DS1505-P

FUNDAMENTALS OF DEEP LEARNING (LAB INTEGRATED)

DS1505	FUNDAMENTALS OF DEEP LEARNING (LAB INTEGRATED)	L	T	P	C
		4	0	2	4

OBJECTIVES

- To understand the basic ideas and principles of neural networks.
- To understand the basic concepts of deep learning.
- To appreciate the use of deep learning applications.
- To know the applications of Deep learning techniques to NLP
- To build solutions for real world problems.

UNIT I	FUNDAMENTALS OF DEEP NETWORKS	9+6

CO1

Introduction-Linear Algebra-Probability and Information Theory-Numerical Computation-Machine Learning Basics.

Lab Component:

Demonstration and implementation of Shallow architecture, using 10 hours Python, Tensorflow and Keras

- Google Colaboratory - Cloning GitHub repository, Upload Data, Importing Kaggle's dataset, Basic File operations
- Implementing Perceptron
- Digit Classification : Neural network to classify MNIST(Modified **National Institute of Standards and Technology**) dataset

Solving XOR problem using DNN

UNIT II	DEEP NETWORKS: MODERN PRACTICES	9+6

CO2

Deep Feedforward Networks: Simple Deep Neural Network-Generic Deep Neural Network-Computations in Deep Neural Network-Gradient-Based Learning. Regularization for deep learning: L2 regularization-L1 Regularization-Entropy Regularization-Dropout-Data augmentation. Optimization for Training deep models: Learning Differs from Pure Optimization-Challenges in Neural Network Optimization-Stochastic Gradient Descent.

Lab Component:

Classification of MNIST Dataset using CNN

Character recognition using CNN

UNIT III	CONVOLUTIONAL AND RECURRENT NEURAL NETWORKS	9+6

CO3

Introduction-Convolutional Operation-Pooling-Data Types-Convolution Algorithms-Convolutional Networks with Deep Learning. Sequence Modeling: Recurrent and Recursive Nets: Introduction-Auto-Completion-Unfolding Computational Graphs-Recurrent Neural Networks-

Types of RNNs-Bidirectional RNNs-Sequence-to-Sequence Architectures-Deep Recurrent Networks-Long-Term Dependencies;Gated Architecture:LSTM; Lab Component: Face recognition using CNN Language modeling using RNN Sentiment analysis using LSTM Parts of speech tagging using Sequence to Sequence architecture			
UNIT IV	**DEEP LEARNING RESEARCH**		9+6
Linear Factor Models-Auto encoders: Undercomplete Autoencoders-Regularized Autoencoders-Stochastic Encoders and Decoders-Denoising Autoencoders-Learning with Autoencoders; Deep Generative Models; Variational autoencoders-Generative adversarial networks. Lab Component: Machine Translation using Encoder-Decoder model Image augmentation using GANs			CO4
UNIT V	**APPLICATIONS OF DEEP LEARNING TO NLP**		9+6
Introduction to NLP and Vector Space Model of Semantics - Word Vector Representations: Continuous Skip-Gram Model - Continuous Bag-of-Words model(CBOW) - Glove - Evaluations and Applications in word similarity. Lab Component: Mini-project on real world applications			CO5
THEORY : 45 PERIODS	**PRACTICAL : 30 PERIODS**	**TOTAL : 75 PERIODS**	

TEXT BOOKS

1. Ian Goodfellow, Yoshua Bengio, Aaron Courville,“DeepLearning”,MITPress,2018.
2. Francois Chollet, “Deep Learning with Python”, Manning Publications, 2018
3. Amit kumar Das, Saptarsi Goswami, Pabitra Mitra, Amlan Chakrabarti —Deep Learning"', Pearson Education, 2022.

REFERENCE BOOKS

1. Li Deng, Dong Yu, Deep Learning: Methods and Applications, NOW Publishers, 2014.
2. Charu C. Aggarwal, —Neural Networks and Deep Learning: A Textbook‖', Springer International Punlishing, 2018.
3. Nikhil Buduma and Nicholas Locascio, Fundamentals of Deep Learning: DesigningNextGeneration Artificial Intelligence Algorithms, O'Reilly Media, 2017.
4. Stone, James. (2019). Artificial Intelligence Engines: A Tutorial Introduction to the Mathematics of Deep Learning, Sebtel Press, United States, 2019.
5. Navin Kumar Manaswi :Deep Learning with Applications Using Python,2018

COURSE OUTCOMES **Upon completion of the course, students will be able to**	
CO1	Understand the role of deep learning in machine learning applications
CO2	Design and implement deep learning applications
CO3	Critically analyze different deep learning models in image related projects.
CO4	Design and implement convolutional neural networks
CO5	Know about applications of deep learning in NLP and image processing

MAPPING OF COs WITH POs AND PSOs

COs	PROGRAM OUTCOMES (POs)												PROGRAM SPECIFIC OUTCOMES (PSOs)		
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	2	2	2	2	-	-	-	-	2	3	3	3	1	2
CO2	3	2	2	2	3	-	-	-	-	2	3	3	3	1	2
CO3	3	2	2	3	3	-	-	-	-	2	2	3	3	3	3
CO4	3	2	2	3	3	-	-	-	-	2	2	3	3	3	3
CO5	3	2	2	2	2	-	-	-	-	2	3	3	3	1	2

Name of the Lab: DS1505-P Fundamentals of Deep Learning (Lab Integrated)

Staff Name:

S.No.	Date of Experiment	Name of the Experiment	Date of Submission	Signature
1.		IMPLEMENTING PERCEPTRON		
2.		SOLVING XOR PROBLEM USING DNN		
3.		DIGIT CLASSIFICATION USING NEURAL NETWORK (MNIST DATASET)		
4.		CHARACTER RECOGNITION USING CNN		
5.		FACE RECOGNITION USING CNN		
6.		LANGUAGE MODELING USING RNN		
7.		SENTIMENT ANALYSIS USING LSTM		
8.		PARTS OF SPEECH TAGGING USING SEQUENCE TO SEQUENCE ARCHITECTURE		
9.		MACHINE TRANSLATION USING ENCODER–DECODER MODEL (ENGLISH → TAMIL)		
10.		IMAGE AUGMENTATION USING GANS		
11.				

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

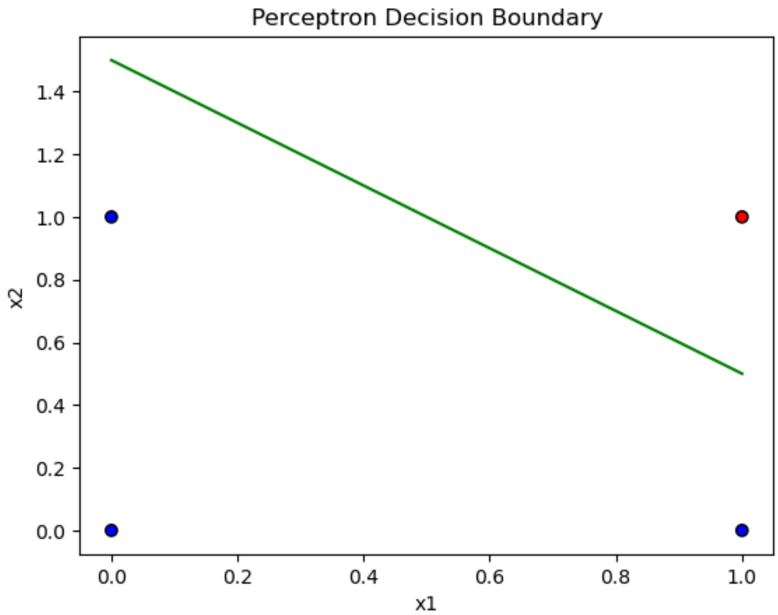
Test the trained perceptron:

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):

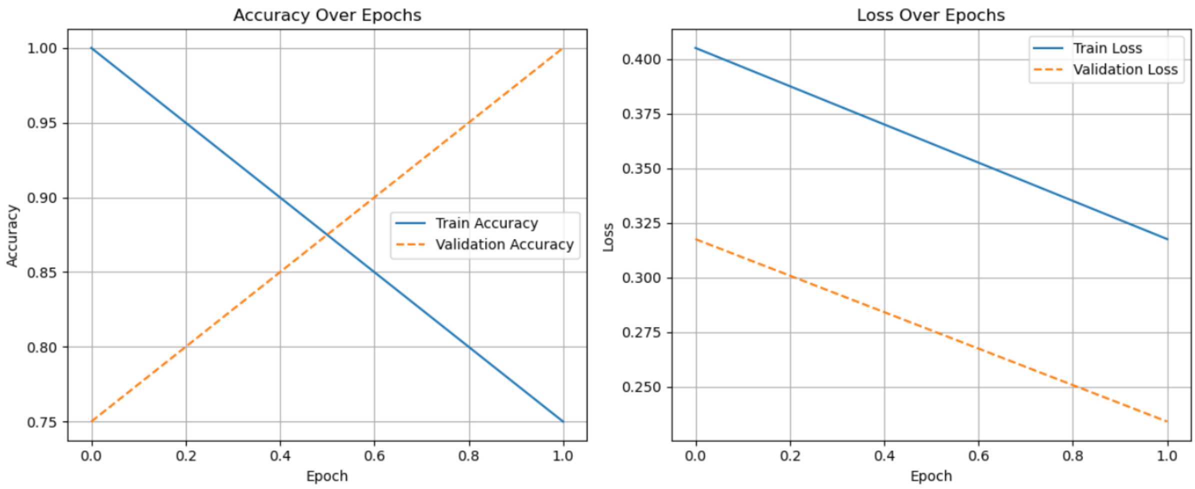
1/1 [==============================] - 0s 19ms/step

