



SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)
SRM Nagar, Kattankulathur-603203.



DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

ACADEMIC YEAR: 2026-2027 (ODD SEMESTER)

LAB MANUAL

(REGULATION - 2023)

AD3764 – Deep Learning Laboratory

SEVENTH SEMESTER

Prepared by

Ms. M. Abinaya, Assistant Professor / AI&DS

TABLE OF CONTENTS

Exp. No.	EXPERIMENT NAME	Page No.
A	PEO,PO,PSO	3
B	Syllabus	5
C	CO, CO-PO Matrix, CO-PSO Matrix	7
D	Mode of Assessment	10
1	Solving XOR Problem Using DNN	11
2	Character recognition using CNN.	17
3	Face recognition using CNN.	28
4	Language modeling using RNN.	34
5	Sentiment analysis using LSTM.	38
6	Parts of speech tagging using Sequence to Sequence architecture.	45
7	Machine Translation using Encoder-Decoder model.	54
8	Image augmentation using GANs.	61
9	Mini-project on real world applications databases.	69
10	TOPIC BEYOND THE SYLLABUS - Implementation of Doubly Linked List.	136

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

1. To afford the necessary background in the field of Information Technology to deal with engineering problems to excel as engineering professionals in industries.
2. To improve the qualities like creativity, leadership, teamwork, and skill thus contributing towards the growth and development of society.
3. To develop the ability among students towards innovation and entrepreneurship that caters to the needs of Industry and society.
4. To inculcate an attitude toward a life-long learning process through the use of information technology sources.
5. To prepare them to be innovative and ethical leaders, both in their chosen profession and in other activities.

PROGRAM OUTCOMES (POs)

After going through the four years of study, Bachelor of Technology in Information Technology Graduates will exhibit the ability to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to solve complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics:** Apply ethical principles and commit to professional ethics, responsibilities, and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OBJECTIVES (PSOs)

By the completion Bachelor of Technology in Information Technology program the student will have the following Program specific outcomes.

1. Design secured database applications involving planning, development, and maintenance using state-of-the-art methodologies based on ethical values.
2. Design and develop solutions for modern business environments coherent with advanced technologies and tools.
3. Design, plan, and set up a network that is helpful for contemporary business environments using the latest hardware components.
4. Planning and defining test activities by preparing test cases that can predict and correct errors ensuring a socially transformed product catering all technological needs.

1.5

OBJECTIVES:

- To understand the tools and techniques to implement deep neural networks.
- To apply different deep learning architectures for solving problems.
- To implement generative models for suitable applications.
- To learn to build and validate different models.
- To explore real world applications with deep neural networks.

LIST OF EXPERIMENTS:

1. Solving XOR problem using DNN.
2. Character recognition using CNN.
3. Face recognition using CNN.
4. Language modeling using RNN.
5. Sentiment analysis using LSTM.
6. Parts of speech tagging using Sequence to Sequence architecture.
7. Machine Translation using Encoder-Decoder model.
8. Image augmentation using GANs.
9. Mini-project on real world applications databases

OUTCOMES:**TOTAL: 45 PERIODS****At the end of the course, the student should be able to:**

- Apply deep neural network for simple problems.
- Apply Convolution Neural Network for image processing.
- Apply Recurrent Neural Network and its variants for text analysis.
- Apply generative models for data augmentation.
- Develop real-world solutions using suitable deep neural networks.

SOFTWARE REQUIREMENTS

Colab and Anaconda Navigator

CO – PO – PSO Mapping

CO	PO												PSO			
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4
1	3	2	-	-	-	-	-	-	-	2	1	-	-	2	3	2
2	-	3	1	2	2	-	-	-	3	-	-	-	-	2	2	-
3	3	2	-	2	-	-	-	-	-	-	2	-	-	2	2	3
4	3	3	-	2	-	-	-	-	-	-	-	-	-	1	2	-
5	3	2	2	3	2	-	-	-	2	-	-	-	-	2	3	2
Average	3.0	2.4	1.5	2.3	2.0	-	-	-	2.5	2.0	1.5	-	-	1.8	2.4	2.3



COURSE OUTCOMES:**Course Name: AD3764 – DEEP LEARNING LABORATORY****Year of study: 2026 –2027**

AD3764.1	Implement deep neural network (DNN) models to solve basic machine learning problems.
AD3764.2	Develop and evaluate Convolutional Neural Network (CNN) models for image recognition and computer vision applications.
AD3764.3	Implement Recurrent Neural Network (RNN) architectures and their variants (LSTM, Seq2Seq, Encoder–Decoder) for natural language processing tasks.
AD3764.4	Apply Generative Adversarial Networks (GANs) for image augmentation and generative learning applications.
AD3764.5	Design, implement, validate, and present deep learning solutions for real-world problems using appropriate datasets and modern deep learning frameworks.

CO-PO Matrix:

CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
AD3764.1	3	2	-	-	-	-	-	-	-	2	1	-
AD3764.2	-	3	1	2	2	-	-	-	3	-	-	-
AD3764.3	3	2	-	2	-	-	-	-	-	-	2	-
AD3764.4	3	3	-	2	-	-	-	-	-	-	-	-
AD3764.5	3	2	2	3	2	-	-	-	2	-	-	-

Justification:

Course Outcome	PO	Value	Justification
AD3764.1	PO1	3	Implement deep neural networks to solve simple problems.
	PO2	2	Analyze the performance of deep learning models.
	PO10	2	Present the implementation and results of deep learning models effectively.
	PO11	1	Use available resources effectively while implementing deep learning experiments.
AD3764.2	PO2	3	Analyze image processing problems using CNN models.
	PO3	1	Design CNN models for image recognition tasks.
	PO4	2	Evaluate the performance of CNN models using experimental results.

	PO5	2	Use deep learning tools to implement CNN models.
	PO9	3	Work effectively in a team to complete CNN-based laboratory experiments.
AD3764.3	PO1	3	Apply RNN and LSTM models for text analysis tasks.
	PO2	2	Analyze text data using suitable deep learning models.
	PO4	2	Evaluate the performance of RNN and LSTM models.
	PO11	2	Use computing resources effectively for text analysis applications.
AD3764.4	PO1	3	Apply generative models for data augmentation.
	PO2	3	Analyze the performance of generative models.
	PO4	2	Evaluate the results of data augmentation using generative models.
AD3764.5	PO1	3	Apply deep learning concepts to solve real-world problems.
	PO2	2	Analyze real-world problems and select suitable deep learning models.
	PO3	2	Design deep learning solutions for real-world applications.
	PO4	3	Evaluate the performance of developed deep learning models.
	PO5	2	Use modern deep learning tools to develop and test applications.
	PO9	2	Work effectively as a team to complete the mini-project.

CO-PO Average:

CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
AD3764	3.0	2.4	1.5	2.3	2.0	-	-	-	2.5	2.0	1.5	-

CO-PSO Matrix:

CO	PSO1	PSO2	PSO3	PSO4
AD3764.1	-	2	3	2
AD3764.2	-	2	2	-
AD3764.3	-	2	2	3
AD3764.4	-	1	2	-
AD3764.5	-	2	3	2

Justification:

Course Outcome	PSO	Value	Justification
AD3764.1	PSO2	2	Apply deep learning techniques to solve simple problems.
	PSO3	3	Use programming tools to implement deep neural networks.
	PSO4	2	Develop basic deep learning solutions for practical applications.

