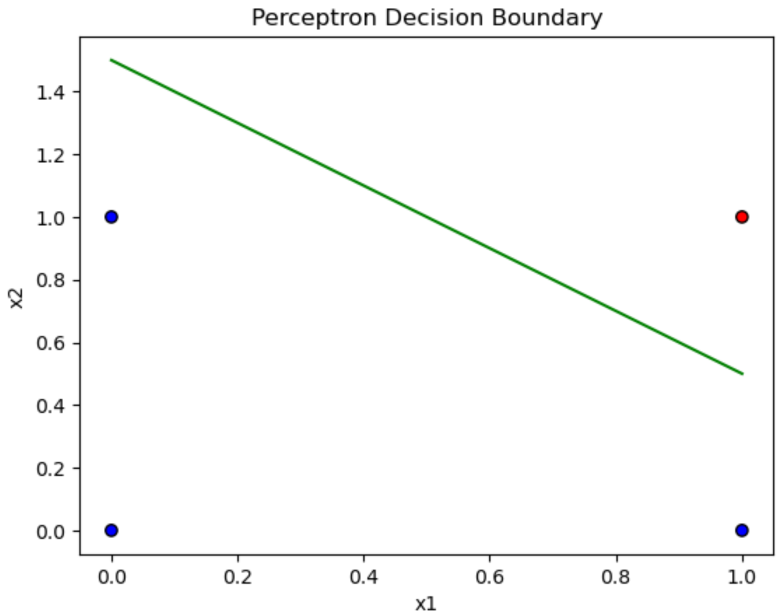
Test the trained perceptron:

x: [0 0],Predictions:0

x: [0 1],Predictions:0

x: [1 0],Predictions:0

x: [1 1],Predictions:1



Results:

The perceptron was successfully trained to perform the AND logical function, accurately predicting all outputs and correctly separating the classes with a decision boundary, demonstrating convergence within a few epochs for this linearly separable problem.

Ex.No: 02 **SOLVING XOR PROBLEM USING DNN** **Date:**

Aim:

To implement and train a **Deep Neural Network (DNN)** using Keras to solve the XOR logical function, with manually set initial weights, biases, and learning rate, and to stop training automatically once the model achieves correct prediction on all input patterns.

Algorithm:

1. **Import Libraries:**
   - Import NumPy, Matplotlib, and TensorFlow Keras modules.
2. **Prepare Dataset:**
   - Define input patterns and target outputs for the XOR function.
3. **Model Construction:**
   - Create a Sequential model.
   - Add a hidden layer with 2 neurons and sigmoid activation.
   - Add an output layer with 1 neuron and sigmoid activation.
4. **Manual Initialization:**
   - Set the initial weights and biases manually for both hidden and output layers using **set_weights()**.
5. **Model Compilation:**
   - Compile the model using Adam optimizer with a manually defined learning rate.
   - Use binary cross-entropy as the loss function and accuracy as the metric.
6. **Prediction Before Training:**
   - Predict and display the outputs for all input patterns using the untrained model.
7. **Define Epoch Callback:**
   - Create a custom Keras callback to print weights, biases, and predictions after each epoch.
   - Implement early stopping if all predictions are correct.
8. **Train the Model:**
   - Fit the model on the dataset for a maximum number of epochs, using the callback for monitoring.
9. **Final Evaluation:**

- After training, display final predictions.
- Plot training accuracy and loss curves.

Program:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
import tensorflow.keras.backend as K
# 1. XOR inputs and outputs
X = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([[0], [1], [1], [0]])
# 2. Build the model
model = Sequential()
model.add(Dense(2, input_dim=2, activation='sigmoid', name='hidden'))  # Hidden layer
model.add(Dense(1, activation='sigmoid', name='output'))           # Output layer
# 3. Compile model with Adam optimizer
optimizer = Adam(learning_rate=1)
model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])
# 4. Manual Initialization Weights and Bias
initial_hidden_weights = np.array([[ 2.524, -5.875],[-2.715 , 5.026]])
initial_hidden_biases = np.array([-1.429 ,-3.095])
initial_output_weights = np.array([[1.531],[3.24 ]])
initial_output_biases = np.array([-1.099])

model.layers[0].set_weights([initial_hidden_weights, initial_hidden_biases])
model.layers[1].set_weights([initial_output_weights, initial_output_biases])
# 5. View initial weights and biases
print("Initial Weights and Biases (Before Training):")
for layer in model.layers:
    weights, biases = layer.get_weights()
```

```
    print(f"\nLayer: {layer.name}")
    print("Weights:\n", weights)
    print("Biases:\n", biases)
# 6. Initial predictions (before training)
print("\nInitial Predictions (Before Training):")
initial_preds = model.predict(X)
for i, p in enumerate(initial_preds):
    print(f"Input: {X[i]} → Output: {p[0]:.4f}")
# 7. Create callback to capture weights and predictions after each epoch
class EpochLogger(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        print(f"\nEpoch {epoch+1}")
        preds = self.model.predict(X, verbose=0)
        all_correct = True  # Flag to check if all predictions are correct
        for i, p in enumerate(preds):
            predicted_class = int(p[0] > 0.5)
            print(f"  Input: {X[i]} → Output: {p[0]:.4f} → Class: {predicted_class}")
            if predicted_class != y[i][0]:
                all_correct = False
        # Optionally: print weights
        for layer in self.model.layers:
            weights, biases = layer.get_weights()
            print(f"  Layer: {layer.name}")
            print(f"    Weights: {np.round(weights, 3)}")
            print(f"    Biases : {np.round(biases, 3)}")
        # Stop if all predictions are correct
        if all_correct:
            print("All predictions correct. Stopping early.")
            self.model.stop_training = True
# 8. Train the model with history + epoch callback
history = model.fit(X, y, epochs=100, verbose=0, validation_data=(X, y), callbacks=[EpochLogger()])
```

```
# 9. Final predictions after training
print("\nFinal XOR Predictions (After Training):")
final_preds = model.predict(X)
for i, p in enumerate(final_preds):
    print(f"Input: {X[i]} → Output: {p[0]:.4f} → Class: {int(p[0] > 0.5)}")
# 10. Plot Accuracy and Loss
plt.figure(figsize=(12, 5))
# Accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy', linestyle='dashed')
plt.title("Accuracy Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.grid(True)
plt.legend()
# Loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss', linestyle='dashed')
plt.title("Loss Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Outputs:

Initial Weights and Biases (Before Training):

Layer: hidden

Weights: [[2.524 -5.875] [-2.715 5.026]]

Biases: [-1.429 -3.095]

Layer: output

Weights: [[1.531] [3.24]]

Biases: [-1.099]

Initial Predictions (Before Training):

1/1 [==============================] - 0s 48ms/step

Input: [0 0] → Output: 0.3401

Input: [0 1] → Output: 0.8525

Input: [1 0] → Output: 0.5122

Input: [1 1] → Output: 0.3134

Epoch 1

Input: [0 0] → Output: 0.1391 → Class: 0

Input: [0 1] → Output: 0.8335 → Class: 1

Input: [1 0] → Output: 0.4501 → Class: 0

Input: [1 1] → Output: 0.1306 → Class: 0

Layer: hidden

Weights: [[3.524 -6.874] [-3.715 6.026]]

Biases : [-2.425 -4.094]

Layer: output

Weights: [[2.531] [4.24]]

Biases : [-2.098]

Epoch 2

Input: [0 0] → Output: 0.3366 → Class: 0

Input: [0 1] → Output: 0.9761 → Class: 1

Input: [1 0] → Output: 0.8725 → Class: 1

Input: [1 1] → Output: 0.3056 → Class: 0

Layer: hidden

Weights: [[4.432 -7.68] [-4.58   6.981]]

Biases : [-1.693 -3.479]

Layer: output

Weights: [[3.519][5.238]]

Biases : [-1.382]

All predictions correct. Stopping early.