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

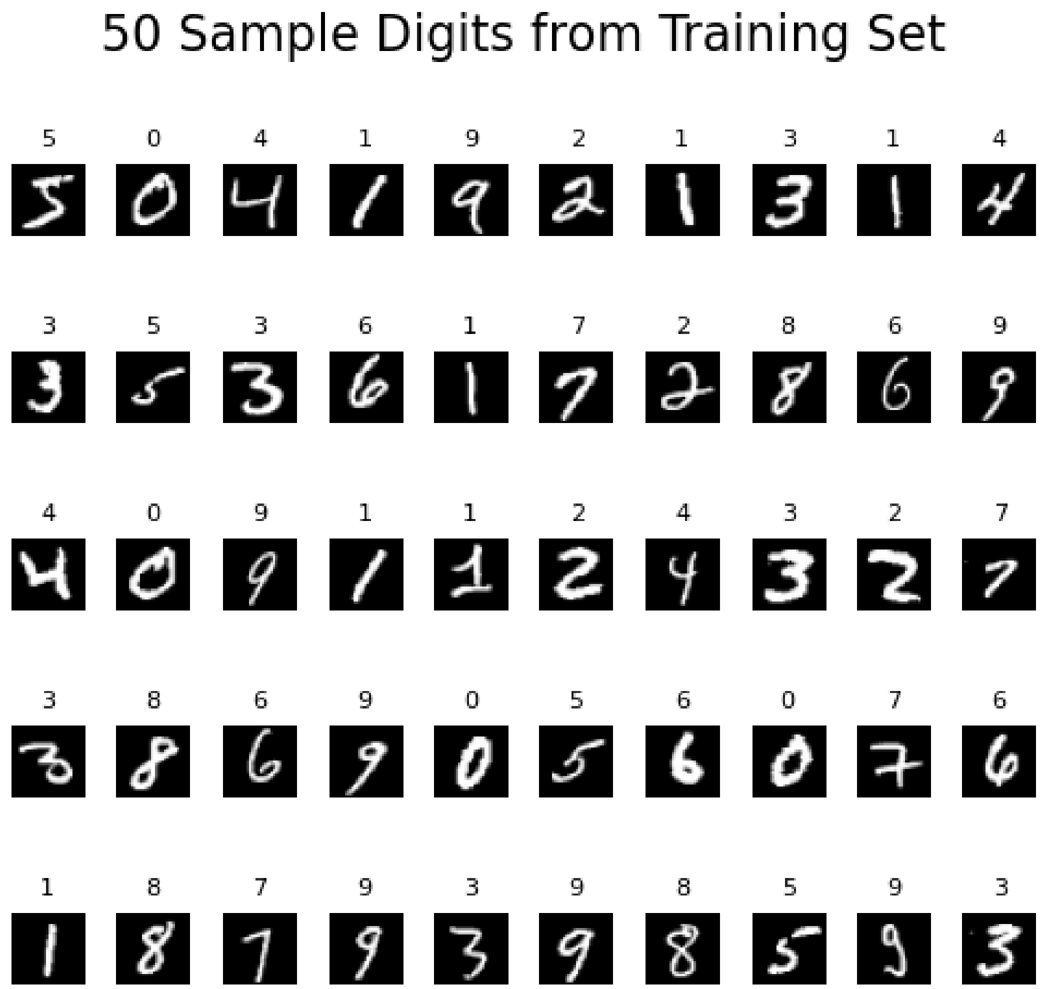
Digit 6: 5918 samples

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

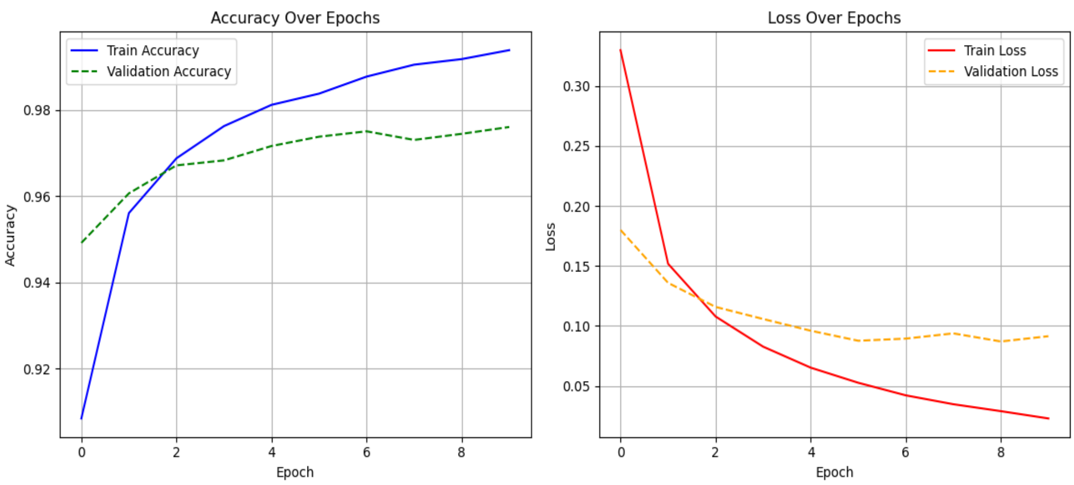
Epoch 10/10

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

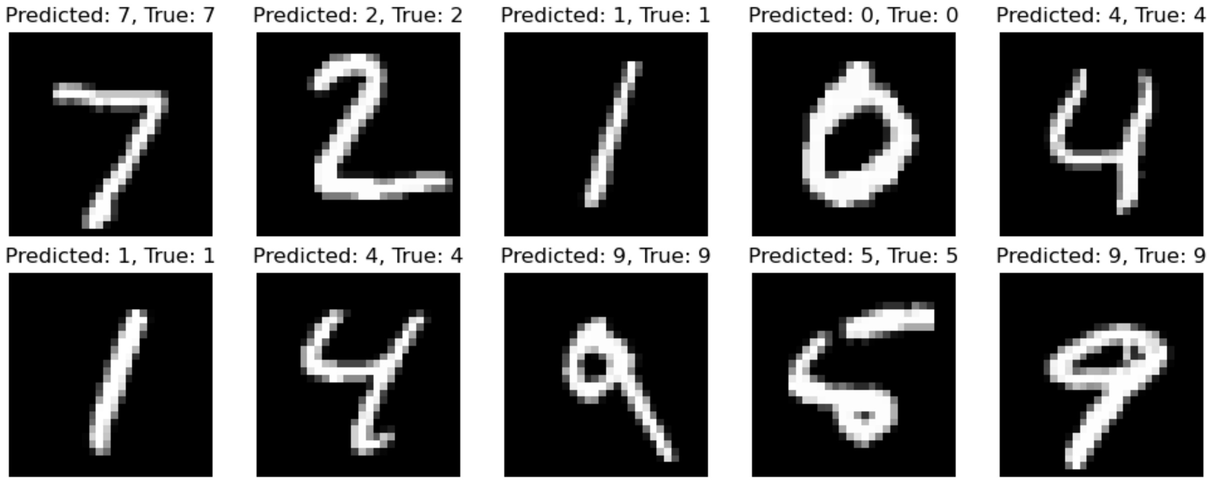
Predict and display one sample image from test set:

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
```

```python
    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

```python
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

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

 # 9. Plot Accuracy and Loss

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

 # Accuracy

```python
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

```python
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

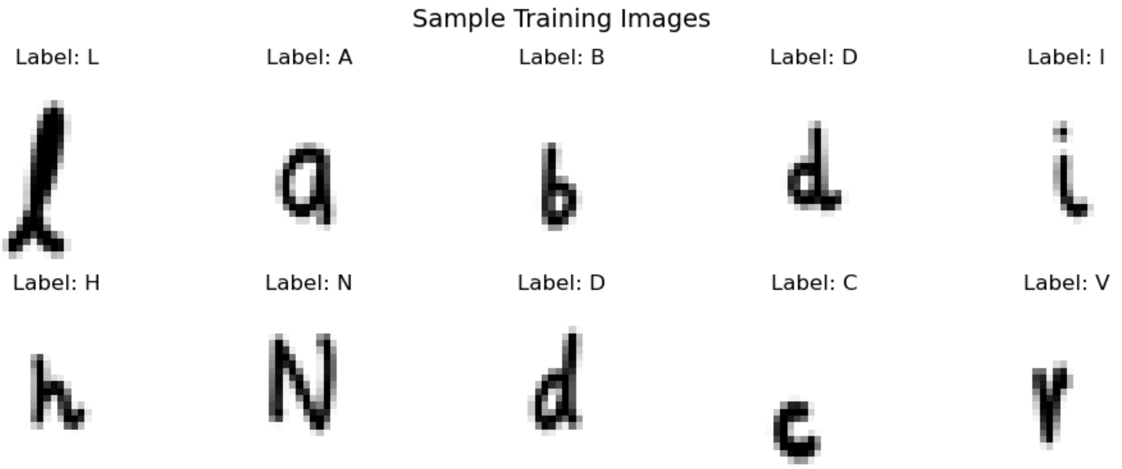
4 Img/img001-005.png 0

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

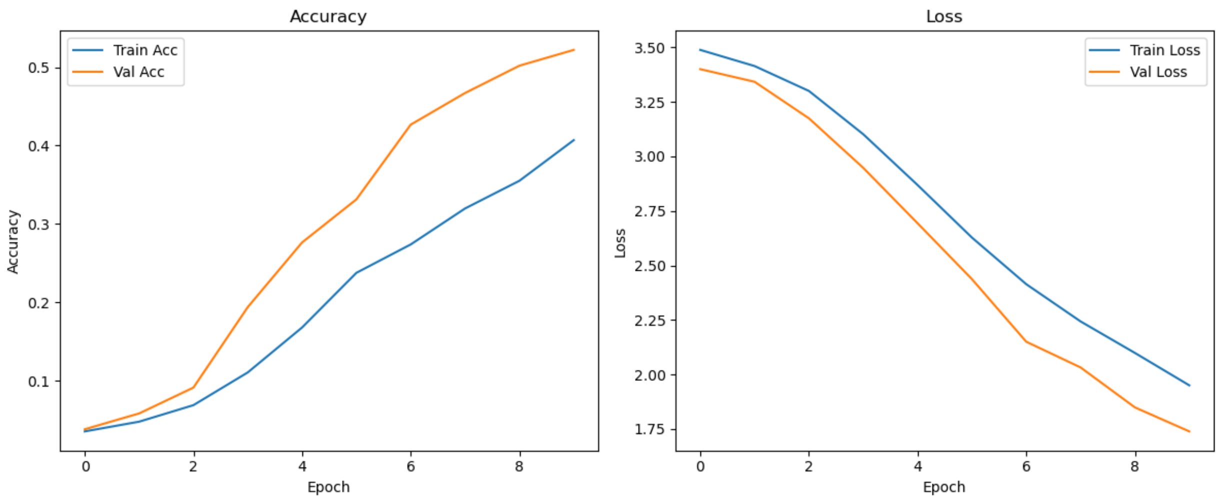
Epoch 9/10

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

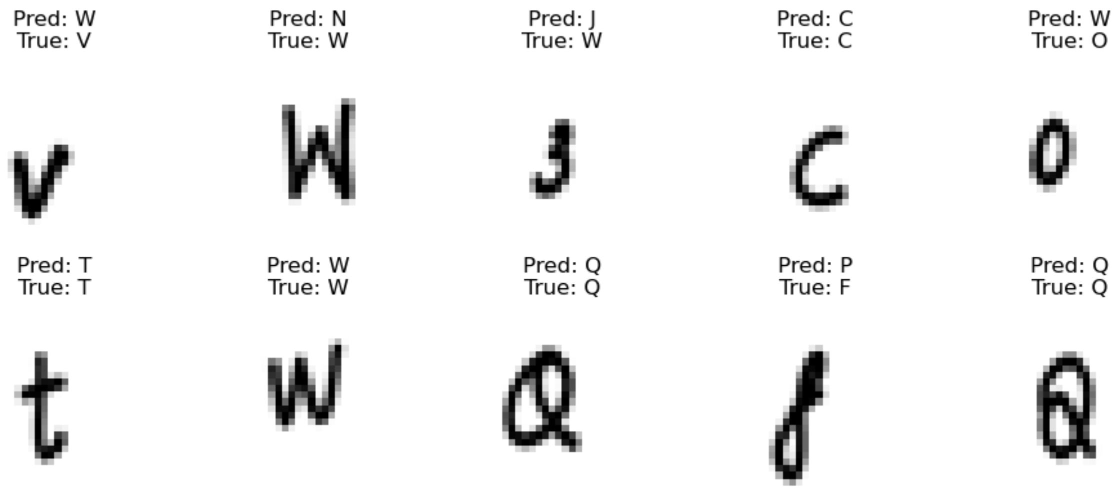


Evaluate on Test Set

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...

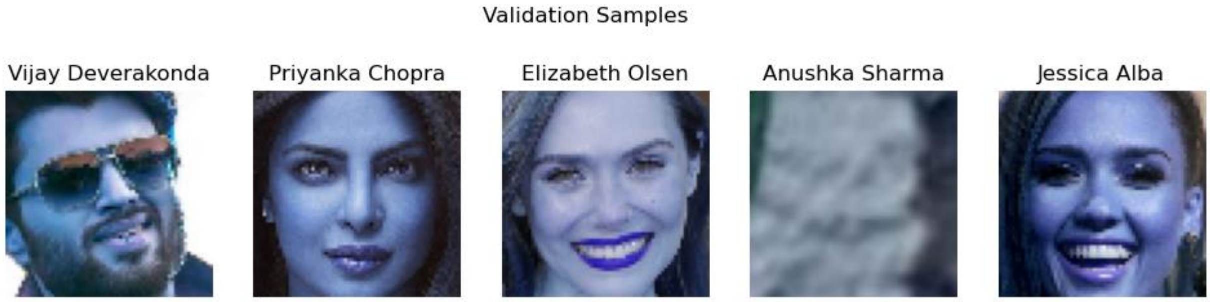
Loaded 748 training faces.

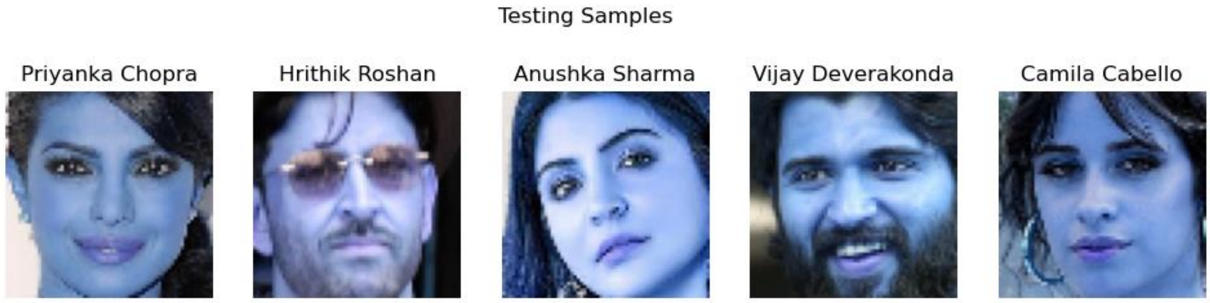
Loading validation data...

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

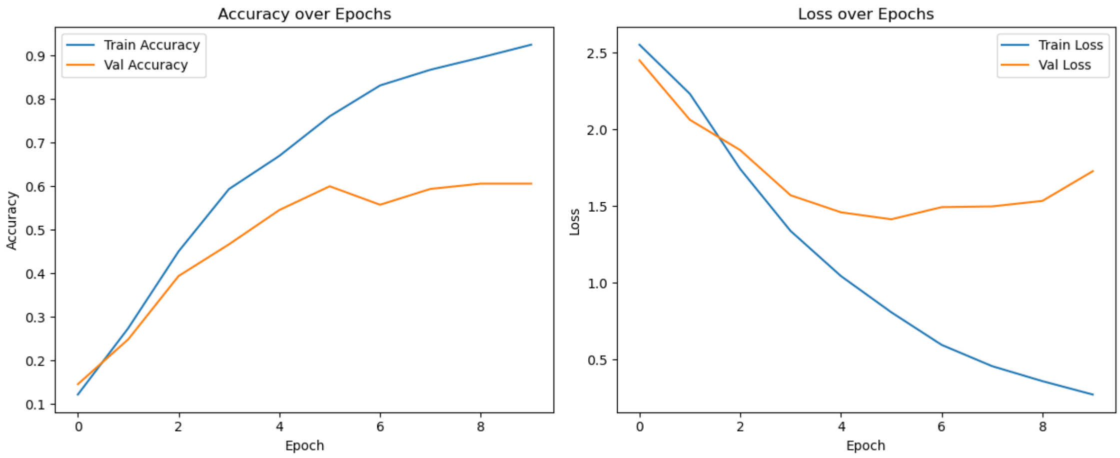
24/24 [==============================] - 2s 96ms/step - loss: 0.2715 - accuracy: 0.9251 - val_loss: 1.7278 - val_accuracy: 0.6061

Testing Accuracy

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

Test Accuracy: 62.57%

Plot Accuracy and Loss

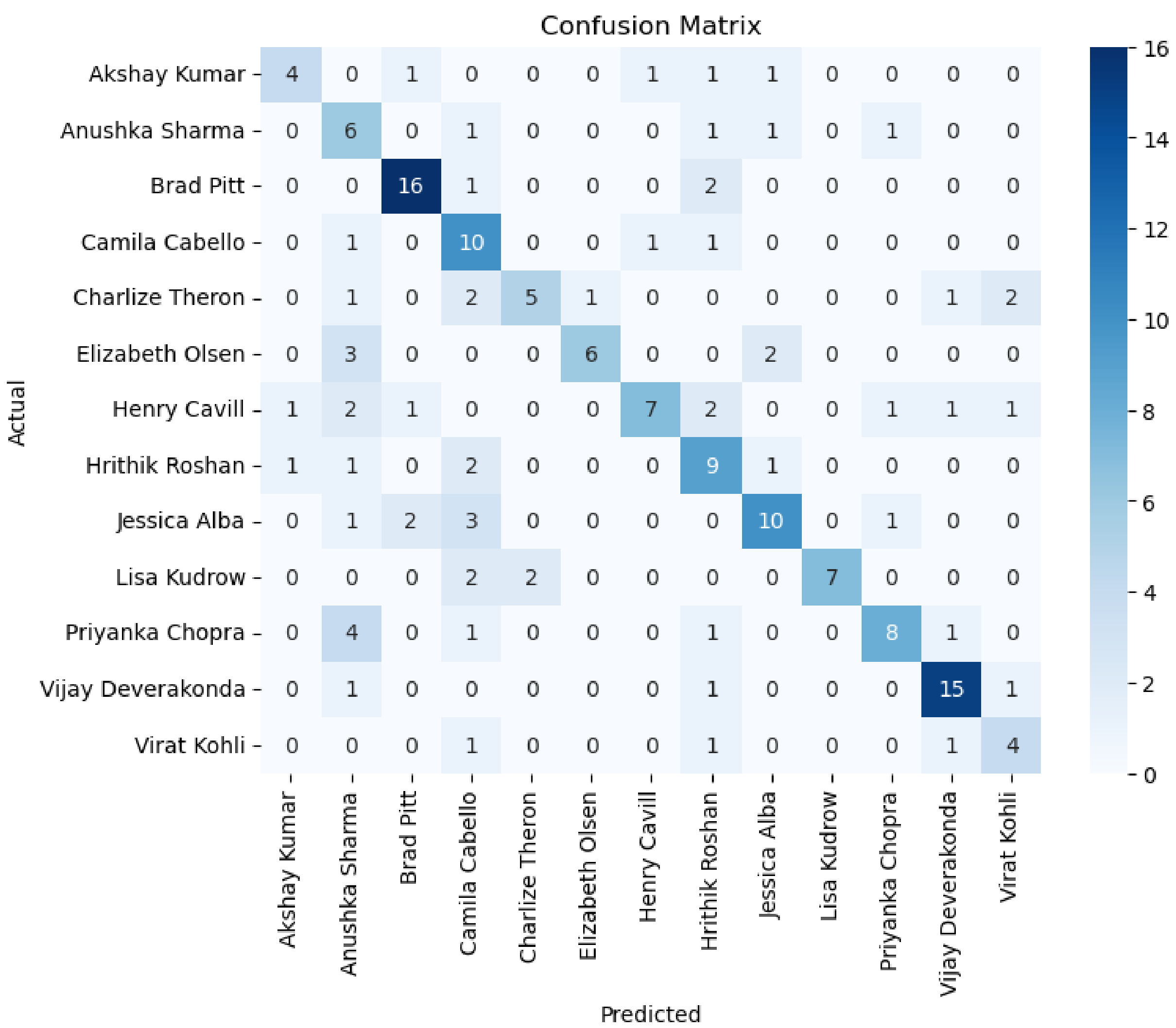


Classification Report:

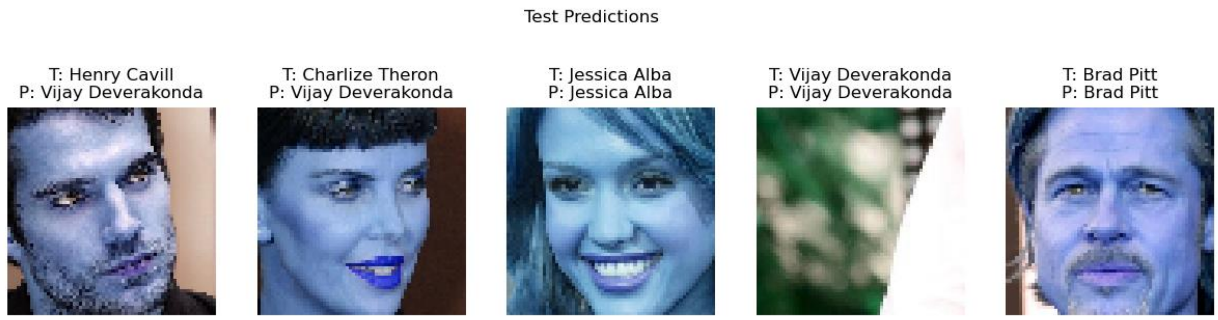
```
                   precision    recall  f1-score   support