AD3764.2	PSO2	2	Apply CNN techniques for image processing applications.
	PSO3	2	Use deep learning tools to implement CNN models.
AD3764.3	PSO2	2	Apply RNN and LSTM models for text analysis.
	PSO3	2	Use deep learning tools to implement text processing models.
	PSO4	3	Develop text analysis applications using deep learning models.
AD3764.4	PSO2	1	Apply generative models for data augmentation.
	PSO3	2	Use deep learning tools to implement generative models.
AD3764.5	PSO2	2	Apply deep learning techniques to solve real-world problems.
	PSO3	3	Use deep learning tools to develop real-world applications.
	PSO4	2	Develop practical deep learning solutions for real-world applications.

CO-PSO Average:

CO	PSO 1	PSO 2	PSO 3	PSO 4
AD3764	-	1.8	2.4	2.3



ASSESSMENT METHOD

i. EVALUATION PROCEDURE FOR EACH EXPERIMENTS

S. No.	Description	Marks
1	Aim & Pre-Lab discussion	20
2	Observation	20
3	Conduction and Execution	30
4	Output & Result	10
5	Viva	20
	Total	100

ii. INTERNAL ASSESSMENT FOR LABORATORY

S. No.	Description	Marks
1	Conduction & Execution of Experiment	50
2	Record	17
3	Model Test	33
	Total	100

Ex.No: 1 SOLVING XOR PROBLEM USING DNN

Aim:

To write a program to Solving XOR problem using DNN.

Algorithm:

Step 1: Start the program and import the required Python libraries such as NumPy for numerical computations, Matplotlib for plotting the learning curve, and Sys for displaying the training progress.

Step 2: Define the input dataset **X** consisting of four training samples representing the XOR logic gate combinations.

Step 3: Define the corresponding target output **y** for each training sample.

Step 4: Initialize the neural network architecture by specifying:

- Number of input neurons = 2
- Number of hidden neurons = 2
- Number of output neurons = 1

Step 5: Set the hyperparameters including:

- Learning rate
- Regularization parameter
- Maximum number of training iterations
- Number of training samples.

Step 6: Initialize the random seed to ensure reproducible results during every execution.

Step 7: Randomly initialize the weight matrices **W1** (Input to Hidden Layer) and **W2** (Hidden Layer to Output Layer).

Step 8: Randomly initialize the bias vectors **B1** for the hidden layer and **B2** for the output layer.

Step 9: Define the Sigmoid activation function to transform the weighted sum into values between 0 and 1.

Step 10: Define the derivative of the Sigmoid function, which is used during the backpropagation process to compute gradients.

Step 11: Perform **Forward Propagation** by calculating the weighted sum of the input layer and hidden layer using:

- Hidden Layer Input:
 $Z_1 = W_1 \times X + B_1$
- Hidden Layer Output:
 $A_1 = \text{Sigmoid}(Z_1)$

Step 12: Compute the weighted sum at the output layer using:

- Output Layer Input:
 $Z_2 = W_2 \times A_1 + B_2$
- Final Network Output:
 $A_2 = \text{Sigmoid}(Z_2)$

Step 13: Compare the predicted output with the actual target output and compute the Binary Cross-Entropy (Log Loss) cost function.

Step 14: Begin the **Backpropagation** process by calculating the output layer error as the difference between the predicted output and the actual output.

Step 15: Compute the gradients of the output layer weights and biases using the calculated output error.

Step 16: Propagate the error backward to the hidden layer by multiplying the output error with the transpose of the output weight matrix and the derivative of the Sigmoid activation function.

Step 17: Calculate the gradients of the hidden layer weights and biases using the hidden layer error.

Step 18: Accumulate the gradients for all training samples before updating the network parameters.

Step 19: Update the weights and biases using the Gradient Descent optimization algorithm according to the learning rate.

Step 20: Apply regularization (if specified) while updating the weights to minimize overfitting.

Step 21: Store the computed cost value after each iteration for monitoring the convergence of the network.

Step 22: Repeat the forward propagation, backpropagation, gradient computation, and parameter update process until the maximum number of iterations is reached.

Step 23: After completion of training, obtain the optimized weights and biases for the neural network.

Step 24: Plot the Cost versus Iterations graph using Matplotlib to visualize the reduction in training error and the convergence of the learning process.

Step 25: End the program.

Program:

```
import numpy as np # For matrix math
import matplotlib.pyplot as plt # For plotting
import sys # For printing
# The training data.
X = np.array([
[0, 1],
[1, 0],
[1, 1],
[0, 0]
])
# The labels for the training data.
y = np.array([
[1],
[1],
[0],
[0]
])
num_i_units = 2 # Number of Input units
num_h_units = 2 # Number of Hidden units
num_o_units = 1 # Number of Output units
# The learning rate for Gradient Descent.
learning_rate = 0.01
# The parameter to help with overfitting.
reg_param = 0
# Maximum iterations for Gradient Descent.
max_iter = 5000
# Number of training examples
m = 4
np.random.seed(1)
W1 = np.random.normal(0, 1, (num_h_units, num_i_units)) # 2x2
W2 = np.random.normal(0, 1, (num_o_units, num_h_units)) # 1x2
B1 = np.random.random((num_h_units, 1)) # 2x1
B2 = np.random.random((num_o_units, 1)) # 1x1
def sigmoid(z, derv=False):
if derv: return z * (1 - z)
return 1 / (1 + np.exp(-z))
def forward(x, predict=False):
```

```

a1 = x.reshape(x.shape[0], 1) # Getting the training example as a
column vector.
z2 = W1.dot(a1) + B1 #  $2 \times 2 * 2 \times 1 + 2 \times 1 = 2 \times 1$ 
a2 = sigmoid(z2) #  $2 \times 1$ 
z3 = W2.dot(a2) + B2 #  $1 \times 2 * 2 \times 1 + 1 \times 1 = 1 \times 1$ 
a3 = sigmoid(z3)
if predict: return a3
return (a1, a2, a3)
dW1 = 0 # Gradient for W1
dW2 = 0 # Gradient for W2
dB1 = 0 # Gradient for B1
dB2 = 0 # Gradient for B2
cost = np.zeros((max_iter, 1)) # Column vector to record the cost of the
NN after each Gradient Descent iteration.
def train(_W1, _W2, _B1, _B2): # The arguments are to bypass
UnboundLocalError error
for i in range(max_iter):
c = 0
dW1 = 0
dW2 = 0
dB1 = 0
dB2 = 0
for j in range(m):
sys.stdout.write("\rIteration: {} and {}".format(i + 1, j + 1))
# Forward Prop.
a0 = X[j].reshape(X[j].shape[0], 1) #  $2 \times 1$ 
z1 = _W1.dot(a0) + _B1 #  $2 \times 2 * 2 \times 1 + 2 \times 1 = 2 \times 1$ 
a1 = sigmoid(z1) #  $2 \times 1$ 
z2 = _W2.dot(a1) + _B2 #  $1 \times 2 * 2 \times 1 + 1 \times 1 = 1 \times 1$ 
a2 = sigmoid(z2) #  $1 \times 1$ 
# Back prop.
dz2 = a2 - y[j] #  $1 \times 1$ 
dW2 += dz2 * a1.T #  $1 \times 1 .* 1 \times 2 = 1 \times 2$ 
dz1 = np.multiply(_W2.T * dz2, sigmoid(a1, deriv=True)) #
( $2 \times 1 * 1 \times 1$ ) . $*$   $2 \times 1 = 2 \times 1$ 
dW1 += dz1.dot(a0.T) #  $2 \times 1 * 1 \times 2 = 2 \times 2$ 
dB1 += dz1 #  $2 \times 1$ 
dB2 += dz2 #  $1 \times 1$ 
c = c + ((-y[j] * np.log(a2)) - ((1 - y[j]) * np.log(1 - a2)))
sys.stdout.flush() # Updating the text.
_W1 = _W1 - learning_rate * (dW1 / m) + ( (reg_param / m) *
_W1)
_W2 = _W2 - learning_rate * (dW2 / m) + ( (reg_param / m) *
_W2)
_B1 = _B1 - learning_rate * (dB1 / m)
_B2 = _B2 - learning_rate * (dB2 / m)

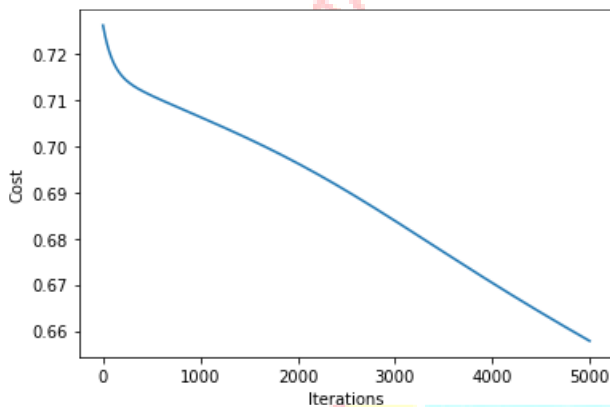
```

```

cost[i] = (c / m) + (
    (reg_param / (2 * m)) *
    (
        np.sum(np.power(_W1, 2)) +
        np.sum(np.power(_W2, 2))
    )
)
return (_W1, _W2, _B1, _B2)
W1, W2, B1, B2 = train(W1, W2, B1, B2)
# Assigning the axes to the different elements.
plt.plot(range(max_iter), cost)
# Labelling the x axis as the iterations axis.
plt.xlabel("Iterations")
# Labelling the y axis as the cost axis.
plt.ylabel("Cost")
# Showing the plot.
plt.show()