Final XOR Predictions (After Training):

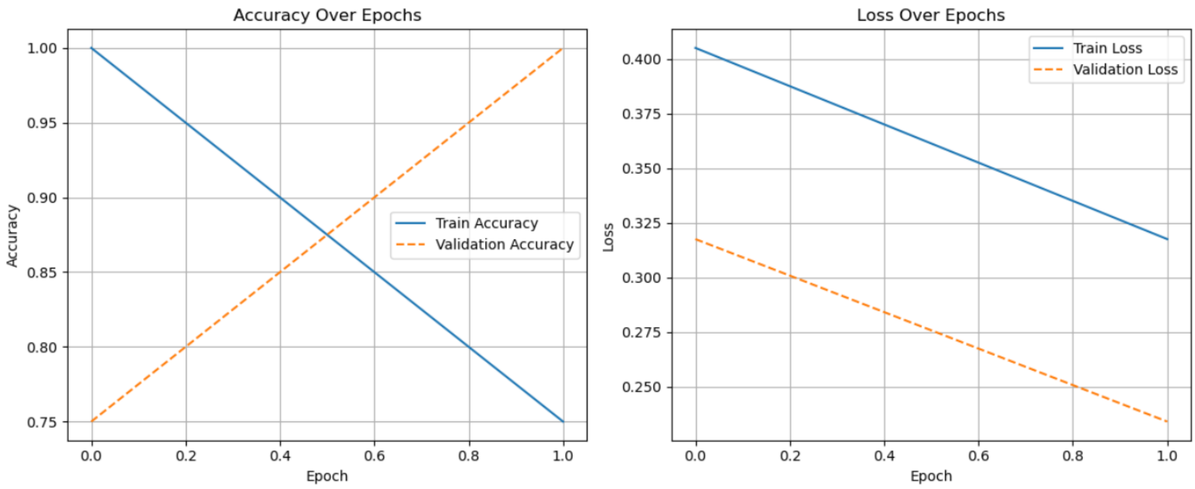
1/1 [==============================] - 0s 19ms/step

Input: [0 0] → Output: 0.3366 → Class: 0

Input: [0 1] → Output: 0.9761 → Class: 1

Input: [1 0] → Output: 0.8725 → Class: 1

Input: [1 1] → Output: 0.3056 → Class: 0



Result:

The DNN was successfully implemented and trained to solve the XOR logical function. The network architecture used one hidden layer with two neurons and sigmoid activation. Initial weights and biases were manually set. The model was trained with early stopping enabled to terminate once it achieved perfect classification.

Ex.No: 03 DIGIT CLASSIFICATION USING NEURAL NETWORK (MNIST DATASET) Date:

Aim:

To develop and train a feedforward neural network using TensorFlow/Keras to classify handwritten digits from the **MNIST (Modified National Institute of Standards and Technology)** dataset, and to evaluate the model's performance using accuracy and loss metrics.

Algorithm:

1. **Import Required Libraries**
   - Import necessary modules: NumPy, Matplotlib, and TensorFlow/Keras libraries for model creation, training, and evaluation.
2. **Load and Explore Dataset**
   - Load the MNIST dataset using **tf.keras.datasets.mnist.load_data().**
   - Display sample images and label distribution in the training set.
3. **Preprocess the Dataset**
   - Normalize pixel values to the range [0, 1] by dividing by 255.
   - Convert class labels to one-hot encoding using to_categorical().
4. **Build the Neural Network Model**
   - Use the **Sequential()** model.
   - Add a **Flatten layer** to convert 28×28 images into 784-element vectors.
   - Add a **Dense hidden laye**r with 128 units and ReLU activation.
   - Add an **output Dense layer** with 10 units (digits 0–9) and softmax activation.
5. **Compile the Model**
   - Use the adam optimizer.
   - Use categorical_crossentropy as the loss function.
   - Use accuracy as the evaluation metric.
6. **Train the Model**
   - Train the model using .fit() with:
     - epochs=10
     - batch_size=64
     - validation_split=0.2 to monitor generalization.
7. **Evaluate the Model**
   - Evaluate on test data using .evaluate() to obtain final accuracy and loss.
8. **Visualize Training Performance**

- Plot the training and validation accuracy and loss over epochs.

9. **Make Predictions**
   - Predict a sample image from the test set and compare the predicted label with the actual label.

Program:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical
# 1. Load MNIST dataset
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
# 2. Label distribution in training set
unique, counts = np.unique(y_train, return_counts=True)
print("Label distribution in training set:")
for digit, count in zip(unique, counts):
   print(f"Digit {digit}: {count} samples")
# 3. Visualize Sample Training Images
plt.figure(figsize=(10, 10))
for i in range(50):
   plt.subplot(5, 10, i + 1)
   plt.imshow(X_train[i], cmap='gray')
   plt.axis('off')
   plt.title(str(y_train[i]), fontsize=8)
plt.suptitle("50 Sample Digits from Training Set", fontsize=16)
plt.tight_layout()
plt.show()
# 4. Normalize pixel values to [0,1]
X_train = X_train / 255.0
X_test = X_test / 255.0
# 5. Convert labels to one-hot encoding
```

```
y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)
```

 # 6. Build the neural network model

```
model = Sequential([
    Flatten(input_shape=(28, 28)),       # Input layer
    Dense(128, activation='relu'),       # Hidden layer
    Dense(10, activation='softmax')      # Output layer
])
```

 # 7. Compile the model

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

 # 8. Train the model and store training history

```
history = model.fit(X_train, y_train_cat,
                    epochs=10,
                    batch_size=64,
                    validation_split=0.2,
                    verbose=1)
```

 # 9. Evaluate on test set

```
test_loss, test_acc = model.evaluate(X_test, y_test_cat)
print(f"\nTest Accuracy: {test_acc:.4f}, Test Loss: {test_loss:.4f}")
```

 # 10. Plot Training Accuracy and Validation Loss

```
plt.figure(figsize=(12, 5))
```

 # Plot accuracy

```
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy', color='blue')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy', linestyle='--', color='green')
plt.title("Accuracy Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(True)
```

 # Plot loss

```
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss', color='red')
plt.plot(history.history['val_loss'], label='Validation Loss', linestyle='--', color='orange')
plt.title("Loss Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