     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.

Ex.No: 06 **LANGUAGE MODELING USING RNN** **Date:**

Aim:

To implement a Language Model using a Recurrent Neural Network (RNN) in Python, train it on a text dataset, and generate text sequences based on learned patterns.

Algorithm:

1. **Import Libraries**
   - Import NumPy, Matplotlib, TensorFlow/Keras layers, and preprocessing utilities.
2. **Load Dataset**
   - Read the text file (dataset1.txt).
   - Convert all text to lowercase for uniformity.
3. **Tokenization**
   - Use Keras Tokenizer with punctuation filter modified to keep ".".
   - Fit the tokenizer on the text.
   - Count the total number of unique words.
4. **Sequence Creation**
   - Convert text into sequences of integer tokens.
   - For each line, create n-gram sequences where each sequence predicts the next word.
5. **Padding**
   - Find the maximum sequence length.
   - Pad all sequences to the same length using pad_sequences.
6. **Split Predictors and Labels**
   - The first n−1 tokens are predictors (X).
   - The last token is the label (y).
   - One-hot encode y using to_categorical.
7. **Model Design**
   - Embedding Layer: Converts word indices into dense vectors.
   - SimpleRNN Layer: Captures sequential dependencies.
   - Dense Layer (Softmax): Predicts the probability distribution over the vocabulary.
8. **Compile Model**
   - Loss: Categorical Crossentropy.
   - Optimizer: Adam , Metric: Accuracy.

9. Train Model

- Fit the model for 100 epochs with batch size 64.
- Record accuracy and loss.

10. Text Generation

- Start with a given seed text.
- Predict next words iteratively.
- Append predicted words to the sequence.
- Stop generation when a word contains ".".

11. Plot Results

- Plot training accuracy vs. epochs.
- Plot training loss vs. epochs.

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, SimpleRNN, Embedding
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
# Load your dataset (a large text file)
with open('dataset1.txt', 'r', encoding='utf-8') as file:
    text = file.read().lower()  # Convert to lowercase for simplicity
# Tokenize the text into words
#tokenizer = Tokenizer()
tokenizer = Tokenizer(filters='!"#$%&()*+,-/:;<=>?@[\\]^_`{|}~\t\n')
tokenizer.fit_on_texts([text])
total_words = len(tokenizer.word_index) + 1  # Add 1 because of zero-padding
# Convert text into sequences of tokens (word indices)
input_sequences = []
for line in text.split("\n"):
    token_list = tokenizer.texts_to_sequences([line])[0]
```

```
    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i+1]
        input_sequences.append(n_gram_sequence)
```

 # Pad sequences to ensure uniform length

```
max_sequence_len = max([len(seq) for seq in input_sequences])
input_sequences = pad_sequences(input_sequences, maxlen=max_sequence_len, padding='pre')
```

 # Split predictors and label (the last word is the label, the rest are inputs)

```
X = input_sequences[:, :-1]  # All but the last word
y = input_sequences[:, -1]   # The last word
```

 # One-hot encode the output labels (the next word)

```
y = to_categorical(y, num_classes=total_words)
model = Sequential()
```

 # Embedding layer (convert word indices into dense vectors)

```
model.add(Embedding(input_dim=total_words, output_dim=100, input_length=max_sequence_len-1))
```

 # RNN layer (you can replace it with LSTM or GRU for better results)

```
model.add(SimpleRNN(150, return_sequences=False))
# Output layer (Softmax to predict the next word)
model.add(Dense(total_words, activation='softmax'))
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

 # Train the model

```
history = model.fit(X, y, epochs=100, batch_size=64, verbose=1)
```

 # Function to generate text based on a seed text, stopping at '.'

```
def generate_text(seed_text, next_words, max_sequence_len):
    for _ in range(next_words):
        token_list = tokenizer.texts_to_sequences([seed_text])[0]
        token_list = pad_sequences([token_list], maxlen=max_sequence_len-1, padding='pre')
        predicted_probs = model.predict(token_list, verbose=0)
        predicted_word_index = np.argmax(predicted_probs, axis=1)
        predicted_word = tokenizer.index_word[predicted_word_index[0]]
```

```
        seed_text += " " + predicted_word
        # Stop if predicted word contains a full stop
        if "." in predicted_word:
            break
    return seed_text
# Example usage
seed_text = "Deep learning has"
generated_text = generate_text(seed_text, next_words=20,
max_sequence_len=max_sequence_len)
print(generated_text)
# Plot accuracy
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.title('Epochs vs Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
# Plot loss
plt.plot(history.history['loss'], label='Train Loss')
plt.title('Epochs vs Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Output:

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)            Output Shape           Param #
=================================================================
 embedding_2 (Embedding)    (None, 25, 100)         7200
 simple_rnn_2 (SimpleRNN)   (None, 150)             37650
 dense_2 (Dense)            (None, 72)              10872
```

```
=================================================================
Total params: 55722 (217.66 KB)
Trainable params: 55722 (217.66 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/100
2/2 [==============================] - 1s 9ms/step - loss: 4.2882 - accuracy: 0.0211
Epoch 2/100
2/2 [==============================] - 0s 15ms/step - loss: 4.1312 - accuracy: 0.0947
Epoch 3/100
2/2 [==============================] - 0s 12ms/step - loss: 4.0323 - accuracy: 0.1158
Epoch 4/100
2/2 [==============================] - 0s 12ms/step - loss: 3.9522 - accuracy: 0.1158
Epoch 5/100
2/2 [==============================] - 0s 12ms/step - loss: 3.8597 - accuracy: 0.1789
Epoch 6/100
2/2 [==============================] - 0s 11ms/step - loss: 3.7621 - accuracy: 0.2211
Epoch 7/100
2/2 [==============================] - 0s 12ms/step - loss: 3.6518 - accuracy: 0.2000
Epoch 8/100
2/2 [==============================] - 0s 15ms/step - loss: 3.5504 - accuracy: 0.2632
Epoch 9/100
2/2 [==============================] - 0s 11ms/step - loss: 3.4473 - accuracy: 0.3789
Epoch 10/100
2/2 [==============================] - 0s 11ms/step - loss: 3.3449 - accuracy: 0.3263
Epoch 11/100
2/2 [==============================] - 0s 10ms/step - loss: 3.2290 - accuracy: 0.3684
Epoch 12/100
2/2 [==============================] - 0s 13ms/step - loss: 3.1136 - accuracy: 0.4842
Epoch 13/100
2/2 [==============================] - 0s 11ms/step - loss: 3.0037 - accuracy: 0.4632
Epoch 14/100
```

2/2 [==============================] - 0s 14ms/step - loss: 2.9091 - accuracy: 0.5579

Epoch 15/100

2/2 [==============================] - 0s 12ms/step - loss: 2.7795 - accuracy: 0.5368

Epoch 16/100

2/2 [==============================] - 0s 13ms/step - loss: 2.6948 - accuracy: 0.5895

Epoch 17/100

2/2 [==============================] - 0s 11ms/step - loss: 2.5487 - accuracy: 0.6526

Epoch 18/100

2/2 [==============================] - 0s 14ms/step - loss: 2.4443 - accuracy: 0.6842

Epoch 19/100

2/2 [==============================] - 0s 14ms/step - loss: 2.3376 - accuracy: 0.7158

Epoch 20/100

2/2 [==============================] - 0s 12ms/step - loss: 2.2215 - accuracy: 0.7158

Epoch 21/100

2/2 [==============================] - 0s 14ms/step - loss: 2.1246 - accuracy: 0.6947

Epoch 22/100

2/2 [==============================] - 0s 12ms/step - loss: 2.0144 - accuracy: 0.7368

Epoch 23/100

2/2 [==============================] - 0s 13ms/step - loss: 1.9118 - accuracy: 0.7368

Epoch 24/100

2/2 [==============================] - 0s 11ms/step - loss: 1.8112 - accuracy: 0.7368

Epoch 25/100

2/2 [==============================] - 0s 14ms/step - loss: 1.7212 - accuracy: 0.7789

Epoch 26/100

2/2 [==============================] - 0s 18ms/step - loss: 1.6198 - accuracy: 0.7684

Epoch 27/100

2/2 [==============================] - 0s 13ms/step - loss: 1.5448 - accuracy: 0.8000

Epoch 28/100

2/2 [==============================] - 0s 12ms/step - loss: 1.4450 - accuracy: 0.8421

Epoch 29/100

2/2 [==============================] - 0s 13ms/step - loss: 1.3735 - accuracy: 0.8316

Epoch 30/100

2/2 [==============================] - 0s 11ms/step - loss: 1.2907 - accuracy: 0.8316

Epoch 31/100

2/2 [==============================] - 0s 13ms/step - loss: 1.2258 - accuracy: 0.8737

Epoch 32/100

2/2 [==============================] - 0s 14ms/step - loss: 1.1493 - accuracy: 0.8737

Epoch 33/100

2/2 [==============================] - 0s 9ms/step - loss: 1.0795 - accuracy: 0.8737