```

Output:



Result:

Thus the Python program successfully to Solving XOR problem using DNN.

Ex.No: 2 CHARACTER RECOGNITION USING CNN

Aim:

To write a python program to implement the Character recognition using CNN.

Algorithm:

Step 1: Start the program and install the required Python libraries such as OpenCV, Keras, TensorFlow, NumPy, Pandas, Matplotlib, and Scikit-learn.

Step 2: Import all the necessary libraries required for image processing, data manipulation, visualization, model creation, and evaluation.

Step 3: Load the handwritten alphabet dataset from the CSV file into a Pandas DataFrame.

Step 4: Separate the dataset into:

- **Input features (X):** Pixel values of handwritten character images.
- **Output labels (y):** Corresponding alphabet classes (A–Z).

Step 5: Split the dataset into training and testing sets using the train-test split method to evaluate the model performance.

Step 6: Reshape the flattened pixel values into 28×28 grayscale images for both training and testing datasets.

Step 7: Create a dictionary that maps numerical class labels (0–25) to their corresponding alphabet characters (A–Z).

Step 8: Count the number of samples available for each alphabet class to understand the dataset distribution.

Step 9: Plot a horizontal bar chart showing the frequency of each alphabet in the dataset using Matplotlib.

Step 10: Shuffle a subset of training images randomly to visualize different handwritten alphabet samples.

Step 11: Apply binary thresholding to the shuffled images for better visualization of handwritten characters.

Step 12: Display the sample handwritten alphabet images using grayscale color mapping.

Step 13: Reshape the training and testing datasets into four-dimensional tensors of size:

- **(Number of Samples, 28, 28, 1)**
to make them compatible with the Convolutional Neural Network.

Step 14: Convert the class labels into **One-Hot Encoded** vectors using categorical encoding for multi-class classification.

Step 15: Initialize a Sequential Convolutional Neural Network (CNN) model.

Step 16: Add the first **Convolutional Layer** with 32 filters of size 3×3 and ReLU activation to extract low-level image features.

Step 17: Add a **Max Pooling Layer** to reduce the spatial dimensions and computational complexity.

Step 18: Add the second **Convolutional Layer** with 64 filters followed by another Max Pooling layer to learn more complex image features.

Step 19: Add the third **Convolutional Layer** with 128 filters followed by another Max Pooling layer to capture high-level feature representations.

Step 20: Flatten the extracted feature maps into a one-dimensional feature vector.

Step 21: Add fully connected (Dense) hidden layers with ReLU activation to perform feature learning and classification.

Step 22: Add the final output layer consisting of **26 neurons** with **Softmax activation** to classify the handwritten character into one of the 26 alphabet classes.

Step 23: Compile the CNN model using:

- **Optimizer:** Adam
- **Loss Function:** Categorical Cross-Entropy
- **Evaluation Metric:** Accuracy

Step 24: Train the CNN model using the training dataset for the specified number of epochs while validating the performance on the testing dataset.

Step 25: Display the model architecture and summary, including the number of layers, parameters, and output shapes.

Step 26: Save the trained CNN model into an H5 file for future prediction without retraining.

Step 27: Display the training accuracy, validation accuracy, training loss, and validation loss obtained after model training.

Step 28: Predict the handwritten characters for the test dataset using the trained CNN model.

Step 29: Display the predicted alphabet labels along with their corresponding handwritten test images.

Step 30: Load an external handwritten character image using OpenCV.

Step 31: Preprocess the external image by:

- Converting it to RGB format.
- Resizing the image.
- Applying Gaussian Blur to remove noise.
- Converting it into grayscale.
- Applying binary thresholding.
- Resizing it to **28 × 28** pixels.
- Reshaping it into the input format required by the CNN model.

Step 32: Feed the processed image into the trained CNN model and predict the corresponding alphabet character.

Step 33: Display the predicted character on the original image using OpenCV text annotation.

Step 34: Show the final output window containing the handwritten image and its predicted alphabet.

Step 35: Wait until the user presses the **Esc** key to close the display window.

Step 36: End the program.

Program:

```

pip install opencv-python
pip install keras
pip install tensorflow
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPool2D, Dropout
from keras.optimizers import SGD, Adam
from keras.callbacks import ReduceLROnPlateau, EarlyStopping
from keras.utils import to_categorical
data = pd.read_csv(r"A_Z Handwritten Data.csv").astype('float32')
X = data.drop('0',axis = 1)
y = data['0']
train_x, test_x, train_y, test_y = train_test_split(X, y, test_size = 0.2)
train_x = np.reshape(train_x.values, (train_x.shape[0], 28,28))
test_x = np.reshape(test_x.values, (test_x.shape[0], 28,28))
word_dict =

```

```

{0:'A',1:'B',2:'C',3:'D',4:'E',5:'F',6:'G',7:'H',8:'I',9:'J',10:'K',11:'L',12:'M',13:'N',14:'O',
15:'P',16:'Q',17:'R',18:'S',19:'T',20:'U',21:'V',22:'W',23:'X',24:'Y',25:'Z'}
y_int = np.int0(y)
count = np.zeros(26, dtype='int')
for i in y_int:
    count[i] +=1
alphabets = []
for i in word_dict.values():
    alphabets.append(i)
fig, ax = plt.subplots(1,1, figsize=(10,10))
ax.barh(alphabets, count)
# naming the x axis
plt.xlabel("Number of elements")
# naming the y axis
plt.ylabel("Alphabets")
# giving a title
plt.title("Plotting the number of alphabets")
# Turn on the minor TICKS, which are required for the minor GRID
plt.minorticks_on()
# Customize the major grid
plt.grid(which='major', linestyle='-', linewidth='0.5', color='red')
# Customize the minor grid
plt.grid(which='minor', linestyle=':', linewidth='0.5', color='black')
plt.show()
uff = shuffle(train_x[:100])
fig, ax = plt.subplots(3,3, figsize = (10,10))
axes = ax.flatten()
for i in range(9):
    _, shu = cv2.threshold(shuff[i], 30, 200, cv2.THRESH_BINARY)
    axes[i].imshow(np.reshape(shuff[i], (28,28)), cmap=plt.get_cmap('gray'))
plt.show()
# Reshape data for model creation
train_X = train_x.reshape(train_x.shape[0],train_x.shape[1],train_x.shape[2],1)
print("The new shape of train data: ", train_X.shape)
test_X = test_x.reshape(test_x.shape[0], test_x.shape[1], test_x.shape[2],1)
print("The new shape of train data: ", test_X.shape)
train_yOHE = to_categorical(train_y, num_classes = 26, dtype='int')
print("The new shape of train labels: ", train_yOHE.shape)
test_yOHE = to_categorical(test_y, num_classes = 26, dtype='int')
print("The new shape of test labels: ", test_yOHE.shape)
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(3, 3), activation='relu',
input_shape=(28,28,1)))
model.add(MaxPool2D(pool_size=(2, 2), strides=2))
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding =
'same'))