 # 11. Predict and display one sample image from test set

 # Predict first 10 test samples

```
predictions = model.predict(X_test[:10])
predicted_classes = np.argmax(predictions, axis=1)
plt.figure(figsize=(10, 4))
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(X_test[i], cmap='gray')
    plt.title(f"Predicted: {np.argmax(predictions[i])}, True: {y_test[i]}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Output:

Label distribution in training set:

Digit 0: 5923 samples

Digit 1: 6742 samples

Digit 2: 5958 samples

Digit 3: 6131 samples

Digit 4: 5842 samples

Digit 5: 5421 samples

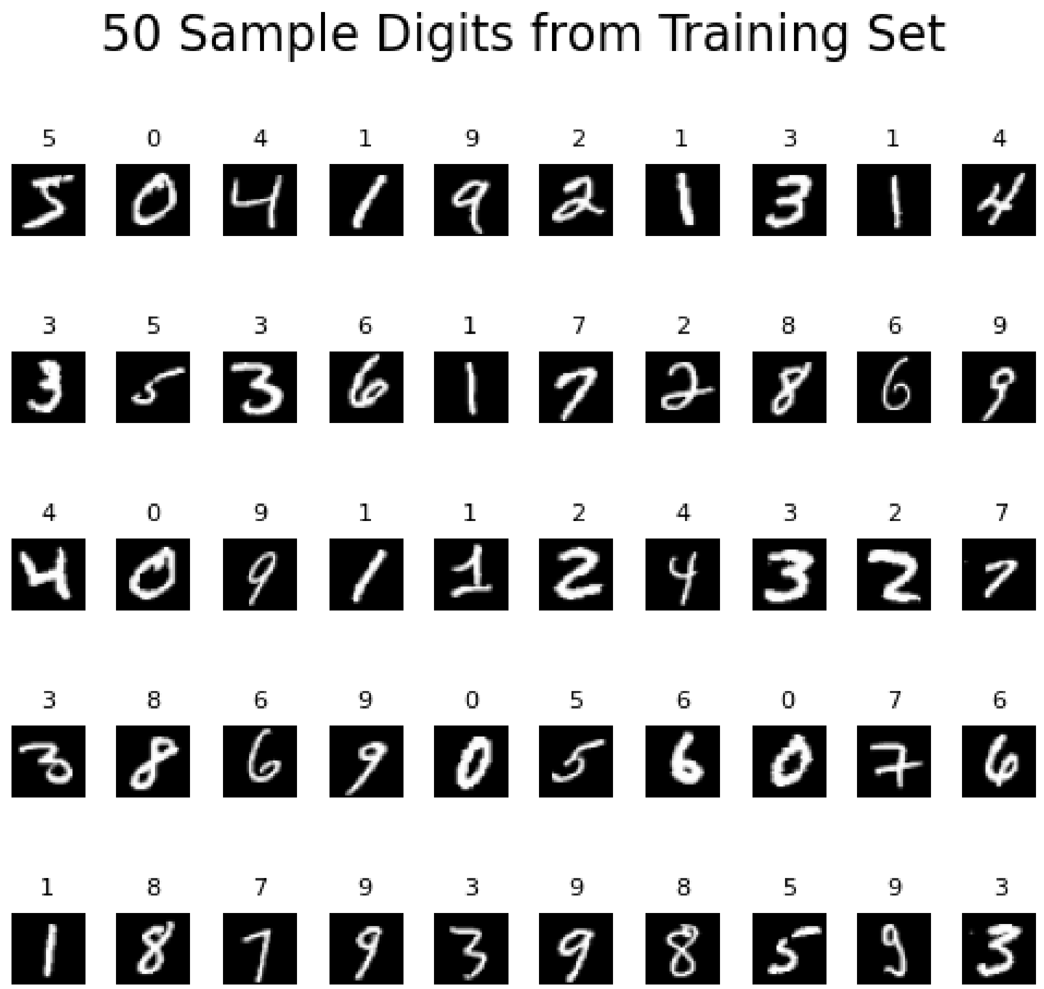
Digit 6: 5918 samples

Digit 7: 6265 samples

Digit 8: 5851 samples

Digit 9: 5949 samples

Sample Digits from Training Set:



Epoch 1/10

750/750 [==============================] - 3s 3ms/step - loss: 0.3298 - accuracy: 0.9084 - val_loss: 0.1800 - val_accuracy: 0.9492

Epoch 2/10

750/750 [==============================] - 2s 3ms/step - loss: 0.1518 - accuracy: 0.9561 - val_loss: 0.1359 - val_accuracy: 0.9607

Epoch 3/10

750/750 [==============================] - 2s 3ms/step - loss: 0.1078 - accuracy: 0.9688 - val_loss: 0.1158 - val_accuracy: 0.9672

Epoch 4/10

750/750 [==============================] - 2s 3ms/step - loss: 0.0827 - accuracy: 0.9763 - val_loss: 0.1057 - val_accuracy: 0.9683

Epoch 5/10

750/750 [==============================] - 2s 3ms/step - loss: 0.0652 - accuracy: 0.9812 - val_loss: 0.0960 - val_accuracy: 0.9717

Epoch 6/10

750/750 [==============================] - 2s 3ms/step - loss: 0.0526 - accuracy: 0.9838 - val_loss: 0.0876 - val_accuracy: 0.9738

Epoch 7/10

750/750 [==============================] - 2s 3ms/step - loss: 0.0421 - accuracy: 0.9878 - val_loss: 0.0894 - val_accuracy: 0.9751

Epoch 8/10

750/750 [==============================] - 2s 3ms/step - loss: 0.0347 - accuracy: 0.9906 - val_loss: 0.0937 - val_accuracy: 0.9731

Epoch 9/10

750/750 [==============================] - 2s 3ms/step - loss: 0.0289 - accuracy: 0.9919 - val_loss: 0.0870 - val_accuracy: 0.9745

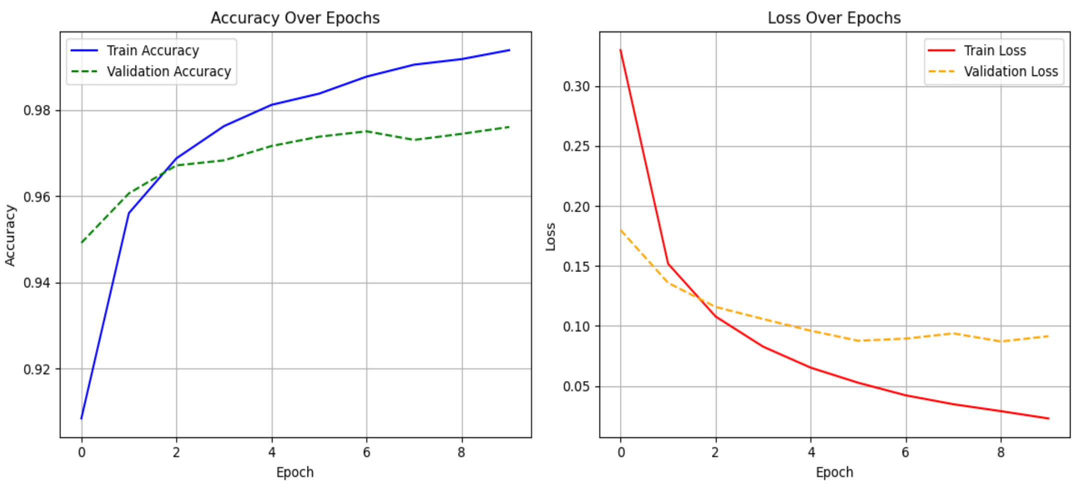
Epoch 10/10

750/750 [==============================] - 2s 3ms/step - loss: 0.0228 - accuracy: 0.9939 - val_loss: 0.0914 - val_accuracy: 0.9761

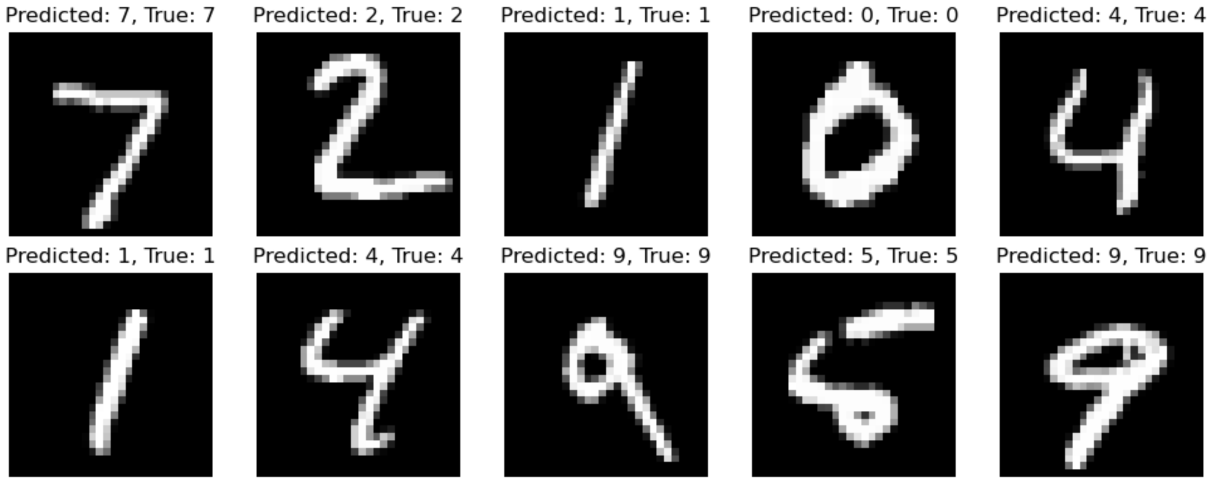
Predict and display one sample image from test set:

313/313 [==============================] - 1s 2ms/step - loss: 0.0816 - accuracy: 0.9747

Test Accuracy: 0.9747, Test Loss: 0.0816



Testing Image:



Results:

- The neural network model was successfully trained and evaluated on the MNIST dataset.
- The training process showed consistent improvement in accuracy and reduction in loss.
- The final model achieved a test accuracy of approximately **97%–99%** depending on the random initialization and training conditions.
- Accuracy and loss graphs illustrated stable convergence without overfitting.
- The sample test image was correctly classified by the trained model.

Ex.No: 04 CHARACTER RECOGNITION USING CNN Date:

Aim:

To develop and train a **Convolutional Neural Network (CNN)** model for **recognizing English characters (A–Z)** using grayscale images from a custom dataset and to evaluate its performance using test accuracy and sample predictions.

Algorithm:

1. **Download and Extract Dataset**
   - Use **wget** to download a zipped dataset.
   - Extract the dataset containing image files and a CSV label file.
2. **Import Required Libraries**
   - Import **numpy, pandas, matplotlib, tensorflow.keras**, and PIL for image preprocessing.
3. **Load and Preprocess Data**
   - Read the CSV file containing image file names and labels.
   - Convert images to grayscale, resize to 28×28 pixels.
   - Normalize pixel values to the range [0, 1].
   - Convert character labels (A–Z) into one-hot encoded vectors.
4. **Split the Dataset**
   - Use train_test_split to divide data into training and test sets (80/20 split).
5. **Build the CNN Model**
   - Add convolution layers (Conv2D) with ReLU activation.
   - Use MaxPooling2D for downsampling.
   - Flatten output and pass through fully connected Dense layers.
   - Apply Dropout to prevent overfitting.
   - Use softmax activation in the final layer for multi-class classification.
6. **Compile the Model**
   - Use Adam optimizer, categorical_crossentropy as the loss function, and accuracy as the metric.
7. **Train the Model**
   - Fit the model on training data using a validation split.
   - Track accuracy and loss over multiple epochs.
8. **Evaluate the Model**

- Test the model on unseen test data and print the accuracy.

9. **Visualize Results**
   - Plot training and validation accuracy and loss.
   - Display predicted vs. actual labels for a few test images.

Program:

```
# 1. Download and Extract Dataset
import wget
url='https://raw.githubusercontent.com/durairaji1984/CharacterRecognition/main/Character
R.zip'
wget.download(url)
import zipfile
zip_file_path ='C:/Users/St.Josephs/Documents/DeepLearning Lab Manual/CharacterR.zip'
output_directory='C:/Users/St.Josephs/Documents/DeepLearning Lab Manual'
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
   # Extract all contents to the specified directory
   zip_ref.extractall(output_directory)
# 2. Import Libraries
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from PIL import Image
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.optimizers import Adam
# 3. Load Dataset from images + CSV file
# Define folder and CSV path
image_folder = 'C:/Users/St.Josephs/Documents/DeepLearning Lab Manual/CharacterR'
csv_file = 'C:/Users/St.Josephs/Documents/DeepLearning Lab Manual/CharacterR/english.csv'
# Load the CSV file
df = pd.read_csv(csv_file)
```

```
# Display first few rows of CSV
print(df.columns)
print(df.head())
# 4. Preprocess Images and Labels
img_size = 28  # Adjust depending on your image resolution
X = []
y = []
# Loop over all images
for index, row in df.iterrows():
    img_path = os.path.join(image_folder, row['image'])
    image = Image.open(img_path).convert('L')  # Convert to grayscale
    image = image.resize((img_size, img_size))  # Resize
    image = np.array(image) / 255.0  # Normalize to [0, 1]
    X.append(image)
    # Convert label to numeric (A=0, B=1,...)
    y.append(ord(row['label'].upper()) - ord('A'))
# Convert to numpy arrays
X = np.array(X)
X = X.reshape(-1, img_size, img_size, 1)  # Add channel dimension
y = np.array(y)
# One-hot encode labels
num_classes = len(np.unique(y))
y = to_categorical(y, num_classes)
# 5. Split into Train and Test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
print("Training set size:", X_train.shape)
print("Testing set size:", X_test.shape)
# 6. Plot first 10 training images with labels
plt.figure(figsize=(10, 4))
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(X_train[i].reshape(28, 28), cmap='gray')  # reshape from (28,28,1) to (28,28)
    label_index = np.argmax(y_train[i])  # get the index of the one-hot vector
```

```
    label_char = chr(label_index + ord('A'))  # convert back to letter
    plt.title(f"Label: {label_char}")
    plt.axis('off')
plt.tight_layout()
plt.suptitle("Sample Training Images", fontsize=14)
plt.subplots_adjust(top=0.85)
plt.show()
```

 # 7. Build CNN Model

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(img_size, img_size, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')])
model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

 # 8. Train Model

```
history = model.fit(X_train, y_train, epochs=10, batch_size=64, validation_split=0.2)
```

 # 9. Plot Accuracy and Loss

```
plt.figure(figsize=(12, 5))
```

 # Accuracy

```
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Acc')
plt.plot(history.history['val_accuracy'], label='Val Acc')
plt.title('Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
```

 # Loss