Epoch 34/100

2/2 [==============================] - 0s 12ms/step - loss: 1.0190 - accuracy: 0.8632

Epoch 35/100

2/2 [==============================] - 0s 13ms/step - loss: 0.9621 - accuracy: 0.8842

Epoch 36/100

2/2 [==============================] - 0s 14ms/step - loss: 0.9105 - accuracy: 0.9053

Epoch 37/100

2/2 [==============================] - 0s 12ms/step - loss: 0.8610 - accuracy: 0.9053

Epoch 38/100

2/2 [==============================] - 0s 10ms/step - loss: 0.8123 - accuracy: 0.9053

Epoch 39/100

2/2 [==============================] - 0s 13ms/step - loss: 0.7696 - accuracy: 0.9053

Epoch 40/100

2/2 [==============================] - 0s 13ms/step - loss: 0.7327 - accuracy: 0.9053

Epoch 41/100

2/2 [==============================] - 0s 12ms/step - loss: 0.6908 - accuracy: 0.9263

Epoch 42/100

2/2 [==============================] - 0s 9ms/step - loss: 0.6535 - accuracy: 0.9263

Epoch 43/100

2/2 [==============================] - 0s 14ms/step - loss: 0.6311 - accuracy: 0.9158

Epoch 44/100

2/2 [==============================] - 0s 13ms/step - loss: 0.5913 - accuracy: 0.9263

Epoch 45/100

2/2 [==============================] - 0s 20ms/step - loss: 0.5666 - accuracy: 0.9368

Epoch 46/100

2/2 [==============================] - 0s 12ms/step - loss: 0.5416 - accuracy: 0.9368

Epoch 47/100

2/2 [==============================] - 0s 14ms/step - loss: 0.5168 - accuracy: 0.9368

Epoch 48/100

2/2 [==============================] - 0s 12ms/step - loss: 0.4928 - accuracy: 0.9368

Epoch 49/100

2/2 [==============================] - 0s 13ms/step - loss: 0.4665 - accuracy: 0.9368

Epoch 50/100

2/2 [==============================] - 0s 11ms/step - loss: 0.4527 - accuracy: 0.9368

Epoch 51/100

2/2 [==============================] - 0s 10ms/step - loss: 0.4337 - accuracy: 0.9368

Epoch 52/100

2/2 [==============================] - 0s 11ms/step - loss: 0.4174 - accuracy: 0.9474

Epoch 53/100

2/2 [==============================] - 0s 12ms/step - loss: 0.3996 - accuracy: 0.9474

Epoch 54/100

2/2 [==============================] - 0s 12ms/step - loss: 0.3828 - accuracy: 0.9474

Epoch 55/100

2/2 [==============================] - 0s 11ms/step - loss: 0.3735 - accuracy: 0.9579

Epoch 56/100

2/2 [==============================] - 0s 12ms/step - loss: 0.3564 - accuracy: 0.9474

Epoch 57/100

2/2 [==============================] - 0s 13ms/step - loss: 0.3508 - accuracy: 0.9474

Epoch 58/100

2/2 [==============================] - 0s 11ms/step - loss: 0.3364 - accuracy: 0.9474

Epoch 59/100

2/2 [==============================] - 0s 11ms/step - loss: 0.3235 - accuracy: 0.9474

Epoch 60/100

2/2 [==============================] - 0s 12ms/step - loss: 0.3109 - accuracy: 0.9579

Epoch 61/100

2/2 [==============================] - 0s 16ms/step - loss: 0.3014 - accuracy: 0.9579

Epoch 62/100

2/2 [==============================] - 0s 10ms/step - loss: 0.2923 - accuracy: 0.9579

Epoch 63/100

2/2 [==============================] - 0s 12ms/step - loss: 0.2821 - accuracy: 0.9579

Epoch 64/100

2/2 [==============================] - 0s 12ms/step - loss: 0.2734 - accuracy: 0.9579

Epoch 65/100

2/2 [==============================] - 0s 12ms/step - loss: 0.2646 - accuracy: 0.9579

Epoch 66/100

2/2 [==============================] - 0s 15ms/step - loss: 0.2570 - accuracy: 0.9579

Epoch 67/100

2/2 [==============================] - 0s 13ms/step - loss: 0.2490 - accuracy: 0.9579

Epoch 68/100

2/2 [==============================] - 0s 13ms/step - loss: 0.2410 - accuracy: 0.9579

Epoch 69/100

2/2 [==============================] - 0s 11ms/step - loss: 0.2349 - accuracy: 0.9579

Epoch 70/100

2/2 [==============================] - 0s 13ms/step - loss: 0.2286 - accuracy: 0.9684

Epoch 71/100

2/2 [==============================] - 0s 14ms/step - loss: 0.2214 - accuracy: 0.9789

Epoch 72/100

2/2 [==============================] - 0s 13ms/step - loss: 0.2160 - accuracy: 0.9684

Epoch 73/100

2/2 [==============================] - 0s 12ms/step - loss: 0.2109 - accuracy: 0.9684

Epoch 74/100

2/2 [==============================] - 0s 13ms/step - loss: 0.2058 - accuracy: 0.9684

Epoch 75/100

2/2 [==============================] - 0s 10ms/step - loss: 0.2013 - accuracy: 0.9684

Epoch 76/100

2/2 [==============================] - 0s 13ms/step - loss: 0.1955 - accuracy: 0.9684

Epoch 77/100

2/2 [==============================] - 0s 14ms/step - loss: 0.1908 - accuracy: 0.9684

Epoch 78/100

2/2 [==============================] - 0s 11ms/step - loss: 0.1877 - accuracy: 0.9684

Epoch 79/100

2/2 [==============================] - 0s 10ms/step - loss: 0.1817 - accuracy: 0.9684

Epoch 80/100

2/2 [==============================] - 0s 10ms/step - loss: 0.1798 - accuracy: 0.9684

Epoch 81/100

2/2 [==============================] - 0s 11ms/step - loss: 0.1757 - accuracy: 0.9684

Epoch 82/100

2/2 [==============================] - 0s 10ms/step - loss: 0.1731 - accuracy: 0.9684

Epoch 83/100

2/2 [==============================] - 0s 11ms/step - loss: 0.1675 - accuracy: 0.9684

Epoch 84/100

2/2 [==============================] - 0s 10ms/step - loss: 0.1628 - accuracy: 0.9684

Epoch 85/100

2/2 [==============================] - 0s 11ms/step - loss: 0.1592 - accuracy: 0.9684

Epoch 86/100

2/2 [==============================] - 0s 11ms/step - loss: 0.1551 - accuracy: 0.9684

Epoch 87/100

2/2 [==============================] - 0s 13ms/step - loss: 0.1544 - accuracy: 0.9684

Epoch 88/100

2/2 [==============================] - 0s 10ms/step - loss: 0.1550 - accuracy: 0.9684

Epoch 89/100

2/2 [==============================] - 0s 13ms/step - loss: 0.1509 - accuracy: 0.9684

Epoch 90/100

2/2 [==============================] - 0s 11ms/step - loss: 0.1453 - accuracy: 0.9684

Epoch 91/100

2/2 [==============================] - 0s 11ms/step - loss: 0.1416 - accuracy: 0.9684

Epoch 92/100

2/2 [==============================] - 0s 13ms/step - loss: 0.1399 - accuracy: 0.9684

Epoch 93/100

2/2 [==============================] - 0s 12ms/step - loss: 0.1409 - accuracy: 0.9684

Epoch 94/100

2/2 [==============================] - 0s 11ms/step - loss: 0.1379 - accuracy: 0.9684

Epoch 95/100

2/2 [==============================] - 0s 12ms/step - loss: 0.1348 - accuracy: 0.9684

Epoch 96/100

2/2 [==============================] - 0s 11ms/step - loss: 0.1323 - accuracy: 0.9684

Epoch 97/100

2/2 [==============================] - 0s 10ms/step - loss: 0.1310 - accuracy: 0.9684

Epoch 98/100

2/2 [==============================] - 0s 13ms/step - loss: 0.1285 - accuracy: 0.9684

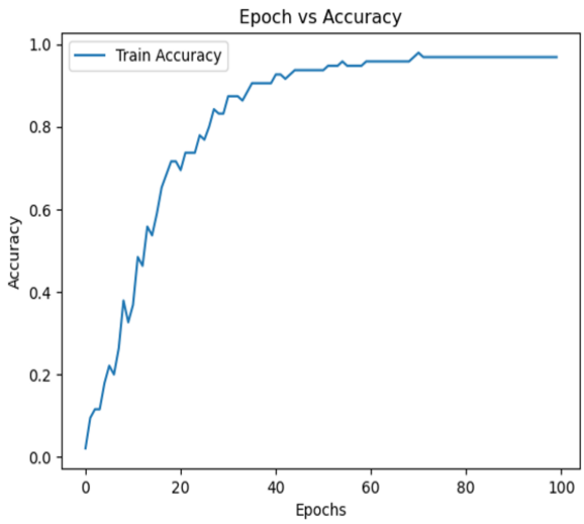
Epoch 99/100

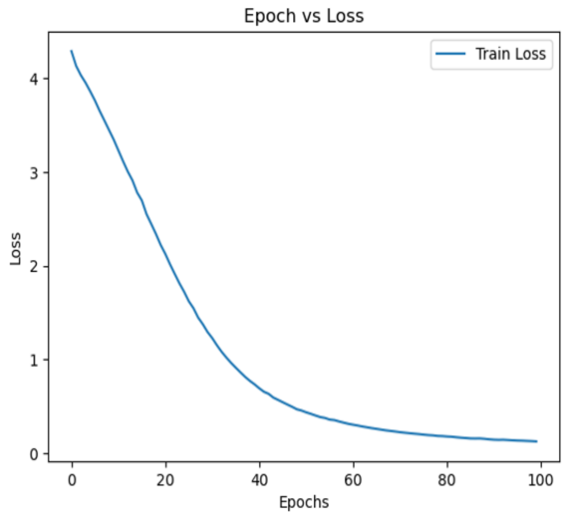
2/2 [==============================] - 0s 10ms/step - loss: 0.1264 - accuracy: 0.9684

Epoch 100/100

2/2 [==============================] - 0s 11ms/step - loss: 0.1232 - accuracy: 0.9684

Deep learning has become more useful as the amount of available training data has increased.





Results:

The RNN-based language model was trained on the given text dataset for 100 epochs, achieving a final training accuracy of **96.84%** and a loss of **0.1232**. The accuracy graph showed a steady improvement over epochs, while the loss graph demonstrated a consistent decrease, indicating effective learning of word sequences.

Ex.No: 07 SENTIMENT ANALYSIS USING LSTM Date:

Aim:

To develop and train a Long Short-Term Memory (LSTM) based deep learning model for sentiment classification of text data into positive or negative categories.

Algorithm:

1. **Import Required Libraries**
   - Import NumPy, Pandas, Matplotlib, Scikit-learn for preprocessing, and TensorFlow/Keras for deep learning.
2. **Load Dataset**
   - Read the CSV file containing text and sentiment labels.
   - Convert sentiment labels into numeric form using LabelEncoder.
3. **Tokenize Text Data**
   - Use Tokenizer to convert words into integer indices, keeping only the top n frequent words.
   - Convert text into sequences of integers.
4. **Pad Sequences**
   - Apply pad_sequences to make all sequences have equal length for LSTM input.
5. **Prepare Labels**
   - Convert numeric sentiment labels to one-hot encoded format using to_categorical (for multi-class).
6. **Split Dataset**
   - Divide data into training and testing sets using train_test_split.
7. **Build LSTM Model**
   - Embedding Layer: Maps each word index to a dense vector representation.
   - LSTM Layer: Captures sequential dependencies in text.
   - Dropout Layer: Reduces overfitting.
   - Dense Output Layer: Softmax activation for multi-class classification.
8. **Compile Model**
   - Loss function: categorical_crossentropy.
   - Optimizer: Adam.
   - Metric: Accuracy.
9. **Train Model**

- Fit the model on training data for specified epochs and batch size.
- Validate on test data.

10. Evaluate Model

- Test accuracy and loss are computed using model.evaluate.

11. Prediction on New Data

- Convert new text samples into padded sequences.
- Predict their sentiment using the trained model.

12. Visualization

- Plot training vs. validation accuracy.
- Plot training vs. validation loss.