```

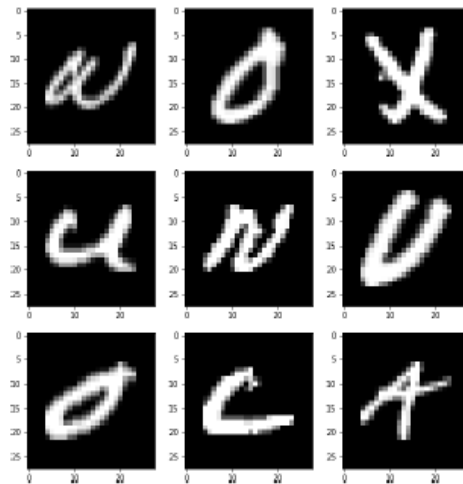
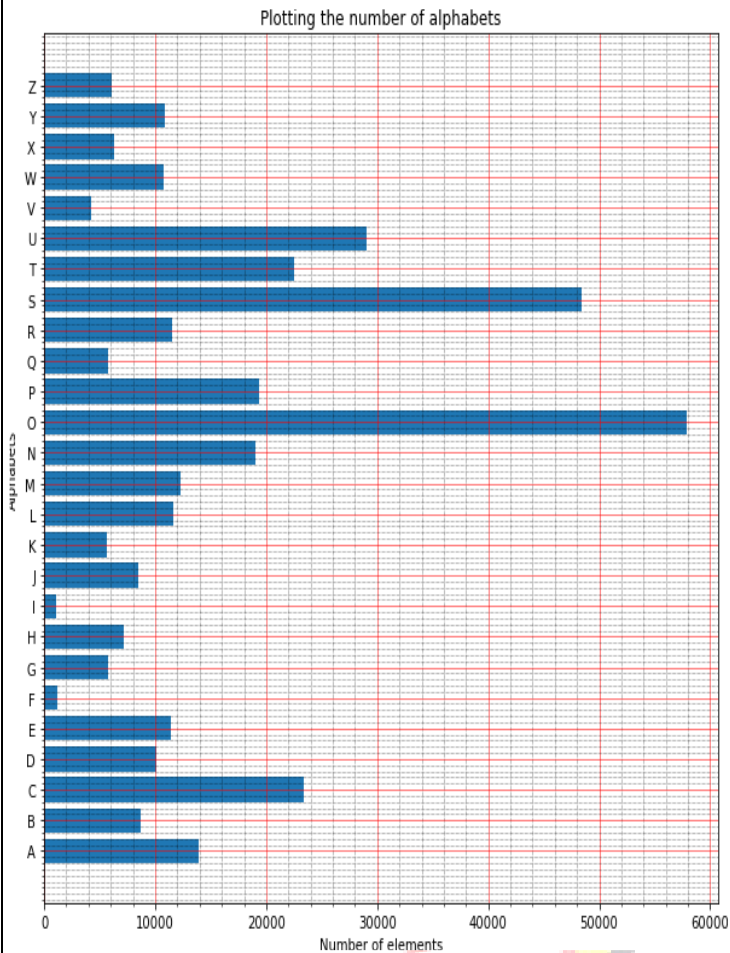
```

model.add(MaxPool2D(pool_size=(2, 2), strides=2))
model.add(Conv2D(filters=128, kernel_size=(3, 3), activation='relu', padding =
'valid'))
model.add(MaxPool2D(pool_size=(2, 2), strides=2))
model.add(Flatten())
model.add(Dense(64,activation ="relu"))
model.add(Dense(128,activation ="relu"))
model.add(Dense(26,activation ="softmax"))
model.compile(optimizer = Adam(learning_rate=0.001),
loss='categorical_crossentropy', metrics=['accuracy'])
history = model.fit(train_X, train_yOHE, epochs=1, validation_data =
(test_X,test_yOHE))
model.summary()
model.save(r'model_hand.h5')
print("The validation accuracy is :", history.history['val_accuracy'])
print("The training accuracy is :", history.history['accuracy'])
print("The validation loss is :", history.history['val_loss'])
print("The training loss is :", history.history['loss'])
# Prediction on test data
fig, axes = plt.subplots(3,3, figsize=(8,9))
axes = axes.flatten()
for i,ax in enumerate(axes):
img = np.reshape(test_X[i], (28,28))
ax.imshow(img, cmap=plt.get_cmap('gray'))
pred = word_dict[np.argmax(test_yOHE[i])]
ax.set_title("Prediction: "+pred)
# Prededction on External Image
img = cv2.imread(r'test_image.jpg')
img_copy = img.copy()
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img = cv2.resize(img, (400,440))
img_copy = cv2.GaussianBlur(img_copy, (7,7), 0)
img_gray = cv2.cvtColor(img_copy, cv2.COLOR_BGR2GRAY)
_, img_thresh = cv2.threshold(img_gray, 100, 255, cv2.THRESH_BINARY_INV)
img_final = cv2.resize(img_thresh, (28,28))
img_final = np.reshape(img_final, (1,28,28,1))
img_pred = word_dict[np.argmax(model.predict(img_final))]
cv2.putText(img, "Image Data", (100,25), cv2.FONT_HERSHEY_DUPLEX ,
fontScale= 1, thickness=2, color = (255,0,0))
cv2.putText(img, "Character Prediction: " + img_pred, (10,410),
cv2.FONT_HERSHEY_SIMPLEX, fontScale= 1, thickness=2, color = (0,0,255))
cv2.imshow('Character Recognition', img)
while (1):
k = cv2.waitKey(1) & 0xFF
if k == 27:
break

```

cv2.destroyAllWindows()

Output:



The new shape of train data: (297960, 28, 28, 1)
The new shape of train data: (74490, 28, 28, 1)
The new shape of train labels: (297960, 26)
The new shape of test labels: (74490, 26)

9312/9312 [=====] - 85s 9ms/step - loss : 0.1440
- accuracy: 0.9595 - val_loss: 0.0853 - val_accuracy: 0.9761

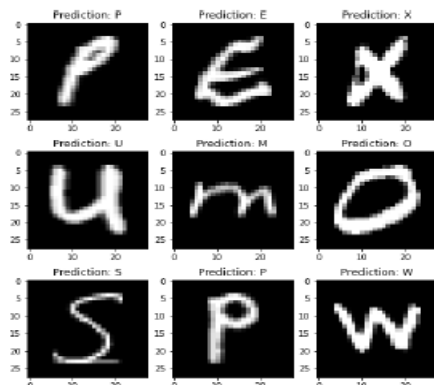
model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	73856

```
max_pooling2d_2 (MaxPooling 2D)      (None, 2, 2, 128)      0
flatten (Flatten)                   (None, 512)             0
dense (Dense)                       (None, 64)              32832
dense_1 (Dense)                    (None, 128)             8320
dense_2 (Dense)                    (None, 26)              3254
```

Total params: 137,178
Trainable params: 137,178
Non-trainable params: 0

The validation accuracy is : [0.9760907292366028]
The training accuracy is : [0.9595012664794922]
The validation loss is : [0.08530429750680923]
The training loss is : [0.1440141350030899]



1/1 [=====] - 0s 79ms/step

Result:

Thus written a python program to implement successfully for the Character recognition using CNN.