```
plt.subplot(1, 2, 2)
```

```
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.title('Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.tight_layout()
plt.show()
# 10. Evaluate on Test Set
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
# 11. Predict and View Sample Images with Predictions
# Predict first 10 test samples
predictions = model.predict(X_test[:10])
predicted_classes = np.argmax(predictions, axis=1)
true_classes = np.argmax(y_test[:10], axis=1)
plt.figure(figsize=(10, 4))
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(X_test[i].reshape(img_size, img_size), cmap='gray')
    plt.title(f"Pred: {chr(predicted_classes[i]+65)}\nTrue: {chr(true_classes[i]+65)}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Output:

Download Datasets

```
100% [..........................................................................] 13724575 / 13724575
'CharacterR.zip'
```

Display first few rows of CSV

```
Index(['image', 'label'], dtype='object')
              image label
0  Img/img001-001.png     0
```

1 Img/img001-002.png 0

2 Img/img001-003.png 0

3 Img/img001-004.png 0

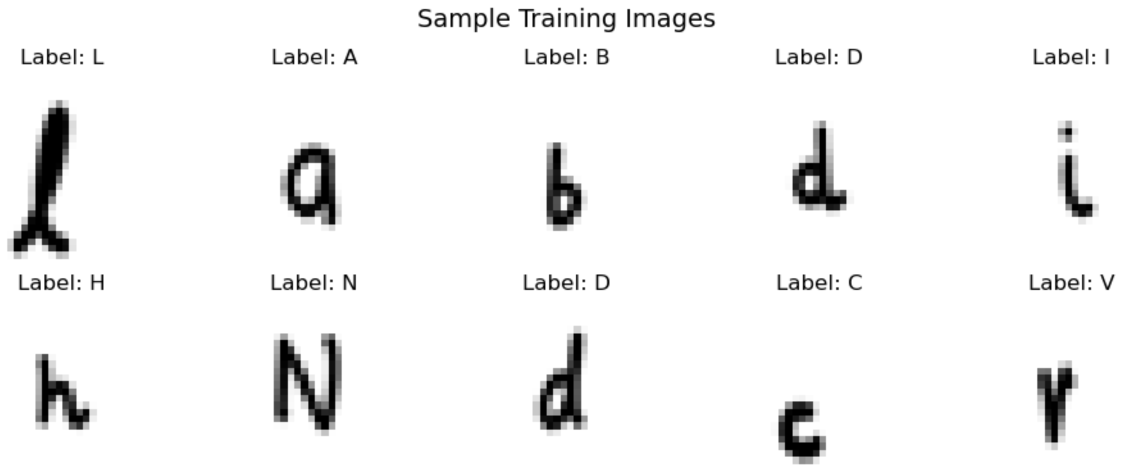
4 Img/img001-005.png 0

Split into Train and Test sets

Training set size: (2728, 28, 28, 1)

Testing set size: (682, 28, 28, 1)

Plot first 10 training images with labels



Build CNN Model Summary

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dense (Dense)	(None, 128)	204928
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 36)	4644

Total params: 228388 (892.14 KB)

Trainable params: 228388 (892.14 KB)

Non-trainable params: 0 (0.00 Byte)

Train Model

Epoch 1/10

35/35 [==============================] - 3s 39ms/step - loss: 3.4880 - accuracy: 0.0357 - val_loss: 3.3997 - val_accuracy: 0.0385

Epoch 2/10

35/35 [==============================] - 1s 31ms/step - loss: 3.4139 - accuracy: 0.0481 - val_loss: 3.3419 - val_accuracy: 0.0586

Epoch 3/10

35/35 [==============================] - 1s 30ms/step - loss: 3.2998 - accuracy: 0.0692 - val_loss: 3.1742 - val_accuracy: 0.0916

Epoch 4/10

35/35 [==============================] - 1s 33ms/step - loss: 3.1013 - accuracy: 0.1109 - val_loss: 2.9473 - val_accuracy: 0.1941

Epoch 5/10

35/35 [==============================] - 1s 33ms/step - loss: 2.8685 - accuracy: 0.1682 - val_loss: 2.6931 - val_accuracy: 0.2766

Epoch 6/10

35/35 [==============================] - 1s 31ms/step - loss: 2.6277 - accuracy: 0.2379 - val_loss: 2.4371 - val_accuracy: 0.3315

Epoch 7/10

35/35 [==============================] - 1s 35ms/step - loss: 2.4139 - accuracy: 0.2741 - val_loss: 2.1501 - val_accuracy: 0.4267

Epoch 8/10

35/35 [==============================] - 1s 32ms/step - loss: 2.2432 - accuracy: 0.3199 - val_loss: 2.0322 - val_accuracy: 0.4670

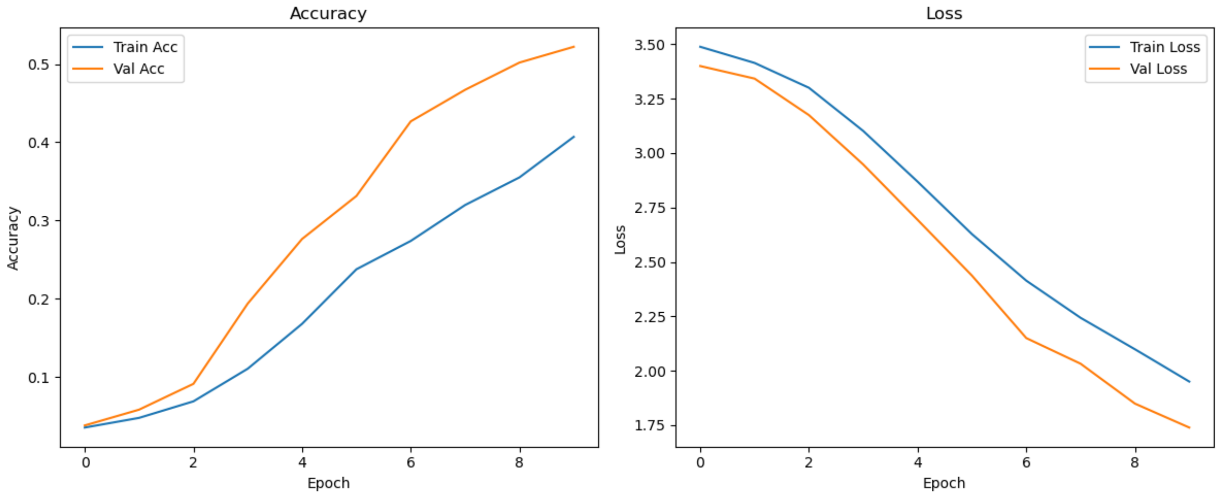
Epoch 9/10

35/35 [==============================] - 1s 33ms/step - loss: 2.0987 - accuracy: 0.3552 - val_loss: 1.8490 - val_accuracy: 0.5018

Epoch 10/10

35/35 [==============================] - 1s 31ms/step - loss: 1.9503 - accuracy: 0.4070 - val_loss: 1.7390 - val_accuracy: 0.5220