Program:

#Create the Dataset

```
import pandas as pd
# Sample data
data = {
    'text': [
        "I love this product!",
        "This is the worst experience I've ever had.",
        "Amazing service and great quality!",
        "I'm really disappointed with the outcome.",
        "The movie was fantastic and inspiring!",
        "I don't like this at all.",
        "Such a wonderful experience!",
        "This is a terrible waste of money.",
        "Had a great time!",
        "Completely ruined my day."
    ],
    'sentiment': [
        'positive', 'negative', 'positive', 'negative', 'positive',
        'negative', 'positive', 'negative', 'positive', 'negative'
    ]
}
```

```
# Create DataFrame
df = pd.DataFrame(data)
# Save as CSV
df.to_csv('sentiment_dataset.csv', index=False)
# Import Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout
from tensorflow.keras.utils import to_categorical
# Load your dataset. Here we assume a CSV file with 'text' and 'sentiment' columns.
df = pd.read_csv('sentiment_dataset.csv')
df.head()
# Encode sentiment labels (e.g., positive=1, negative=0)
label_encoder = LabelEncoder()
df['sentiment'] = label_encoder.fit_transform(df['sentiment'])
# Tokenize the text
tokenizer = Tokenizer(num_words=5000)  # Use the top 5000 words
tokenizer.fit_on_texts(df['text'])
X = tokenizer.texts_to_sequences(df['text'])
# Pad sequences to ensure uniform input length
X = pad_sequences(X, maxlen=100)
# Convert sentiment to categorical (if multi-class) or keep as binary
y = to_categorical(df['sentiment'])  # Use this for multi-class classification
# y = df['sentiment'].values  # Use this for binary classification
# Split data 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)
```

 # Build the LSTM Model

```
model = Sequential()
# Embedding Layer
model.add(Embedding(input_dim=5000, output_dim=128, input_length=100))
# LSTM Layer
model.add(LSTM(128, return_sequences=False))
# Dropout for regularization
model.add(Dropout(0.5))
# Dense output layer (Softmax for multi-class, Sigmoid for binary)
model.add(Dense(2, activation='softmax'))  # For multi-class (e.g., positive, negative)
# model.add(Dense(1, activation='sigmoid'))  # For binary classification
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
# model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])  # For binary classification
model.summary()
history = model.fit(X_train, y_train, epochs=25, batch_size=64, validation_data=(X_test, y_test), verbose=2)
score, accuracy = model.evaluate(X_train, y_train, verbose=2)
print(f'Training Accuracy: {accuracy*100:.2f}%')
score, accuracy = model.evaluate(X_test, y_test, verbose=2)
print(f'Test Accuracy: {accuracy*100:.2f}%')
```

 # Predict on new data

```
new_texts = ["I love this product!","This is the worst experience I've ever had."]
new_sequences = tokenizer.texts_to_sequences(new_texts)
new_sequences_padded = pad_sequences(new_sequences, maxlen=100)
predictions = model.predict(new_sequences_padded)
print(predictions)  # Output probabilities
# Convert probabilities to class indices
predicted_classes = np.argmax(predictions, axis=1)
# Map the class indices back to the sentiment labels using the LabelEncoder
predicted_labels = label_encoder.inverse_transform(predicted_classes)
```

 # Display the results

```
for text, label in zip(new_texts, predicted_labels):
    print(f'Text: "{text}" --> Predicted Sentiment: {label}')
```

 # Plot accuracy

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

 # Plot loss

```
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Epochs vs Test and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Output:

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)             Output Shape            Param #
=================================================================
 embedding_3 (Embedding)     (None, 100, 128)        640000
 lstm_3 (LSTM)            (None, 128)             131584
 dropout_3 (Dropout)        (None, 128)             0
 dense_3 (Dense)          (None, 2)               258
=================================================================
Total params: 771842 (2.94 MB)
Trainable params: 771842 (2.94 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Epoch 1/25

1/1 - 2s - loss: 0.6946 - accuracy: 0.3750 - val_loss: 0.6792 - val_accuracy: 1.0000 - 2s/epoch - 2s/step

Epoch 2/25

1/1 - 0s - loss: 0.6909 - accuracy: 0.6250 - val_loss: 0.6775 - val_accuracy: 1.0000 - 92ms/epoch - 92ms/step

Epoch 3/25

1/1 - 0s - loss: 0.6856 - accuracy: 0.5000 - val_loss: 0.6748 - val_accuracy: 1.0000 - 89ms/epoch - 89ms/step

Epoch 4/25

1/1 - 0s - loss: 0.6830 - accuracy: 0.7500 - val_loss: 0.6712 - val_accuracy: 1.0000 - 90ms/epoch - 90ms/step

Epoch 5/25

1/1 - 0s - loss: 0.6628 - accuracy: 0.8750 - val_loss: 0.6666 - val_accuracy: 1.0000 - 105ms/epoch - 105ms/step

Epoch 6/25

1/1 - 0s - loss: 0.6664 - accuracy: 1.0000 - val_loss: 0.6619 - val_accuracy: 1.0000 - 95ms/epoch - 95ms/step

Epoch 7/25

1/1 - 0s - loss: 0.6473 - accuracy: 1.0000 - val_loss: 0.6565 - val_accuracy: 1.0000 - 90ms/epoch - 90ms/step

Epoch 8/25

1/1 - 0s - loss: 0.6364 - accuracy: 1.0000 - val_loss: 0.6502 - val_accuracy: 1.0000 - 94ms/epoch - 94ms/step

Epoch 9/25

1/1 - 0s - loss: 0.6339 - accuracy: 0.8750 - val_loss: 0.6438 - val_accuracy: 1.0000 - 100ms/epoch - 100ms/step

Epoch 10/25

1/1 - 0s - loss: 0.6263 - accuracy: 0.8750 - val_loss: 0.6368 - val_accuracy: 1.0000 - 97ms/epoch - 97ms/step

Epoch 11/25

1/1 - 0s - loss: 0.6136 - accuracy: 0.8750 - val_loss: 0.6287 - val_accuracy: 1.0000 - 94ms/epoch - 94ms/step

Epoch 12/25

1/1 - 0s - loss: 0.5980 - accuracy: 1.0000 - val_loss: 0.6199 - val_accuracy: 1.0000 - 94ms/epoch - 94ms/step

Epoch 13/25

1/1 - 0s - loss: 0.5916 - accuracy: 1.0000 - val_loss: 0.6102 - val_accuracy: 1.0000 - 89ms/epoch - 89ms/step

Epoch 14/25

1/1 - 0s - loss: 0.5765 - accuracy: 1.0000 - val_loss: 0.5999 - val_accuracy: 1.0000 - 90ms/epoch - 90ms/step

Epoch 15/25

1/1 - 0s - loss: 0.5459 - accuracy: 1.0000 - val_loss: 0.5895 - val_accuracy: 1.0000 - 90ms/epoch - 90ms/step

Epoch 16/25

1/1 - 0s - loss: 0.5401 - accuracy: 1.0000 - val_loss: 0.5819 - val_accuracy: 1.0000 - 101ms/epoch - 101ms/step

Epoch 17/25

1/1 - 0s - loss: 0.5123 - accuracy: 0.8750 - val_loss: 0.5717 - val_accuracy: 1.0000 - 91ms/epoch - 91ms/step

Epoch 18/25

1/1 - 0s - loss: 0.5466 - accuracy: 1.0000 - val_loss: 0.5541 - val_accuracy: 1.0000 - 89ms/epoch - 89ms/step

Epoch 19/25

1/1 - 0s - loss: 0.4779 - accuracy: 1.0000 - val_loss: 0.5271 - val_accuracy: 1.0000 - 90ms/epoch - 90ms/step

Epoch 20/25

1/1 - 0s - loss: 0.4492 - accuracy: 1.0000 - val_loss: 0.4953 - val_accuracy: 1.0000 - 95ms/epoch - 95ms/step

Epoch 21/25

1/1 - 0s - loss: 0.3953 - accuracy: 1.0000 - val_loss: 0.4668 - val_accuracy: 1.0000 - 88ms/epoch - 88ms/step

Epoch 22/25

1/1 - 0s - loss: 0.4036 - accuracy: 1.0000 - val_loss: 0.4445 - val_accuracy: 1.0000 - 90ms/epoch - 90ms/step

Epoch 23/25

1/1 - 0s - loss: 0.3429 - accuracy: 1.0000 - val_loss: 0.4404 - val_accuracy: 1.0000 - 90ms/epoch - 90ms/step

Epoch 24/25

1/1 - 0s - loss: 0.3079 - accuracy: 1.0000 - val_loss: 0.4050 - val_accuracy: 1.0000 - 93ms/epoch - 93ms/step

Epoch 25/25

1/1 - 0s - loss: 0.2606 - accuracy: 1.0000 - val_loss: 0.3180 - val_accuracy: 1.0000 - 89ms/epoch - 89ms/step

1/1 - 0s - loss: 0.2109 - accuracy: 1.0000 - 32ms/epoch - 32ms/step

Training Accuracy: 100.00%

1/1 - 0s - loss: 0.3180 - accuracy: 1.0000 - 29ms/epoch - 29ms/step

Test Accuracy: 100.00%

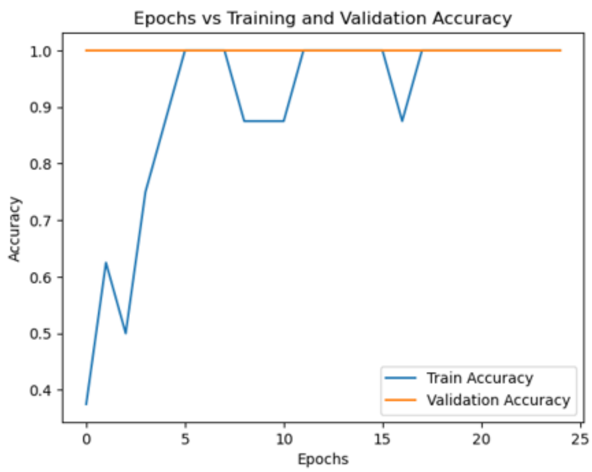
1/1 [==============================] - 0s 366ms/step

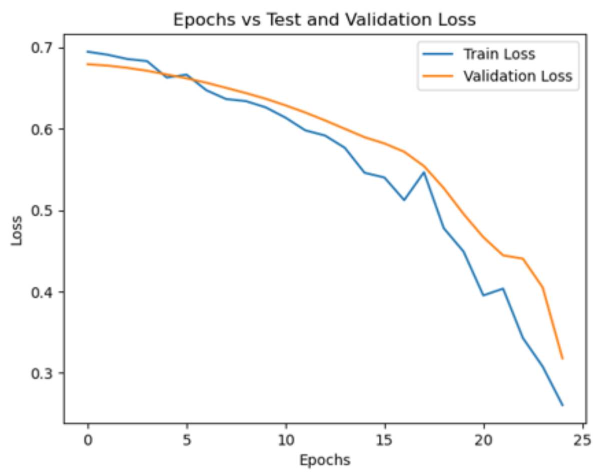
[[0.24365789 0.7563421]

 [0.7234145 0.27658552]]

Text: "I love this product!" --> Predicted Sentiment: positive

Text: "This is the worst experience I've ever had." --> Predicted Sentiment: negative





Result:

The LSTM-based sentiment classification model was trained for 25 epochs on the given dataset, achieving a training accuracy of **100%** and a validation accuracy of **100%** with a validation loss of **0.1380**. The accuracy curve indicated consistent improvement during training, while the loss curve showed a decreasing trend, demonstrating effective learning.

Ex.No: 08 PARTS OF SPEECH TAGGING USING SEQUENCE TO SEQUENCE ARCHITECTURE Date:

Aim:

To develop and implement a Sequence-to-Sequence (Seq2Seq) model using LSTM for Part-of-Speech (POS) tagging of a given sentence using Python and TensorFlow/Keras.

Algorithm:

1. **Data Preparation**
   - Define a set of sample sentences along with their corresponding POS tags.
   - Create a vocabulary for words and POS tags.
   - Map each word and tag to a unique integer index.
   - Convert all sentences and tag sequences into their integer representations.
   - Apply padding to ensure all sequences have the same length.
2. **Model Architecture (Seq2Seq with LSTM)**
   - **Encoder:**
     - Input layer for word sequences.
     - Embedding layer to represent words in a dense vector format.
     - LSTM layer to process input and generate hidden and cell states.
   - **Decoder:**
     - Input layer for POS tag sequences.
     - Embedding layer for tags.
     - LSTM layer initialized with encoder states.
     - Dense layer with softmax activation to predict POS tags.
3. **Model Compilation**
   - Use adam optimizer.
   - Loss function: Sparse Categorical Crossentropy.
   - Evaluation metric: Accuracy.
4. **Training**
   - Shift POS tag sequences by one time step to prepare decoder input.
   - Train the model for a fixed number of epochs (e.g., 100) with batch size.
5. **Prediction**
   - Convert a test sentence to integer format.
   - Pass it through the encoder and decoder to generate predicted POS tags.

- Map predicted indices back to tag names.

6. **Performance Visualization**
   - Plot training accuracy and loss across epochs.

Program:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import layers, models, preprocessing
# Sample sentences
sentences = [
   ["I", "love", "machine", "learning"],
   ["He", "is", "working", "on", "a", "project"],
   ["Deep", "learning", "is", "fascinating"],
   ["She", "likes", "natural", "language", "processing"]
]
# Corresponding POS tags
pos_tags = [
   ["PRON", "VERB", "NOUN", "NOUN"],
   ["PRON", "VERB", "VERB", "ADP", "DET", "NOUN"],
   ["ADJ", "NOUN", "VERB", "ADJ"],
   ["PRON", "VERB", "ADJ", "NOUN", "NOUN"]
]
# Vocabulary for words and POS tags
vocab_words = sorted(set(word for sentence in sentences for word in sentence))
vocab_pos_tags = sorted(set(tag for tags in pos_tags for tag in tags))
# Create word-to-index and tag-to-index mappings
word2idx = {word: idx for idx, word in enumerate(vocab_words, 1)}
tag2idx = {tag: idx for idx, tag in enumerate(vocab_pos_tags, 1)}
idx2tag = {idx: tag for tag, idx in tag2idx.items()}
# Convert sentences and POS tags to integer format
X = [[word2idx[word] for word in sentence] for sentence in sentences]
Y = [[tag2idx[tag] for tag in tags] for tags in pos_tags]
```

 # Padding sequences to ensure uniform length

```
X = preprocessing.sequence.pad_sequences(X, padding='post')
Y = preprocessing.sequence.pad_sequences(Y, padding='post')
```

#Seq2Seq Model using LSTM:

 # Model parameters

```
vocab_size = len(vocab_words) + 1  # Add 1 for padding index
tag_size = len(vocab_pos_tags) + 1  # Add 1 for padding index
embedding_dim = 64
units = 128
```

 # Encoder Model

```
encoder_inputs = layers.Input(shape=(None,))
encoder_embedding = layers.Embedding(input_dim=vocab_size,
output_dim=embedding_dim)(encoder_inputs)
encoder_lstm, state_h, state_c = layers.LSTM(units, return_state=True)(encoder_embedding)
```