Ex.No: 3 FACE RECOGNITION USING CNN

Aim:

To write a python program to implement the Face recognition using CNN.

Algorithm:

Step 1: Start the program and import the required Python libraries such as NumPy, Pandas, Matplotlib, Seaborn, TensorFlow/Keras, and Scikit-learn.

Step 2: Load the **Labeled Faces in the Wild (LFW)** face dataset using the `fetch_lfw_people()` function.

Step 3: Specify the dataset parameters such as:

- Minimum number of images per person.
- Image resizing factor.
- Cropping region of the face.
- Color image format.

Step 4: Determine the total number of unique persons (classes) present in the dataset.

Step 5: Display the names of all individuals and the dimensions of the loaded face images.

Step 6: Visualize a few sample face images along with their corresponding class labels using Matplotlib.

Step 7: Count the number of face images available for each person in the dataset.

Step 8: Create a bar chart to visualize the distribution of face images among different individuals.

Step 9: Create a Boolean mask to select an equal number of images (100 images) for each individual to balance the dataset.

Step 10: Extract the selected face images and their corresponding class labels using the generated mask.

Step 11: Reshape the extracted face images into the four-dimensional format:

- **(Number of Samples, Height, Width, Channels)**

to make them suitable for CNN processing.

Step 12: Normalize all pixel values by dividing them by **255**, thereby scaling the pixel intensity values between **0 and 1**.

Step 13: Convert the class labels into **One-Hot Encoded** vectors for multi-class face classification.

Step 14: Split the normalized dataset into **training** and **testing** sets while maintaining equal class distribution using stratified sampling.

Step 15: Initialize a **Sequential Convolutional Neural Network (CNN)** model.

Step 16: Add the first **Convolutional Layer** with **32 filters** of size 3×3 and ReLU activation to extract basic facial features.

Step 17: Add a **Max Pooling Layer** to reduce the spatial dimensions and retain important facial information.

Step 18: Add the second **Convolutional Layer** with **64 filters** followed by another Max Pooling layer to learn more complex facial features.

Step 19: Add the third **Convolutional Layer** with **64 filters** followed by a Max Pooling layer to capture high-level facial feature representations.

Step 20: Flatten the extracted feature maps into a one-dimensional feature vector.

Step 21: Add a fully connected (**Dense**) hidden layer with **128 neurons** and ReLU activation to perform feature learning.

Step 22: Add the output layer containing neurons equal to the number of face classes using the **Softmax activation function** for multi-class face recognition.

Step 23: Compile the CNN model using:

- **Optimizer:** Adam
- **Loss Function:** Categorical Cross-Entropy
- **Performance Metric:** Accuracy

Step 24: Display the architecture and summary of the CNN model.

Step 25: Train the CNN model using the training dataset with the specified batch size and number of

epochs.

Step 26: Validate the trained model using the testing dataset during each training epoch.

Step 27: Record the training accuracy and validation accuracy after every epoch.

Step 28: Plot the **Training Accuracy** and **Validation Accuracy** curves to analyze the learning performance of the CNN model.

Step 29: Use the trained CNN model to predict the class labels of the testing images.

Step 30: Generate the **Confusion Matrix** by comparing the predicted labels with the actual labels.

Step 31: Display the confusion matrix as a heat map to evaluate the classification performance for each individual.

Step 32: Load an external face image for testing using the Keras image preprocessing utilities.

Step 33: Resize the external image to match the input size expected by the trained CNN model.

Step 34: Display the selected face image for visual verification.

Step 35: Convert the image into an array, normalize its pixel values, and expand its dimensions to create a batch of one image.

Step 36: Feed the processed image into the trained CNN model to predict the identity of the person.

Step 37: Display the probability score for every individual class and identify the person with the highest prediction probability as the recognized face.

Step 38: End the program.

Program:

```
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_lfw_people
faces = fetch_lfw_people(min_faces_per_person=100, resize=1.0, slice_=(slice(60,
188), slice(60, 188)), color=True)
class_count = len(faces.target_names)
print(faces.target_names)
print(faces.images.shape)
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
fig, ax = plt.subplots(3, 6, figsize=(18, 10))
for i, axi in enumerate(ax.flat):
    axi.imshow(faces.images[i] / 255) # Scale pixel values so Matplotlib doesn't clip
    everything above 1.0
    axi.set(xticks=[], yticks=[], xlabel=faces.target_names[faces.target[i]])
from collections import Counter
counts = Counter(faces.target)
names = {}
for key in counts.keys():
    names[faces.target_names[key]] = counts[key]
df = pd.DataFrame.from_dict(names, orient='index')
df.plot(kind='bar')
mask = np.zeros(faces.target.shape, dtype=np.bool)
for target in np.unique(faces.target):
    mask[np.where(faces.target == target)[0][:100]] = 1
```

```

x_faces = faces.data[mask]
y_faces = faces.target[mask]
x_faces = np.reshape(x_faces, (x_faces.shape[0], faces.images.shape[1],
faces.images.shape[2], faces.images.shape[3]))
x_faces.shape
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
face_images = x_faces / 255 # Normalize pixel values
face_labels = to_categorical(y_faces)
x_train, x_test, y_train, y_test = train_test_split(face_images, face_labels,
train_size=0.8, stratify=face_labels, random_state=0)
from keras.layers import Dense
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu',
input_shape=(face_images.shape[1:]))))
model.add(MaxPooling2D(2, 2))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(2, 2))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(2, 2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(class_count, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.summary()
hist = model.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=20,
batch_size=25)
acc = hist.history['accuracy']
val_acc = hist.history['val_accuracy']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, '-', label='Training Accuracy')
plt.plot(epochs, val_acc, ' ', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.plot()
from sklearn.metrics import confusion_matrix
y_predicted = model.predict(x_test)
mat = confusion_matrix(y_test.argmax(axis=1), y_predicted.argmax(axis=1))
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False, cmap='Blues',

```

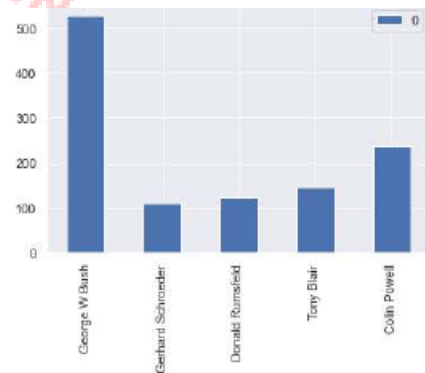
```

xticklabels=faces.target_names,
yticklabels=faces.target_names)
plt.xlabel('Predicted label')
plt.ylabel('Actual label')
import keras.utils as image
x = image.load_img('george.jpg', target_size=(face_images.shape[1:]))
plt.xticks([])
plt.yticks([])
plt.imshow(x)
x = image.img_to_array(x) / 255
x = np.expand_dims(x, axis=0)
y = model.predict(x)[0]
for i in range(len(y)):
    print(faces.target_names[i] + ': ' + str(y[i]))