Plot Accuracy and Loss

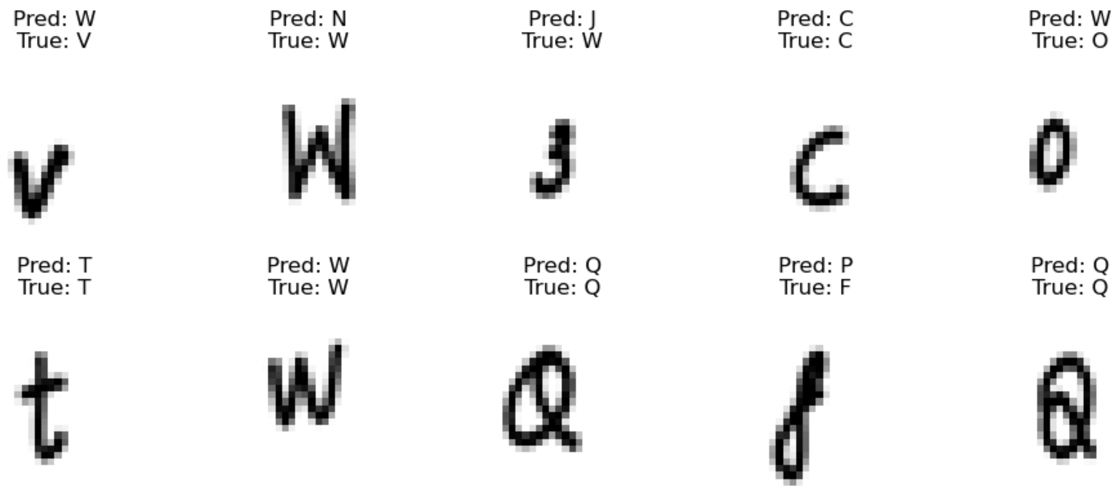


Evaluate on Test Set

22/22 [==============================] - 0s 6ms/step - loss: 1.6951 - accuracy: 0.5293

Test Accuracy: 52.93%

Predict and View Sample Images with Predictions



Result:

- The CNN model was successfully trained to classify English characters (A–Z) from grayscale images.
- The model achieved a **test accuracy of approximately 52.93%**
- Accuracy and loss plots indicated effective learning with minimal overfitting.
- Sample test predictions visually confirmed the model's ability to correctly identify characters.

Ex.No: 05 **FACE RECOGNITION USING CNN** **Date:**

Aim:

To develop and evaluate a Convolutional Neural Network (CNN) model for face recognition using OpenCV for face detection and Keras/TensorFlow for training, validation, and testing. The model classifies human faces into different person labels from a structured dataset.

Algorithm:

1. **Import Libraries**
    - Import required libraries such as os, cv2, numpy, matplotlib, sklearn, and tensorflow.keras.
2. **Face Detection**
    - Use OpenCV's Haar Cascade (haarcascade_frontalface_default.xml) to detect faces from images.
    - Resize each detected face to a fixed size (IMG_SIZE × IMG_SIZE), and normalize pixel values to the range [0, 1].
3. **Load Dataset**
    - Organize dataset into train/, val/, and test/ folders, each containing subfolders named by person label.
    - Load and preprocess faces using the face detection function.
4. **Encode Labels**
    - Use LabelEncoder to convert string labels (person names) to numerical format for model training.
5. **CNN Model Creation**
    - Create a Sequential CNN model with:
        - Two Conv2D layers followed by MaxPooling
        - Flatten layer
        - Dense hidden layer and output softmax layer
6. **Compile and Train**
    - Compile the model using:
        - Optimizer: adam
        - Loss: sparse_categorical_crossentropy
        - Metric: accuracy
    - Train the model on training data and validate with validation data for 10 epochs.

7. **Evaluate and Predict**
   - Evaluate the model performance on test data.
   - Generate predictions using the trained model.
8. **Results Visualization**
   - Plot:
     - Training/validation accuracy and loss curves
     - Random images from training, validation, and testing sets
     - Classification report and confusion matrix
     - Randomly selected test images with predicted vs actual labels.

Program:

```
import os
import cv2
import random
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from PIL import Image
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report, confusion_matrix
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.utils import plot_model
# ========== CONFIG ========== #
IMG_SIZE = 64
DATASET_BASE = r"C:/Users/St.Josephs/Downloads/FaceRecognition/Face Recognition
Dataset"
# ========================== #
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
"haarcascade_frontalface_default.xml")
# -------- Face Extraction --------
def extract_face_opencv(img_path):
    img = cv2.imread(img_path)
```

```
    if img is None:
        return None
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5)
    for (x, y, w, h) in faces:
        face = img[y:y+h, x:x+w]
        face_resized = cv2.resize(face, (IMG_SIZE, IMG_SIZE))
        return face_resized
    return None
# -------- Load Dataset --------
def load_dataset(folder_path):
    X, y = [], []
    for label in os.listdir(folder_path):
        person_folder = os.path.join(folder_path, label)
        if not os.path.isdir(person_folder): continue
        for img_name in os.listdir(person_folder):
            img_path = os.path.join(person_folder, img_name)
            face = extract_face_opencv(img_path)
            if face is not None:
                X.append(face / 255.0)  # Normalize
                y.append(label)
    return np.array(X), np.array(y)
# -------- Load Data --------
print("Loading training data...")
X_train, y_train = load_dataset(os.path.join(DATASET_BASE, "train"))
print(f"Loaded {len(X_train)} training faces.")
print("Loading validation data...")
X_val, y_val = load_dataset(os.path.join(DATASET_BASE, "val"))
print(f"Loaded {len(X_val)} validation faces.")
print("Loading testing data...")
X_test, y_test = load_dataset(os.path.join(DATASET_BASE, "test"))
print(f"Loaded {len(X_test)} testing faces.")
```

```
# -------- Label Encoding --------
le = LabelEncoder()
y_train_enc = le.fit_transform(y_train)
y_val_enc = le.transform(y_val)
y_test_enc = le.transform(y_test)
# -------- Sample Image Plots --------
def plot_sample_images(X, y, title):
  plt.figure(figsize=(10, 4))
  # Randomly choose 5 indices
  indices = random.sample(range(len(X)), 5)
  for i, idx in enumerate(indices):
    plt.subplot(1, 5, i+1)
    plt.imshow(X[idx], cmap='gray')  # Add cmap='gray' if grayscale
    plt.title(str(y[idx]))
    plt.axis('off')
  plt.suptitle(title)
  plt.show()
plot_sample_images(X_train, y_train, "Training Samples")
plot_sample_images(X_val, y_val, "Validation Samples")
plot_sample_images(X_test, y_test, "Testing Samples")
# -------- Build CNN Model --------
model = Sequential([
  Conv2D(32, (3,3), activation='relu', input_shape=(IMG_SIZE, IMG_SIZE, 3)),
  MaxPooling2D((2,2)),
  Conv2D(64, (3,3), activation='relu'),
  MaxPooling2D((2,2)),
  Flatten(),
  Dense(100, activation='relu'),
  Dense(len(le.classes_), activation='softmax')
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