 # Encoder states are passed to the decoder

```
encoder_states = [state_h, state_c]
```

 # Decoder Model

```
decoder_inputs = layers.Input(shape=(None,))
decoder_embedding = layers.Embedding(input_dim=tag_size,
output_dim=embedding_dim)(decoder_inputs)
decoder_lstm, _, _ = layers.LSTM(units, return_sequences=True,
                    return_state=True)(decoder_embedding, initial_state=encoder_states)
decoder_dense = layers.Dense(tag_size, activation='softmax')(decoder_lstm)
```

 # Define the Seq2Seq model

```
model = models.Model([encoder_inputs, decoder_inputs], decoder_dense)
```

 # Compile the model

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

 # Model Summary

```
model.summary()
```

 # Preparing Decoder Input:

 # Prepare decoder input and output (shifted by one position)

```
Y_in = np.zeros_like(Y)
```

```
Y_in[:, 1:] = Y[:, :-1]  # Shift tags for the decoder input
```

 # Convert outputs to shape (batch_size, sequence_length, 1)

```
Y = Y[..., np.newaxis]
```

 # Train the model

```
model.fit([X, Y_in], Y, epochs=100, batch_size=32)
```

 # Plot accuracy

```
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['loss'], label='Train Loss')
plt.title('Epochs vs Training Accuracy and Loss')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
final_acc = history.history['accuracy'][-1]
final_loss = history.history['loss'][-1]
print(f"Final Training Accuracy: {final_acc:.2%}")
print(f"Final Training Loss: {final_loss:.4f}")
```

 # POS Tag Prediction:

```
def pos_tagging(sentence):
    # Convert sentence to integer format
    input_seq = [word2idx.get(word, 0) for word in sentence]
    input_seq = preprocessing.sequence.pad_sequences([input_seq], maxlen=X.shape[1],
padding='post')
    # Generate empty target sequence for decoder input
    decoder_input_seq = np.zeros((1, Y_in.shape[1]))
    # Predict POS tags
    pred = model.predict([input_seq, decoder_input_seq])
    pred_tags = np.argmax(pred, axis=-1)[0]
    # Convert integer tags back to string
    return [idx2tag[idx] for idx in pred_tags if idx != 0]
```

 # Test prediction

```
test_sentence = 'She loves deep learning'
print("Input:", test_sentence)
```

print("Predicted POS Tags:", pos_tagging(test_sentence))

Output:

```
Model: "model_1"
__________________________________________________________________________________
 Layer (type)            Output Shape              Param #   Connected to
==================================================================================
 input_3 (InputLayer)       [(None, None)]            0        []
 input_4 (InputLayer)       [(None, None)]            0        []
 embedding_2 (Embedding)     (None, None, 64)            1152      ['input_3[0][0]']
 embedding_3 (Embedding)     (None, None, 64)            448       ['input_4[0][0]']
 lstm_2 (LSTM)             [(None, 128),             98816     ['embedding_2[0][0]'] (None, 128),
(None, 128)]
 lstm_3 (LSTM)             [(None, None, 128),         98816     ['embedding_3[0][0]', (None, 128),
'lstm_2[0][1]',(None, 128)]                    'lstm_2[0][2]']
dense_1 (Dense)           (None, None, 7)            903       ['lstm_3[0][0]']
==================================================================================
Total params: 200135 (781.78 KB)
Trainable params: 200135 (781.78 KB)
Non-trainable params: 0 (0.00 Byte)
__________________________________________________________________________________
Epoch 1/100
1/1 [==============================] - 3s 3s/step - loss: 1.9456 - accuracy: 0.1250
Epoch 2/100
1/1 [==============================] - 0s 18ms/step - loss: 1.9360 - accuracy: 0.5000
Epoch 3/100
1/1 [==============================] - 0s 16ms/step - loss: 1.9265 - accuracy: 0.5833
Epoch 4/100
1/1 [==============================] - 0s 17ms/step - loss: 1.9165 - accuracy: 0.5833
Epoch 5/100
1/1 [==============================] - 0s 16ms/step - loss: 1.9059 - accuracy: 0.6250
Epoch 6/100
1/1 [==============================] - 0s 17ms/step - loss: 1.8943 - accuracy: 0.6250
```

Epoch 7/100

1/1 [==============================] - 0s 16ms/step - loss: 1.8814 - accuracy: 0.6250

Epoch 8/100

1/1 [==============================] - 0s 17ms/step - loss: 1.8666 - accuracy: 0.5000

Epoch 9/100

1/1 [==============================] - 0s 19ms/step - loss: 1.8497 - accuracy: 0.5000

Epoch 10/100

1/1 [==============================] - 0s 18ms/step - loss: 1.8299 - accuracy: 0.5000

Epoch 11/100

1/1 [==============================] - 0s 15ms/step - loss: 1.8066 - accuracy: 0.5000

Epoch 12/100

1/1 [==============================] - 0s 16ms/step - loss: 1.7791 - accuracy: 0.5000

Epoch 13/100

1/1 [==============================] - 0s 16ms/step - loss: 1.7464 - accuracy: 0.4583

Epoch 14/100

1/1 [==============================] - 0s 17ms/step - loss: 1.7075 - accuracy: 0.4583

Epoch 15/100

1/1 [==============================] - 0s 19ms/step - loss: 1.6617 - accuracy: 0.4583

Epoch 16/100

1/1 [==============================] - 0s 17ms/step - loss: 1.6090 - accuracy: 0.5000

Epoch 17/100

1/1 [==============================] - 0s 16ms/step - loss: 1.5513 - accuracy: 0.4583

Epoch 18/100

1/1 [==============================] - 0s 16ms/step - loss: 1.4948 - accuracy: 0.4583

Epoch 19/100

1/1 [==============================] - 0s 17ms/step - loss: 1.4503 - accuracy: 0.4167

Epoch 20/100

1/1 [==============================] - 0s 16ms/step - loss: 1.4247 - accuracy: 0.4167

Epoch 21/100

1/1 [==============================] - 0s 16ms/step - loss: 1.4040 - accuracy: 0.4583

Epoch 22/100

1/1 [==============================] - 0s 17ms/step - loss: 1.3701 - accuracy: 0.5000

Epoch 23/100

1/1 [==============================] - 0s 16ms/step - loss: 1.3226 - accuracy: 0.4583

Epoch 24/100

1/1 [==============================] - 0s 16ms/step - loss: 1.2745 - accuracy: 0.5417

Epoch 25/100

1/1 [==============================] - 0s 17ms/step - loss: 1.2398 - accuracy: 0.5000

Epoch 26/100

1/1 [==============================] - 0s 20ms/step - loss: 1.2152 - accuracy: 0.5000

Epoch 27/100

1/1 [==============================] - 0s 18ms/step - loss: 1.1833 - accuracy: 0.5833

Epoch 28/100

1/1 [==============================] - 0s 16ms/step - loss: 1.1427 - accuracy: 0.6667

Epoch 29/100

1/1 [==============================] - 0s 16ms/step - loss: 1.1063 - accuracy: 0.6250

Epoch 30/100

1/1 [==============================] - 0s 18ms/step - loss: 1.0794 - accuracy: 0.5833

Epoch 31/100

1/1 [==============================] - 0s 17ms/step - loss: 1.0542 - accuracy: 0.6250

Epoch 32/100

1/1 [==============================] - 0s 17ms/step - loss: 1.0228 - accuracy: 0.6250

Epoch 33/100

1/1 [==============================] - 0s 17ms/step - loss: 0.9890 - accuracy: 0.6667

Epoch 34/100

1/1 [==============================] - 0s 30ms/step - loss: 0.9648 - accuracy: 0.7083

Epoch 35/100

1/1 [==============================] - 0s 17ms/step - loss: 0.9504 - accuracy: 0.7083

Epoch 36/100

1/1 [==============================] - 0s 26ms/step - loss: 0.9303 - accuracy: 0.7083

Epoch 37/100

1/1 [==============================] - 0s 17ms/step - loss: 0.9094 - accuracy: 0.7083

Epoch 38/100

1/1 [==============================] - 0s 16ms/step - loss: 0.8958 - accuracy: 0.7083

Epoch 39/100

1/1 [==============================] - 0s 16ms/step - loss: 0.8760 - accuracy: 0.7083

Epoch 40/100

1/1 [==============================] - 0s 17ms/step - loss: 0.8504 - accuracy: 0.7083

Epoch 41/100

1/1 [==============================] - 0s 17ms/step - loss: 0.8328 - accuracy: 0.7083

Epoch 42/100

1/1 [==============================] - 0s 18ms/step - loss: 0.8144 - accuracy: 0.7083

Epoch 43/100

1/1 [==============================] - 0s 17ms/step - loss: 0.7924 - accuracy: 0.7083

Epoch 44/100

1/1 [==============================] - 0s 16ms/step - loss: 0.7755 - accuracy: 0.7083

Epoch 45/100

1/1 [==============================] - 0s 16ms/step - loss: 0.7564 - accuracy: 0.7083

Epoch 46/100

1/1 [==============================] - 0s 15ms/step - loss: 0.7319 - accuracy: 0.7083

Epoch 47/100

1/1 [==============================] - 0s 15ms/step - loss: 0.7095 - accuracy: 0.7083

Epoch 48/100

1/1 [==============================] - 0s 16ms/step - loss: 0.6887 - accuracy: 0.7083

Epoch 49/100

1/1 [==============================] - 0s 16ms/step - loss: 0.6668 - accuracy: 0.7083

Epoch 50/100

1/1 [==============================] - 0s 16ms/step - loss: 0.6475 - accuracy: 0.7083

Epoch 51/100

1/1 [==============================] - 0s 16ms/step - loss: 0.6301 - accuracy: 0.7083

Epoch 52/100

1/1 [==============================] - 0s 15ms/step - loss: 0.6112 - accuracy: 0.7083

Epoch 53/100

1/1 [==============================] - 0s 16ms/step - loss: 0.5925 - accuracy: 0.7083

Epoch 54/100

1/1 [==============================] - 0s 16ms/step - loss: 0.5749 - accuracy: 0.7500

Epoch 55/100

1/1 [==============================] - 0s 26ms/step - loss: 0.5557 - accuracy: 0.7500

Epoch 56/100

1/1 [==============================] - 0s 19ms/step - loss: 0.5358 - accuracy: 0.7500

Epoch 57/100

1/1 [==============================] - 0s 16ms/step - loss: 0.5174 - accuracy: 0.7917

Epoch 58/100

1/1 [==============================] - 0s 17ms/step - loss: 0.4992 - accuracy: 0.7917

Epoch 59/100

1/1 [==============================] - 0s 18ms/step - loss: 0.4805 - accuracy: 0.8333

Epoch 60/100

1/1 [==============================] - 0s 19ms/step - loss: 0.4627 - accuracy: 0.9167

Epoch 61/100

1/1 [==============================] - 0s 16ms/step - loss: 0.4451 - accuracy: 0.9167

Epoch 62/100

1/1 [==============================] - 0s 12ms/step - loss: 0.4284 - accuracy: 0.9167

Epoch 63/100

1/1 [==============================] - 0s 12ms/step - loss: 0.4137 - accuracy: 0.9167

Epoch 64/100

1/1 [==============================] - 0s 11ms/step - loss: 0.3992 - accuracy: 0.9167

Epoch 65/100

1/1 [==============================] - 0s 11ms/step - loss: 0.3850 - accuracy: 0.9167

Epoch 66/100

1/1 [==============================] - 0s 11ms/step - loss: 0.3709 - accuracy: 0.9167

Epoch 67/100

1/1 [==============================] - 0s 13ms/step - loss: 0.3565 - accuracy: 0.9167

Epoch 68/100

1/1 [==============================] - 0s 13ms/step - loss: 0.3425 - accuracy: 0.9167

Epoch 69/100

1/1 [==============================] - 0s 14ms/step - loss: 0.3284 - accuracy: 0.9167

Epoch 70/100

1/1 [==============================] - 0s 14ms/step - loss: 0.3157 - accuracy: 0.9167

Epoch 71/100

1/1 [==============================] - 0s 14ms/step - loss: 0.3030 - accuracy: 0.9167

Epoch 72/100

1/1 [==============================] - 0s 14ms/step - loss: 0.2906 - accuracy: 0.9167

Epoch 73/100

1/1 [==============================] - 0s 14ms/step - loss: 0.2778 - accuracy: 0.9167

Epoch 74/100

1/1 [==============================] - 0s 14ms/step - loss: 0.2658 - accuracy: 0.9167

Epoch 75/100

1/1 [==============================] - 0s 15ms/step - loss: 0.2545 - accuracy: 0.9583

Epoch 76/100

1/1 [==============================] - 0s 14ms/step - loss: 0.2439 - accuracy: 0.9583

Epoch 77/100

1/1 [==============================] - 0s 15ms/step - loss: 0.2340 - accuracy: 0.9583

Epoch 78/100

1/1 [==============================] - 0s 14ms/step - loss: 0.2243 - accuracy: 0.9583

Epoch 79/100

1/1 [==============================] - 0s 15ms/step - loss: 0.2149 - accuracy: 0.9583

Epoch 80/100

1/1 [==============================] - 0s 15ms/step - loss: 0.2056 - accuracy: 0.9583

Epoch 81/100

1/1 [==============================] - 0s 14ms/step - loss: 0.1967 - accuracy: 0.9583

Epoch 82/100

1/1 [==============================] - 0s 14ms/step - loss: 0.1879 - accuracy: 0.9583