```

Output:

['Colin Powell' 'Donald Rumsfeld' 'George W Bush' 'Gerhard Schroeder'
'Tony Blair']
(1140, 128, 128, 3)



(500, 128, 128, 3)

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 126, 126, 32)	896
max_pooling2d (MaxPooling2D)	(None, 63, 63, 32)	0
conv2d_1 (Conv2D)	(None, 61, 61, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 30, 30, 64)	0
conv2d_2 (Conv2D)	(None, 28, 28, 64)	36928
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 64)	0
flatten (Flatten)	(None, 12544)	0
dense (Dense)	(None, 128)	1605760
dense_1 (Dense)	(None, 5)	645
Total params: 1,662,725		
Trainable params: 1,662,725		
Non-trainable params: 0		

EPOCH 1/20

16/16 [=====] - 2s 123ms/step - loss: 1.6558 - accuracy: 0.1925 -
val_loss: 1.6038 - val_accuracy: 0.2000
Epoch 2/20
16/16 [=====] - 2s 110ms/step - loss: 1.5860 - accuracy: 0.3175 -
val_loss: 1.5416 - val_accuracy: 0.3200
Epoch 3/20
16/16 [=====] - 2s 112ms/step - loss: 1.4851 - accuracy: 0.3675 -
val_loss: 1.3706 - val_accuracy: 0.4500
Epoch 4/20
16/16 [=====] - 2s 110ms/step - loss: 1.1602 - accuracy: 0.5775 -
val_loss: 1.0931 - val_accuracy: 0.5900
Epoch 5/20
16/16 [=====] - 2s 112ms/step - loss: 0.8385 - accuracy: 0.7000 -
val_loss: 0.8494 - val_accuracy: 0.6700
Epoch 6/20
16/16 [=====] - 2s 111ms/step - loss: 0.5011 - accuracy: 0.8275 -
val_loss: 0.8085 - val_accuracy: 0.6900
Epoch 7/20
16/16 [=====] - 2s 111ms/step - loss: 0.3819 - accuracy: 0.8550 -
val_loss: 0.7241 - val_accuracy: 0.7200
Epoch 8/20
16/16 [=====] - 2s 110ms/step - loss: 0.3558 - accuracy: 0.8950 -
val_loss: 0.5499 - val_accuracy: 0.7800
Epoch 9/20
16/16 [=====] - 2s 114ms/step - loss: 0.1407 - accuracy: 0.9575 -
val_loss: 0.7090 - val_accuracy: 0.8000
Epoch 10/20
16/16 [=====] - 2s 115ms/step - loss: 0.0869 - accuracy: 0.9875 -
val_loss: 0.6296 - val_accuracy: 0.8400
Epoch 11/20
16/16 [=====] - 2s 111ms/step - loss: 0.0413 - accuracy: 0.9950 -
val_loss: 0.5816 - val_accuracy: 0.8300
Epoch 12/20
16/16 [=====] - 2s 110ms/step - loss: 0.0325 - accuracy: 0.9950 -
val_loss: 0.5888 - val_accuracy: 0.8300
Epoch 13/20
16/16 [=====] - 2s 110ms/step - loss: 0.0359 - accuracy: 0.9900 -
val_loss: 0.6945 - val_accuracy: 0.8100
Epoch 14/20
16/16 [=====] - 2s 110ms/step - loss: 0.0085 - accuracy: 1.0000 -
val_loss: 0.5278 - val_accuracy: 0.8600
Epoch 15/20
16/16 [=====] - 2s 111ms/step - loss: 0.0048 - accuracy: 1.0000 -
val_loss: 0.5697 - val_accuracy: 0.8500
Epoch 16/20

16/16 [=====] - 2s 111ms/step - loss: 0.0032 - accuracy: 1.0000 -
 val_loss: 0.6065 - val_accuracy: 0.8500
 Epoch 17/20
 16/16 [=====] - 2s 110ms/step - loss: 0.0022 - accuracy: 1.0000 -
 val_loss: 0.6007 - val_accuracy: 0.8500
 Epoch 18/20
 16/16 [=====] - 2s 112ms/step - loss: 0.0017 - accuracy: 1.0000 -
 val_loss: 0.6242 - val_accuracy: 0.8500
 Epoch 19/20
 16/16 [=====] - 2s 118ms/step - loss: 0.0013 - accuracy: 1.0000 -
 val_loss: 0.6333 - val_accuracy: 0.8500
 Epoch 20/20
 16/16 [=====] - 2s 111ms/step - loss: 0.0011 - accuracy: 1.0000 -
 val_loss: 0.6541 - val_accuracy: 0.8500



4/4 [=====] - 0s 26ms/step
 Text(89.18, 0.5, 'Actual label')

Actual label	Colin Powell	16	2	0	0	0
	Donald Rumsfeld	4	17	4	0	0
	George W Bush	0	1	16	0	0
	Gerhard Schroeder	0	0	0	16	0
	Tony Blair	0	0	0	4	20
Predicted label		Colin Powell	Donald Rumsfeld	George W Bush	Gerhard Schroeder	Tony Blair

1/1 [=====] - 0s 48ms/step
 Colin Powell: 0.20101844
 Donald Rumsfeld: 0.20214622
 George W Bush: 0.2216323
 Gerhard Schroeder: 0.21147959
 Tony Blair: 0.16372345



<matplotlib.image.AxesImage at 0x1ec80d4d910>

Result:

Thus written a python program to implemented successfully for the Face recognition using CNN.

Ex.No: 4 LANGUAGE MODELING USING RNN

Aim:

To write a python program to implement the Language modeling using RNN.

Algorithm:

Step 1: Start the program and import the required Python libraries such as Future, Glob, OS, UnicodeData, String, PyTorch, Matplotlib, Random, Time, and Math.

Step 2: Define the set of valid characters consisting of uppercase letters, lowercase letters, spaces, punctuation symbols, and an **End-of-Sequence (EOS)** marker.

Step 3: Create a function to locate all text files containing names from different language categories.

Step 4: Define a function to convert Unicode characters into plain ASCII format by removing accent marks and unsupported symbols.

Step 5: Read each text file containing names and convert every name into ASCII format.

Step 6: Create a dictionary that stores:

- **Category name** (language)
- **List of names** belonging to that category.

Step 7: Store all available language categories and determine the total number of categories.

Step 8: Verify that the dataset is available. If no data is found, terminate the program with an appropriate error message.

Step 9: Display the total number of language categories and their corresponding names.

Step 10: Import the PyTorch deep learning library and define a custom **Recurrent Neural Network (RNN)** model.

Step 11: Initialize the RNN by defining:

- Input layer
- Hidden layer
- Output layer
- Dropout layer
- LogSoftmax activation function.

Step 12: Create the hidden state initialization function by assigning all hidden layer values to zero.

Step 13: Define the **Forward Propagation** process of the RNN by:

- Combining category information, current input character, and previous hidden state.
- Computing the new hidden state.
- Generating the output probabilities.
- Applying dropout to reduce overfitting.
- Applying the LogSoftmax activation function to obtain output probabilities.

Step 14: Create a function to randomly select a language category from the dataset.

Step 15: Randomly select a name belonging to the chosen language category.

Step 16: Generate a random training pair consisting of:

- Language category
- Corresponding name.

Step 17: Convert the selected language category into a **One-Hot Encoded** vector.

Step 18: Convert every character of the selected name into a sequence of one-hot encoded vectors representing the input sequence.

Step 19: Generate the target sequence by shifting the input characters by one position and appending the **End-of-Sequence (EOS)** marker.