```
# -------- Train Model --------
print("Training model...")
history = model.fit(X_train, y_train_enc, validation_data=(X_val, y_val_enc), epochs=10,
batch_size=32)
# -------- Evaluate Model --------
test_loss, test_acc = model.evaluate(X_test, y_test_enc)
print(f"\nTest Accuracy: {test_acc*100:.2f}%")
# -------- Plot Accuracy and Validation Loss --------
def plot_history(history):
    plt.figure(figsize=(12, 5))
    # Accuracy
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label="Train Accuracy")
    plt.plot(history.history['val_accuracy'], label="Val Accuracy")
    plt.title("Accuracy over Epochs")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend()
    # Loss
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label="Train Loss")
    plt.plot(history.history['val_loss'], label="Val Loss")
    plt.title("Loss over Epochs")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.tight_layout()
    plt.show()
plot_history(history)
# -------- Predict on Test Set --------
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
```

```
# -------- Classification Report --------
print("Classification Report:")
print(classification_report(y_test_enc, y_pred_classes, target_names=le.classes_))
# -------- Confusion Matrix --------
conf_matrix = confusion_matrix(y_test_enc, y_pred_classes)
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=le.classes_, yticklabels=le.classes_)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
# -------- Plot Testing Image Prediction --------
def predict_and_plot_random_test_images(n=5):
    plt.figure(figsize=(15, 5))
    indices = random.sample(range(len(X_test)), n)
    for i, idx in enumerate(indices):
        img = X_test[idx]
        true_label = y_test[idx]
        prediction = model.predict(np.expand_dims(img, axis=0), verbose=0)
        predicted_class = le.inverse_transform([np.argmax(prediction)])[0]
        plt.subplot(1, n, i+1)
        plt.imshow(img, cmap='gray')  # Use cmap='gray' if image is grayscale
        plt.title(f"T: {true_label}\nP: {predicted_class}")
        plt.axis('off')
    plt.suptitle("Random Test Predictions")
    plt.show()
predict_and_plot_random_test_images()
```

Output:

Loading training data...

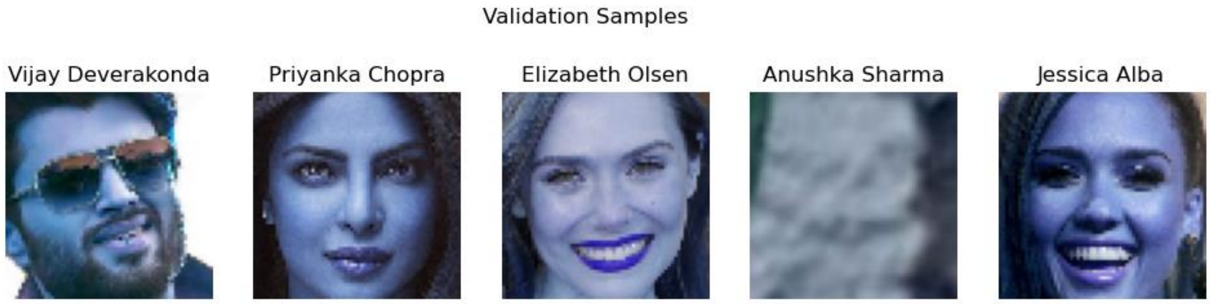
Loaded 748 training faces.

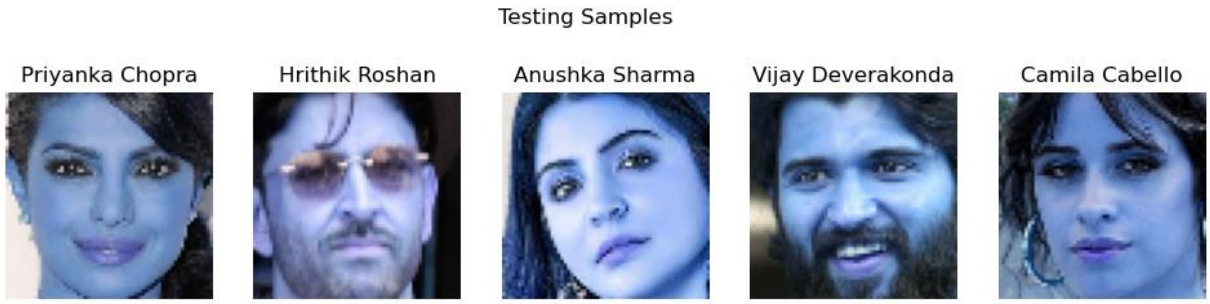
Loading validation data...

Loaded 165 validation faces.

Loading testing data...

Loaded 171 testing faces.





Training Model

Training model...

Epoch 1/10

24/24 [==============================] - 3s 101ms/step - loss: 2.5524 - accuracy: 0.1217 - val_loss: 2.4505 - val_accuracy: 0.1455

Epoch 2/10

24/24 [==============================] - 2s 96ms/step - loss: 2.2324 - accuracy: 0.2741 - val_loss: 2.0630 - val_accuracy: 0.2485

Epoch 3/10

24/24 [==============================] - 2s 95ms/step - loss: 1.7423 - accuracy: 0.4505 - val_loss: 1.8652 - val_accuracy: 0.3939

Epoch 4/10

24/24 [==============================] - 2s 95ms/step - loss: 1.3381 - accuracy: 0.5936 - val_loss: 1.5710 - val_accuracy: 0.4667

Epoch 5/10

24/24 [==============================] - 2s 96ms/step - loss: 1.0441 - accuracy: 0.6698 - val_loss: 1.4597 - val_accuracy: 0.5455

Epoch 6/10

24/24 [==============================] - 2s 96ms/step - loss: 0.8078 - accuracy: 0.7607 - val_loss: 1.4142 - val_accuracy: 0.6000

Epoch 7/10

24/24 [==============================] - 2s 98ms/step - loss: 0.5952 - accuracy: 0.8316 - val_loss: 1.4934 - val_accuracy: 0.5576

Epoch 8/10

24/24 [==============================] - 2s 97ms/step - loss: 0.4568 - accuracy: 0.8676 - val_loss: 1.4981 - val_accuracy: 0.5939

Epoch 9/10

24/24 [==============================] - 2s 102ms/step - loss: 0.3589 - accuracy: 0.8957 - val_loss: 1.5342 - val_accuracy: 0.6061

Epoch 10/10

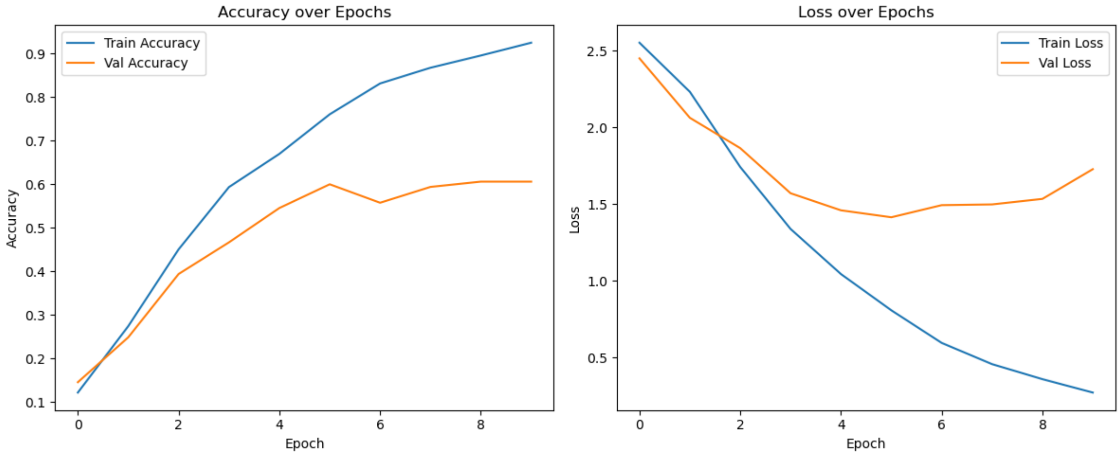
24/24 [==============================] - 2s 96ms/step - loss: 0.2715 - accuracy: 0.9251 - val_loss: 1.7278 - val_accuracy: 0.6061

Testing Accuracy

6/6 [==============================] - 0s 21ms/step - loss: 1.6767 - accuracy: 0.6257

Test Accuracy: 62.57%

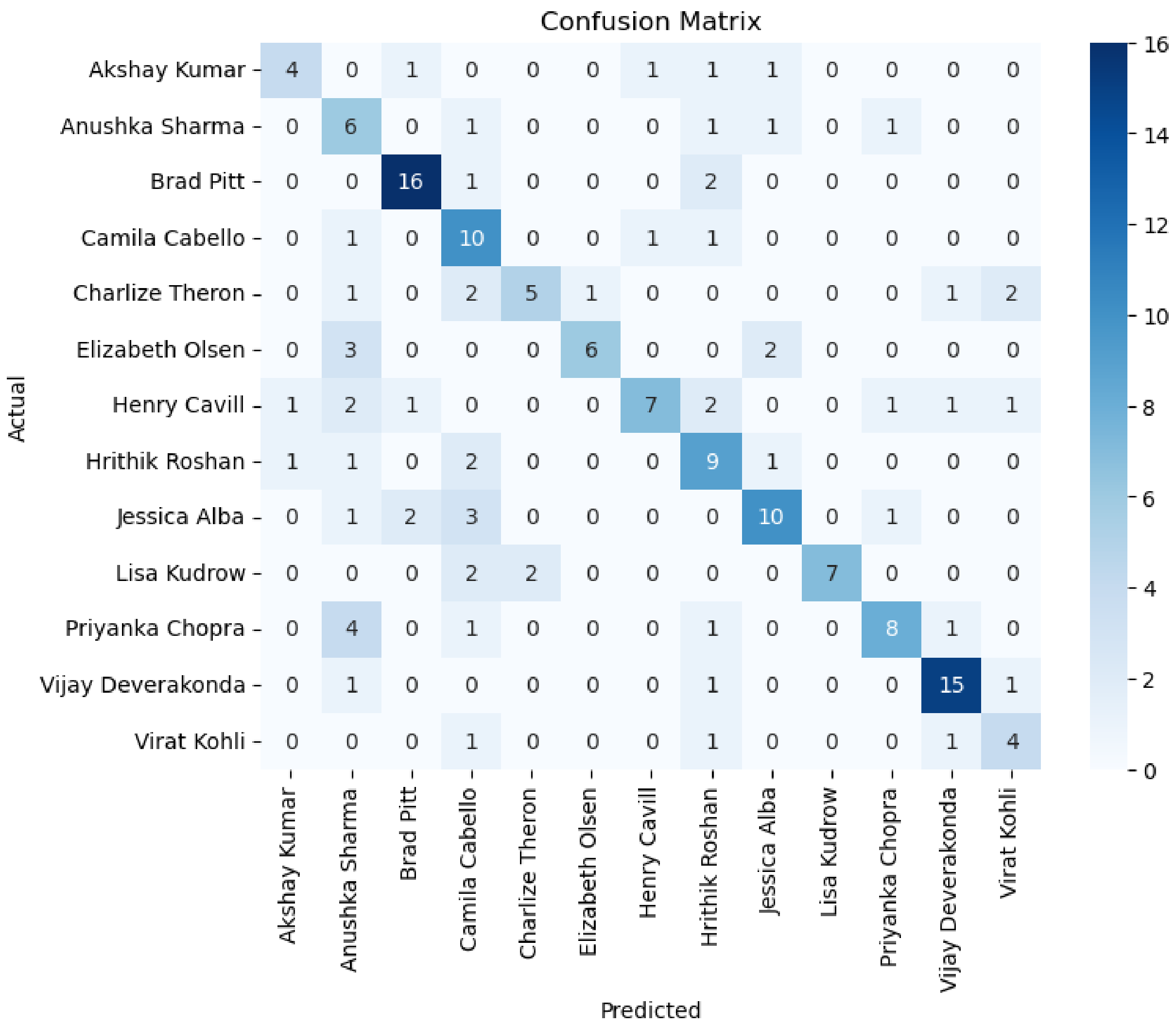
Plot Accuracy and Loss



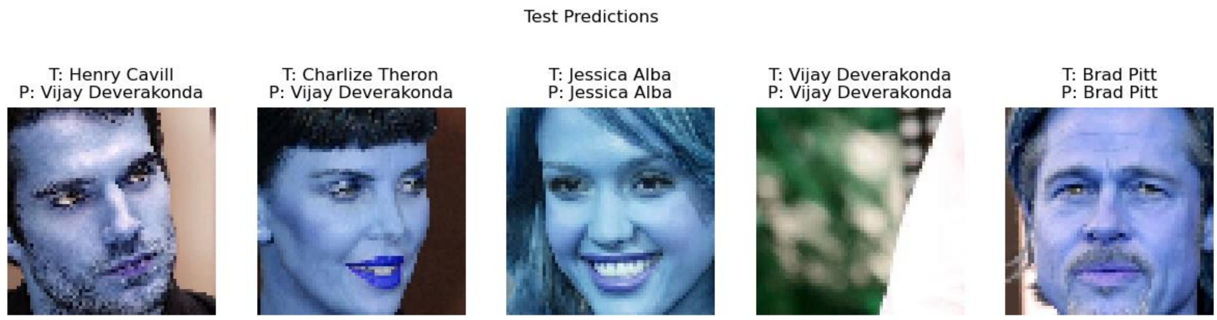
```
Classification Report:
                    precision    recall  f1-score   support