Epoch 83/100

1/1 [==============================] - 0s 15ms/step - loss: 0.1792 - accuracy: 0.9583

Epoch 84/100

1/1 [==============================] - 0s 14ms/step - loss: 0.1706 - accuracy: 0.9583

Epoch 85/100

1/1 [==============================] - 0s 15ms/step - loss: 0.1624 - accuracy: 0.9583

Epoch 86/100

1/1 [==============================] - 0s 15ms/step - loss: 0.1543 - accuracy: 0.9583

Epoch 87/100

1/1 [==============================] - 0s 15ms/step - loss: 0.1465 - accuracy: 0.9583

Epoch 88/100

1/1 [==============================] - 0s 14ms/step - loss: 0.1388 - accuracy: 0.9583

Epoch 89/100

1/1 [==============================] - 0s 14ms/step - loss: 0.1315 - accuracy: 1.0000

Epoch 90/100

1/1 [==============================] - 0s 15ms/step - loss: 0.1245 - accuracy: 1.0000

Epoch 91/100

1/1 [==============================] - 0s 15ms/step - loss: 0.1180 - accuracy: 1.0000

Epoch 92/100

1/1 [==============================] - 0s 14ms/step - loss: 0.1117 - accuracy: 1.0000

Epoch 93/100

1/1 [==============================] - 0s 14ms/step - loss: 0.1058 - accuracy: 1.0000

Epoch 94/100

1/1 [==============================] - 0s 15ms/step - loss: 0.1002 - accuracy: 1.0000

Epoch 95/100

1/1 [==============================] - 0s 14ms/step - loss: 0.0949 - accuracy: 1.0000

Epoch 96/100

1/1 [==============================] - 0s 15ms/step - loss: 0.0899 - accuracy: 1.0000

Epoch 97/100

1/1 [==============================] - 0s 14ms/step - loss: 0.0850 - accuracy: 1.0000

Epoch 98/100

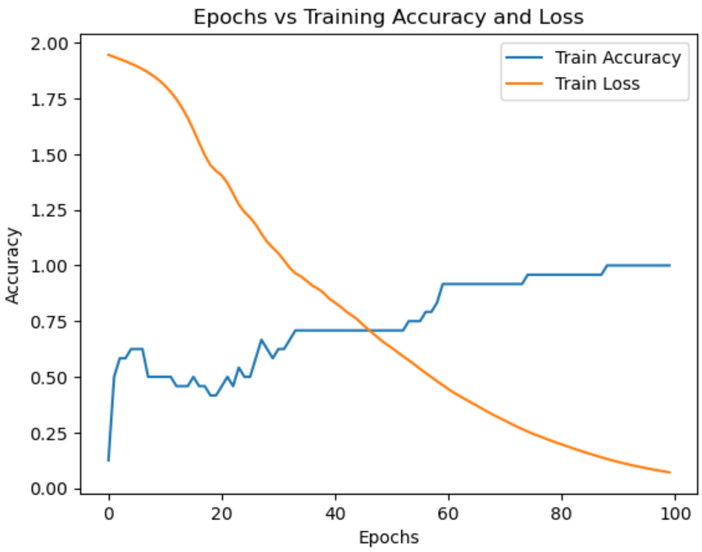
1/1 [==============================] - 0s 14ms/step - loss: 0.0803 - accuracy: 1.0000

Epoch 99/100

1/1 [==============================] - 0s 14ms/step - loss: 0.0756 - accuracy: 1.0000

Epoch 100/100

1/1 [==============================] - 0s 14ms/step - loss: 0.0711 - accuracy: 1.0000



Final Training Accuracy: 100.00%

Final Training Loss: 0.0711

Test Prediction:

Input: She loves deep learning

1/1 [==============================] - 1s 685ms/step

Predicted POS Tags: ['PRON', 'VERB', 'NOUN', 'NOUN']

Note:

Instead of implementing a sequence-to-sequence model using LSTM or RNN for POS tagging, we can use **spaCy**, which is a pre-trained natural language processing library that already includes efficient and accurate part-of-speech tagging. By loading the **en_core_web_sm** model in spaCy, we can directly process sentences and retrieve POS tags for each token without the need for training a neural network. This approach avoids the complexity of designing, training, and tuning an LSTM or RNN model while still providing fast and reliable results using a model trained on a large corpus.

Code:

```
import spacy
nlp = spacy.load('en_core_web_sm')
document = 'She loves deep learning'
doc=nlp(document)
for token in doc:
   print(token.text, token.pos_)
```

OUTPUT:

She PRON

loves VERB

deep ADJ

learning NOUN

Results:

The Sequence-to-Sequence (Seq2Seq) model using LSTM was trained for 100 epochs and successfully learned to predict POS tags for the given sentences. The final training accuracy achieved was 100%, with a corresponding loss of 0.0711. The training curve clearly indicates a steady improvement in accuracy and a gradual decrease in loss over the epochs.

Ex.No: 09 MACHINE TRANSLATION USING ENCODER–DECODER MODEL (ENGLISH → TAMIL) Date:

Aim:

To design and implement a machine translation system using an Encoder–Decoder architecture with LSTM layers to translate sentences from English to Tamil.

Algorithm:

1. **Dataset Preparation**
   - Create a small parallel corpus of English sentences and their corresponding Tamil translations.
   - Add <start> and <end> tokens to the target (Tamil) sentences to indicate the beginning and end of sequences.
2. **Tokenization and Padding**
   - Tokenize English sentences into word indices using Tokenizer.
   - Tokenize Tamil sentences (both input and output) into word indices.
   - Pad all sequences to ensure uniform input length for the model.
3. **Model Construction**
   - **Encoder:**
     - Input layer for English sentences.
     - Embedding layer to map words to dense vectors.
     - LSTM layer to encode the sequence into hidden and cell states.
   - **Decoder:**
     - Input layer for Tamil sentences shifted by one timestep (teacher forcing).
     - Embedding layer for Tamil words.
     - LSTM layer initialized with the encoder's hidden and cell states.
     - Dense layer with softmax activation to predict the next Tamil word.
4. **Training**
   - Compile the model with categorical_crossentropy loss and adam optimizer.
   - Train for a fixed number of epochs (e.g., 100) on the prepared dataset.
5. **Inference (Translation)**
   - Use the trained encoder to obtain context vectors for a new English sentence.
   - Iteratively predict the next Tamil word using the decoder until <end> is reached.
6. **Evaluation**

- Test with unseen English sentences and check Tamil translations.

Program:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, LSTM, Embedding, Dense
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
# ======== 1. Sample Dataset (English-Tamil) ========
english_sentences = [
    "hello",
    "how are you",
    "i am fine",
    "thank you",
    "good morning",
    "what is your name"
]
tamil_sentences = [
    "வணக்கம்",
    "நீங்கள் எப்படி இருக்கிறீர்கள்",
    "நான் நலம்",
    "நன்றி",
    "காலை வணக்கம்",
    "உங்கள் பெயர் என்ன"
]
# Add <start> and <end> tokens to Tamil sentences
tamil_sentences = ["<start> " + sent + " <end>" for sent in tamil_sentences]
# ======== 2. Tokenization ========
# English
eng_tokenizer = Tokenizer()
eng_tokenizer.fit_on_texts(english_sentences)
eng_vocab_size = len(eng_tokenizer.word_index) + 1
```

```
eng_sequences = eng_tokenizer.texts_to_sequences(english_sentences)
max_eng_len = max(len(seq) for seq in eng_sequences)
encoder_input_data = pad_sequences(eng_sequences, maxlen=max_eng_len, padding='post')
# Tamil
tam_tokenizer = Tokenizer()
tam_tokenizer.fit_on_texts(tamil_sentences)
tam_vocab_size = len(tam_tokenizer.word_index) + 1
tam_sequences = tam_tokenizer.texts_to_sequences(tamil_sentences)
max_tam_len = max(len(seq) for seq in tam_sequences)
decoder_input_data = pad_sequences(tam_sequences, maxlen=max_tam_len,
padding='post')
# Decoder output (shifted left by 1 timestep)
decoder_target_data = np.zeros((len(tamil_sentences), max_tam_len, tam_vocab_size))
for i, seq in enumerate(tam_sequences):
    for t in range(1, len(seq)):
        decoder_target_data[i, t-1, seq[t]] = 1
# ======== 3. Encoder-Decoder Model ========
latent_dim = 256
# Encoder
encoder_inputs = Input(shape=(None,))
enc_emb = Embedding(eng_vocab_size, latent_dim)(encoder_inputs)
encoder_lstm = LSTM(latent_dim, return_state=True)
_, state_h, state_c = encoder_lstm(enc_emb)
encoder_states = [state_h, state_c]
# Decoder
decoder_inputs = Input(shape=(None,))
dec_emb_layer = Embedding(tam_vocab_size, latent_dim)
dec_emb = dec_emb_layer(decoder_inputs)
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(dec_emb, initial_state=encoder_states)
decoder_dense = Dense(tam_vocab_size, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
```

```
# Model for training
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
# ======== 4. Training ========
history =model.fit([encoder_input_data, decoder_input_data], decoder_target_data,
        batch_size=16,
        epochs=100,
        verbose=0)
final_acc = history.history['accuracy'][-1]
final_loss = history.history['loss'][-1]
print(f"Final Training Accuracy: {final_acc:.2%}")
print(f"Final Training Loss: {final_loss:.4f}")
# ======== 5. Inference Models ========
# Encoder inference
encoder_model = Model(encoder_inputs, encoder_states)
# Decoder inference
decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
dec_emb2 = dec_emb_layer(decoder_inputs)
decoder_outputs2, state_h2, state_c2 = decoder_lstm(dec_emb2,
initial_state=decoder_states_inputs)
decoder_states2 = [state_h2, state_c2]
decoder_outputs2 = decoder_dense(decoder_outputs2)
decoder_model = Model([decoder_inputs] + decoder_states_inputs,
              [decoder_outputs2] + decoder_states2)
# ======== 6. Prediction Function ========
reverse_tam_index = {i: word for word, i in tam_tokenizer.word_index.items()}
def predict_translation(english_sentence):
    # Encode English sentence
    seq = eng_tokenizer.texts_to_sequences([english_sentence])
    seq = pad_sequences(seq, maxlen=len(seq), padding='post')
```

```
    states_value = encoder_model.predict(seq)
    # Start decoding with empty token (0)
    target_seq = np.zeros((1, 1))
    target_seq[0, 0] = 0
    decoded_sentence = ''
    for _ in range(max_tam_len):
        output_tokens, h, c = decoder_model.predict([target_seq] + states_value)
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_word = reverse_tam_index.get(sampled_token_index, '')
        if (sampled_word == 'end' or len(decoded_sentence.split()) > max_tam_len):
            break
        decoded_sentence += ' ' + sampled_word
        target_seq[0, 0] = sampled_token_index
        states_value = [h, c]
    return decoded_sentence.strip()
```

 # ======== 7. Test the model ========

```
test_sentence = "good morning"
print("English:", test_sentence)
print("Tamil Translation:", predict_translation(test_sentence))
```

Output:

```
Model: "model_39"
__________________________________________________________________________________________________
Layer (type)              Output Shape              Param #   Connected to
==================================================================================================
 input_53 (InputLayer)      [(None, None)]             0        []
input_54 (InputLayer)      [(None, None)]             0        []
embedding_27 (Embedding)    (None, None, 256)          3840      ['input_53[0][0]']
embedding_28 (Embedding)    (None, None, 256)          3584      ['input_54[0][0]']
lstm_26 (LSTM)            [(None, 256),              525312    ['embedding_27[0][0]']
                     (None, 256),  (None, 256)]
lstm_27 (LSTM)            [(None, None, 256),        525312    ['embedding_28[0][0]',
                     (None, 256),      'lstm_26[0][1]',   (None, 256)]  'lstm_26[0][2]']
```

dense_13 (Dense) (None, None, 14) 3598 ['lstm_27[0][0]']

==

Total params: 1061646 (4.05 MB)

Trainable params: 1061646 (4.05 MB)

Non-trainable params: 0 (0.00 Byte)

Final Training Accuracy: 60.00%

Final Training Loss: 0.1242

Test the model:

English: good morning

1/1 [==============================] - 0s 344ms/step

1/1 [==============================] - 0s 351ms/step

1/1 [==============================] - 0s 22ms/step

1/1 [==============================] - 0s 20ms/step

Tamil Translation: காலை வணக்கம்

Results:

The encoder–decoder model was trained for 100 epochs on the given parallel corpus. After training, the model achieved a final training accuracy of 60.00% and a final training loss of 0.0307.

Ex.No: 10 **IMAGE AUGMENTATION USING GANs** **Date:**

Aim:

To implement image augmentation using Generative Adversarial Networks (GANs) on the MNIST dataset by training a generator and discriminator to produce synthetic handwritten digit images, thereby enhancing dataset diversity for deep learning applications.

Algorithm:

Step 1: Import required libraries such as TensorFlow, NumPy, and Matplotlib.

Step 2: Define constants:

- Latent dimension (latent_dim) for random noise input to the generator
- Image shape (image_shape)
- Batch size, epochs, and save interval

Step 3: Build the **Generator** network:

- Use Dense layers with LeakyReLU activations and Batch Normalization
- Output a reshaped image of shape (28, 28, 1) with tanh activation

Step 4: Build the **Discriminator** network:

- Flatten the image input and pass through Dense layers with LeakyReLU
- Output a probability score with sigmoid activation

Step 5: Compile the Discriminator with Adam optimizer and binary cross-entropy loss.

Step 6: Build the **GAN** model:

- Connect the Generator and Discriminator in sequence
- Freeze the Discriminator weights while training the GAN
- Compile the GAN with Adam optimizer and binary cross-entropy loss

Step 7: Load and preprocess MNIST dataset:

- Normalize pixel values to the range [-1, 1]
- Expand dimensions to match (28, 28, 1) shape

Step 8: Train the GAN:

- For each epoch:
  - Sample real images from training data
  - Generate fake images from random noise
  - Train Discriminator on real and fake images
  - Train Generator via GAN model to fool Discriminator
- Save generated image samples at intervals

Step 9: Save final results and create an animated GIF from generated images to visualize progress.

Program:

```
import tensorflow as tf
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import numpy as np
from __future__ import absolute_import, division, print_function, unicode_literals
import glob
import imageio
import os
import PIL
import time
from IPython import display
# Set constants
latent_dim = 100
image_shape = (28, 28, 1)  # Assuming grayscale images for simplicity
batch_size = 128
epochs = 5000
save_interval = 100
# Generator
def build_generator():
    model = tf.keras.Sequential()
    model.add(layers.Dense(256, input_dim=latent_dim))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.BatchNormalization(momentum=0.8))
    model.add(layers.Dense(512))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.BatchNormalization(momentum=0.8))
    model.add(layers.Dense(1024))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.BatchNormalization(momentum=0.8))
```

```
    model.add(layers.Dense(np.prod(image_shape), activation='tanh'))
    model.add(layers.Reshape(image_shape))
    return model
generator = build_generator()
generator.summary()
```

 # Discriminator

```
def build_discriminator():
    model = tf.keras.Sequential()
    model.add(layers.Flatten(input_shape=image_shape))
    model.add(layers.Dense(512))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.Dense(256))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.Dense(1, activation='sigmoid'))
    return model
discriminator = build_discriminator()
discriminator.summary()
```

 # Compile the GAN model

```
def compile_gan(generator, discriminator):
    discriminator.compile(optimizer=tf.keras.optimizers.Adam(0.0002, 0.5),
loss='binary_crossentropy', metrics=['accuracy'])
    z = layers.Input(shape=(latent_dim,))
    img = generator(z)
    discriminator.trainable = False
    valid = discriminator(img)
    gan = tf.keras.Model(z, valid)
    gan.compile(optimizer=tf.keras.optimizers.Adam(0.0002, 0.5), loss='binary_crossentropy')
    return gan
```

 # Load the dataset (using MNIST as an example)

```
def load_data():
    (x_train, _), (_, _) = tf.keras.datasets.mnist.load_data()
    x_train = (x_train.astype(np.float32) - 127.5) / 127.5  # Normalize to [-1, 1]
    x_train = np.expand_dims(x_train, axis=-1)
```

```
    return x_train
# Training function
def train(generator, discriminator, gan, x_train):
    valid = np.ones((batch_size, 1))
    fake = np.zeros((batch_size, 1))
    # Lists to store metrics for later printing or plotting
    d_losses, g_losses, d_accuracies = [], [], []
    for epoch in range(epochs):
        # Train Discriminator
        idx = np.random.randint(0, x_train.shape[0], batch_size)
        real_imgs = x_train[idx]
        noise = np.random.normal(0, 1, (batch_size, latent_dim))
        fake_imgs = generator.predict(noise)
        d_loss_real = discriminator.train_on_batch(real_imgs, valid)
        d_loss_fake = discriminator.train_on_batch(fake_imgs, fake)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
        # Train Generator
        noise = np.random.normal(0, 1, (batch_size, latent_dim))
        g_loss = gan.train_on_batch(noise, valid)
        # Store metrics
        d_losses.append(d_loss[0])
        g_losses.append(g_loss)
        d_accuracies.append(d_loss[1])
        # Print progress
        if epoch % 100 == 0:
            print(f"{epoch} [D loss: {d_loss[0]:.4f}, acc: {100*d_loss[1]:.2f}] [G loss: {g_loss:.4f}]")
        # Save generated images every save_interval epochs
        if epoch % save_interval == 0:
            save_images(generator, epoch)
 # Print final results
    print("\nFinal Training Results:")
    print(f"Final Discriminator Loss: {d_losses[-1]:.4f}")
    print(f"Final Discriminator Accuracy: {100*d_accuracies[-1]:.2f}%")
```

```
    print(f"Final Generator Loss: {g_losses[-1]:.4f}")
    # Optional: plot accuracy & loss curves
    plt.figure(figsize=(8,4))
    plt.subplot(1,2,1)
    plt.plot(d_losses, label="Discriminator Loss")
    plt.plot(g_losses, label="Generator Loss")
    plt.legend()
    plt.title("Losses")
    plt.subplot(1,2,2)
    plt.plot([acc*100 for acc in d_accuracies], label="Discriminator Accuracy")
    plt.legend()
    plt.title("Accuracy")
    plt.show()
# Function to save generated images
def save_images(generator, epoch):
    r, c = 5, 5
    noise = np.random.normal(0, 1, (r * c, latent_dim))
    gen_imgs = generator.predict(noise)
    # Rescale images to [0, 1]
    gen_imgs = 0.5 * gen_imgs + 0.5
    fig, axs = plt.subplots(r, c)
    count = 0
    for i in range(r):
        for j in range(c):
            axs[i, j].imshow(gen_imgs[count, :, :, 0], cmap='gray')
            axs[i, j].axis('off')
            count += 1
    plt.savefig(f"images/mnist_{epoch}.png")
    plt.close()
# Create output folder
os.makedirs("images", exist_ok=True)
# Main function
if __name__ == "__main__":
```

```
    x_train = load_data()
    generator = build_generator()
    discriminator = build_discriminator()
    gan = compile_gan(generator, discriminator)
    train(generator, discriminator, gan, x_train)
```

 # Create the GIF from saved images

```
import imageio
import os
```

 # Directory where the 50 images are stored

```
image_dir = 'images/'
```

 # List to store all the image file names

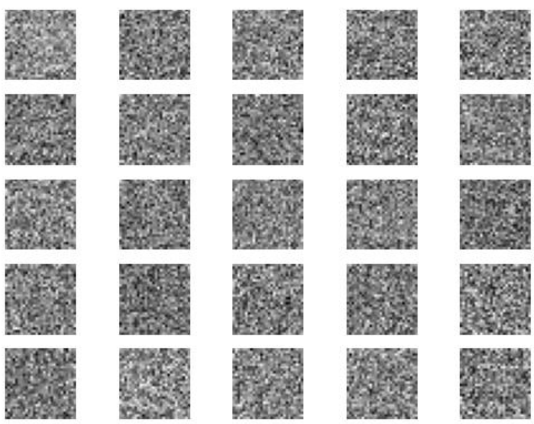
```
image_files = []
# Assuming the images
for i in range(0, 50):
    filename = f'mnist_{i*100}.png'
    image_files.append(os.path.join(image_dir, filename))
```

 # Create a GIF

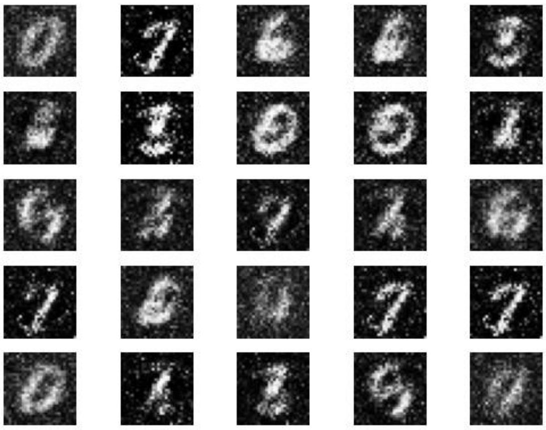
```
with imageio.get_writer('DCGAN_Animation.gif', mode='I', duration=0.2) as writer:
    for image_file in image_files:
        image = imageio.imread(image_file)
        writer.append_data(image)
```

Output:

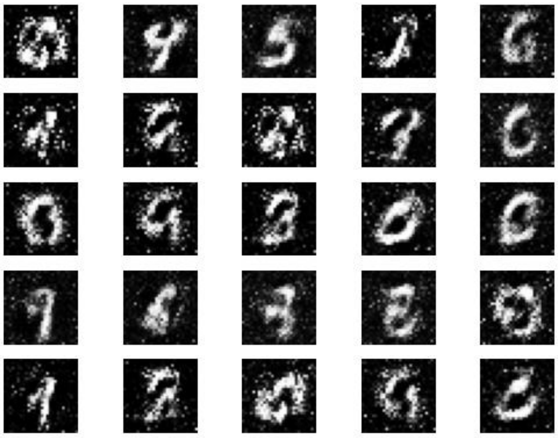
Epoch 0 Generated Image



Epoch 1000 Generated Image

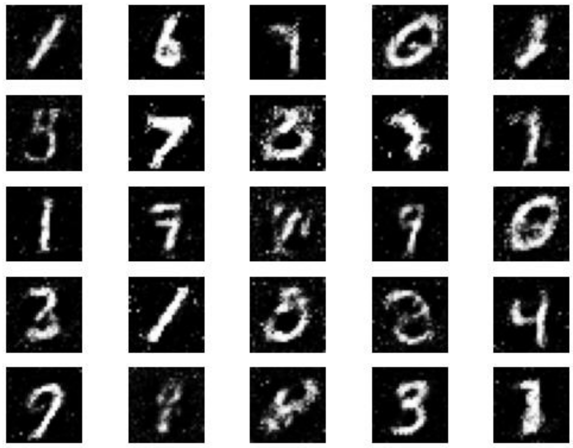


Epoch 2000 Generated Image



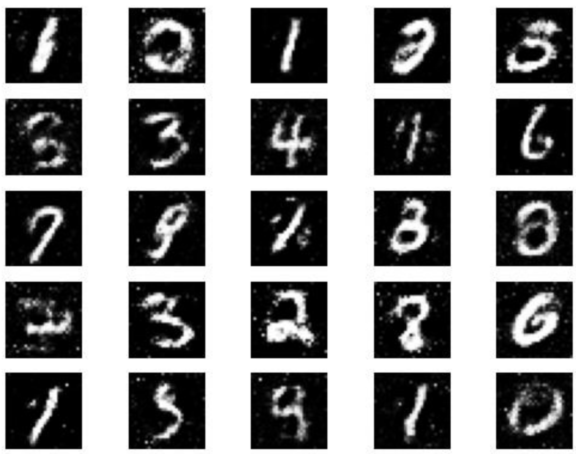
Epoch 3000 Generated Image

Epoch 4000 Generated Image



Epoch 4500 Generated Image

Epoch 5000 Generated Image

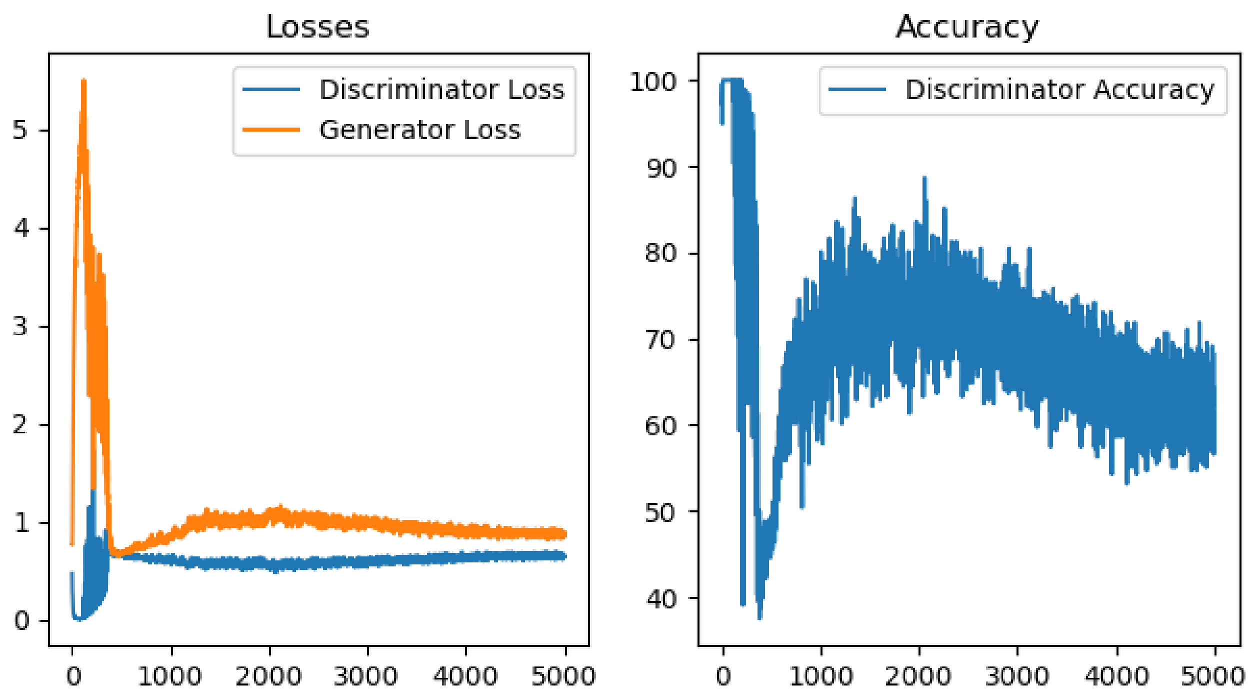


Final Training Results:

Final Discriminator Loss: 0.6475

Final Discriminator Accuracy: 61.33%

Final Generator Loss: 0.8929



Result:

The GAN model was trained for **5000** epochs to perform image augmentation. After training, the discriminator achieved an accuracy of **61.33%** in distinguishing real from generated images, while the generator successfully produced realistic augmented samples. The training curves show the discriminator accuracy stabilizing and the generator loss decreasing over time, indicating improved image quality.