Step 20: Create the complete training example consisting of:

- Category tensor
- Input tensor

- Target tensor.

Step 21: Define the **Negative Log-Likelihood (NLL) Loss** function to measure prediction error.

Step 22: Specify the learning rate used for updating the network parameters during training.

Step 23: Initialize the hidden state before processing each training example.

Step 24: Reset all gradients before starting the training process.

Step 25: Feed each character of the input name sequentially into the RNN.

Step 26: For every character:

- Perform forward propagation.
- Predict the next character.
- Compute the loss by comparing the predicted character with the target character.

Step 27: Accumulate the total loss over the complete character sequence.

Step 28: Perform **Backpropagation Through Time (BPTT)** to compute the gradients of all trainable parameters.

Step 29: Update the weights and biases using **Gradient Descent** based on the computed gradients and learning rate.

Step 30: Return the average loss for the processed training example.

Step 31: Record the training start time to monitor the execution duration.

Step 32: Initialize the RNN model with:

- Input size
- Hidden layer size
- Output size.

Step 33: Specify the total number of training iterations, printing interval, and plotting interval.

Step 34: Repeat the training process for the specified number of iterations.

Step 35: During each iteration:

- Generate a random training example.
- Train the RNN.
- Accumulate the training loss.

Step 36: Display the current iteration number, elapsed training time, percentage of completion, and average loss at regular intervals.

Step 37: Store the average training loss periodically for visualization.

Step 38: Plot the **Training Loss versus Iterations** graph to observe the learning performance of the RNN.

Step 39: Specify the maximum length of the generated name.

Step 40: Define the sampling function to generate a new name by:

- Selecting a language category.
- Providing an initial starting letter.
- Initializing the hidden state.

Step 41: Generate one character at a time by repeatedly:

- Performing forward propagation.
- Selecting the character with the highest prediction probability.
- Appending the predicted character to the generated name.
- Stopping when the **End-of-Sequence (EOS)** marker is predicted or the maximum length is reached.

Step 42: Create a function to generate multiple sample names for a selected language using different starting letters.

Step 43: Generate sample names for different language categories such as:

- Russian

- German
- Spanish
- Chinese

using the trained RNN model.

Step 44: Display the generated names for each language category.

Step 45: End the program.

Program:

```
from future import unicode_literals, print_function, division
from io import open
import glob
import os
import unicodedata
import string
all_letters = string.ascii_letters + ".,;'-"
n_letters = len(all_letters) + 1 # Plus EOS marker
def findFiles(path): return glob.glob(path)
# Turn a Unicode string to plain ASCII, thanks to
https://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return "".join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )
# Read a file and split into lines
def readLines(filename):
    with open(filename, encoding='utf-8') as some_file:
        return [unicodeToAscii(line.strip()) for line in some_file]
# Build the category_lines dictionary, a list of lines per category
category_lines = {}
all_categories = []
for filename in findFiles('data/names/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
    all_categories.append(category)
    lines = readLines(filename)
    category_lines[category] = lines
n_categories = len(all_categories)
if n_categories == 0:
    raise RuntimeError('Data not found. Make sure that you downloaded data '
        'from https://download.pytorch.org/tutorial/data.zip and extract it to '
        'the current directory.')
print('# categories:', n_categories, all_categories)
print(unicodeToAscii("O'Néàl"))
pip install torch
import torch
import torch.nn as nn
```

```

class RNN(nn.Module):
def init (self, input_size, hidden_size, output_size):
super(RNN, self). init ()
self.hidden_size = hidden_size
self.i2h = nn.Linear(n_categories + input_size + hidden_size, hidden_size)
self.i2o = nn.Linear(n_categories + input_size + hidden_size, output_size)
self.o2o = nn.Linear(hidden_size + output_size, output_size)
self.dropout = nn.Dropout(0.1)
self.softmax = nn.LogSoftmax(dim=1)
def forward(self, category, input, hidden):
input_combined = torch.cat((category, input, hidden), 1)
hidden = self.i2h(input_combined)
output = self.i2o(input_combined)
output_combined = torch.cat((hidden, output), 1)
output = self.o2o(output_combined)
output = self.dropout(output)
output = self.softmax(output)
return output, hidden
def initHidden(self):
return torch.zeros(1, self.hidden_size)
import random
# Random item from a list
def randomChoice(l):
return l[random.randint(0, len(l) - 1)]
# Get a random category and random line from that category
def randomTrainingPair():
category = randomChoice(all_categories)
line = randomChoice(category_lines[category])
return category, line
# One-hot vector for category
def categoryTensor(category):
li = all_categories.index(category)
tensor = torch.zeros(1, n_categories)
tensor[0][li] = 1
return tensor
# One-hot matrix of first to last letters (not including EOS) for input
def inputTensor(line):
tensor = torch.zeros(len(line), 1, n_letters)
for li in range(len(line)):
letter = line[li]
tensor[li][0][all_letters.find(letter)] = 1
return tensor
# ``LongTensor`` of second letter to end (EOS) for target
def targetTensor(line):
letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
letter_indexes.append(n_letters - 1) # EOS

```

```

return torch.LongTensor(letter_indexes)
# Make category, input, and target tensors from a random category, line pair
def randomTrainingExample():
    category, line = randomTrainingPair()
    category_tensor = categoryTensor(category)
    input_line_tensor = inputTensor(line)
    target_line_tensor = targetTensor(line)
    return category_tensor, input_line_tensor, target_line_tensor
criterion = nn.NLLLoss()
learning_rate = 0.0005
def train(category_tensor, input_line_tensor, target_line_tensor):
    target_line_tensor.unsqueeze_(-1)
    hidden = rnn.initHidden()
    rnn.zero_grad()
    loss = 0
    for i in range(input_line_tensor.size(0)):
        output, hidden = rnn(category_tensor, input_line_tensor[i], hidden)
        l = criterion(output, target_line_tensor[i])
        loss += l
    loss.backward()
    for p in rnn.parameters():
        p.data.add_(p.grad.data, alpha=-learning_rate)
    return output, loss.item() / input_line_tensor.size(0)
import time
import math
def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)
rnn = RNN(n_letters, 128, n_letters)
n_iters = 100000
print_every = 5000
plot_every = 500
all_losses = []
total_loss = 0 # Reset every ``plot_every`` ``iters``
start = time.time()
for iter in range(1, n_iters + 1):
    output, loss = train(*randomTrainingExample())
    total_loss += loss
    if iter % print_every == 0:
        print('%s (%d %d%%) %.4f' % (timeSince(start), iter, iter / n_iters * 100, loss))
    if iter % plot_every == 0:
        all_losses.append(total_loss / plot_every)
        total_loss = 0

```

```

import matplotlib.pyplot as plt
plt.figure()
plt.plot(all_losses)
max_length = 20
# Sample from a category and starting letter
def sample(category, start_letter='A'):
    with torch.no_grad(): # no need to track history in sampling
        category_tensor = categoryTensor(category)
        input = inputTensor(start_letter)
        hidden = rnn.initHidden()
        output_name = start_letter
        for i in range(max_length):
            output, hidden = rnn(category_tensor, input[0], hidden)
            topv, topi = output.topk(1)
            topi = topi[0][0]
            if topi == n_letters - 1:
                break
            else:
                letter = all_letters[topi]
                output_name += letter
                input = inputTensor(letter)
        return output_name
# Get multiple samples from one category and multiple starting letters
def samples(category, start_letters='ABC'):
    for start_letter in start_letters:
        print(sample(category, start_letter))
samples('Russian', 'RUS')
samples('German', 'GER')
samples('Spanish', 'SPA')
samples('Chinese', 'CHI')