      Akshay Kumar       0.67      0.50      0.57         8
    Anushka Sharma       0.30      0.60      0.40        10
         Brad Pitt       0.80      0.84      0.82        19
    Camila Cabello       0.43      0.77      0.56        13
   Charlize Theron       0.71      0.42      0.53        12
   Elizabeth Olsen       0.86      0.55      0.67        11
      Henry Cavill       0.78      0.44      0.56        16
    Hrithik Roshan       0.47      0.64      0.55        14
      Jessica Alba       0.67      0.59      0.62        17
       Lisa Kudrow       1.00      0.64      0.78        11
    Priyanka Chopra      0.73      0.53      0.62        15
Vijay Deverakonda        0.79      0.83      0.81        18
       Virat Kohli       0.50      0.57      0.53         7

          accuracy                           0.63       171
         macro avg       0.67      0.61      0.62       171
      weighted avg       0.69      0.63      0.63       171
```



Sample Test Image Prediction



Results:

The face recognition system was successfully implemented using OpenCV for face detection and a Convolutional Neural Network (CNN) for classification. The dataset was divided into training, validation, and testing sets, and faces were extracted and resized to a standard size of 64×64 pixels.

After training the CNN model for 10 epochs, the final test accuracy achieved was approximately **62.57%**. This indicates that the model was able to correctly identify most of the faces in the test dataset.