```

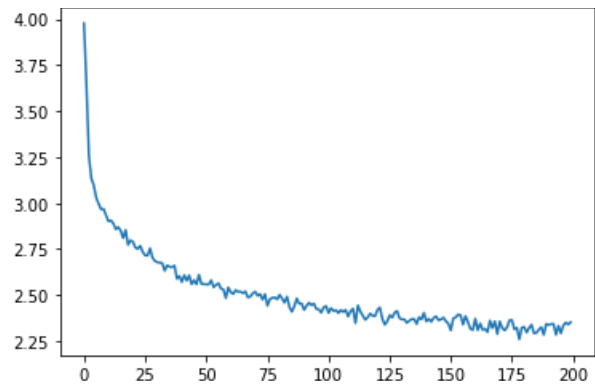
Output:

```

# categories: 18 ['Arabic', 'Chinese', 'Czech', 'Dutch',
'English', 'French', 'German', 'Greek', 'Irish', 'Italian',
', 'Japanese', 'Korean', 'Polish', 'Portuguese', 'Russian',
', 'Scottish', 'Spanish', 'Vietnamese']
O'Neal
0m 5s (5000 5%) 2.6595
0m 11s (10000 10%) 2.9644
0m 16s (15000 15%) 3.3754
0m 22s (20000 20%) 2.0799
0m 27s (25000 25%) 2.6884
0m 33s (30000 30%) 2.2509
0m 38s (35000 35%) 2.3497
0m 43s (40000 40%) 2.5290
0m 49s (45000 45%) 2.9439
0m 54s (50000 50%) 2.7406

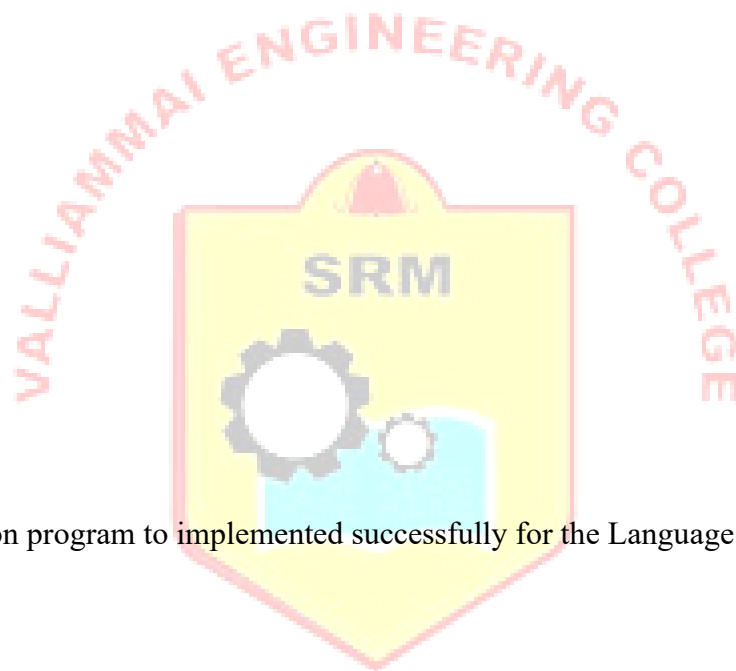
```

0m 59s (55000 55%) 3.0044
 1m 4s (60000 60%) 2.5765
 1m 10s (65000 65%) 2.3694
 1m 15s (70000 70%) 2.2810
 1m 20s (75000 75%) 2.2660
 1m 26s (80000 80%) 2.1720
 1m 31s (85000 85%) 2.4900
 1m 36s (90000 90%) 2.0302
 1m 42s (95000 95%) 1.8320
 1m 47s (100000 100%) 2.4904



[<matplotlib.lines.Line2D at 0x1e56757bcd0>]

Rovonov
 Uarakov
 Shavanov
 Gerre
 Eeren
 Roure
 Salla
 Para
 Allana
 Cha
 Han
 Iun



Result:

Thus written a python program to implemented successfully for the Language Modeling Using RNN.

Ex.No: 5 SENTIMENT ANALYSIS USING LSTM

Aim:

To write a python program to implement the Sentiment analysis using LSTM.

Algorithm:

Step 1: Start the program and install the required package **Keras-Preprocessing**.

Step 2: Import the required Python libraries such as Pandas, NumPy, Regular Expressions (Regex), NLTK, Scikit-learn, and Keras.

Step 3: Load the **IMDB Movie Reviews Dataset** from the CSV file into a Pandas DataFrame.

Step 4: Display the dataset to verify that the reviews and sentiment labels have been loaded successfully.

Step 5: Define a preprocessing function to clean the review text by:

- Removing HTML tags.
- Removing URLs.
- Removing non-alphanumeric characters.
- Converting all text into lowercase.

Step 6: Apply the preprocessing function to every movie review in the dataset.

Step 7: Download the **English Stopwords** corpus from the NLTK library.

Step 8: Remove common stopwords (such as *is, the, and, of*, etc.) from each review to reduce unnecessary words.

Step 9: Download the required NLTK resources such as **WordNet** and **OMW-1.4** for lemmatization.

Step 10: Initialize the **Whitespace Tokenizer** and **WordNet Lemmatizer**.

Step 11: Perform **Lemmatization** by converting each word into its base/root form while preserving its meaning.

Step 12: Apply lemmatization to every review in the dataset.

Step 13: Calculate the average number of words present in each movie review.

Step 14: Compute the total number and percentage of:

- Positive reviews.
- Negative reviews.

Step 15: Separate the dataset into:

- **Input Data:** Movie reviews.
- **Output Labels:** Sentiment (Positive/Negative).

Step 16: Encode the sentiment labels into numerical form using the **Label Encoder**.

Step 17: Split the dataset into **training** and **testing** datasets while maintaining equal class distribution.

Step 18: Define the model hyperparameters including:

- Vocabulary size.
- Out-of-Vocabulary (OOV) token.
- Embedding dimension.
- Maximum review length.
- Padding type.
- Truncation type.

Step 19: Initialize the **Tokenizer** to build the vocabulary from the training reviews.

Step 20: Fit the tokenizer on the training dataset to generate the word index.

Step 21: Convert each training review into a sequence of integer tokens.

Step 22: Pad all training sequences to a fixed length using **post-padding** to ensure equal input size.

Step 23: Convert the testing reviews into token sequences and apply the same padding process.

Step 24: Initialize a **Sequential Deep Learning Model**.

Step 25: Add an **Embedding Layer** to transform word indices into dense vector representations.

Step 26: Add a **Bidirectional Long Short-Term Memory (Bi-LSTM)** layer to learn contextual

information from both forward and backward directions of the review text.

Step 27: Add a fully connected (**Dense**) hidden layer with **ReLU activation** to perform feature learning.

Step 28: Add the output layer with:

- **One neuron**
- **Sigmoid activation function**
for binary sentiment classification.

Step 29: Compile the Bi-LSTM model using:

- **Optimizer:** Adam
- **Loss Function:** Binary Cross-Entropy
- **Evaluation Metric:** Accuracy

Step 30: Display the complete architecture and summary of the Bi-LSTM model.

Step 31: Train the model using the padded training dataset for the specified number of epochs.

Step 32: Validate the model using a portion of the training dataset during every epoch.

Step 33: Use the trained model to predict the sentiment of the testing dataset.

Step 34: Convert the predicted probability values into class labels by applying a threshold:

- Probability $\geq 0.5 \rightarrow$ Positive.
- Probability $< 0.5 \rightarrow$ Negative.

Step 35: Calculate the classification accuracy by comparing the predicted labels with the actual testing labels.

Step 36: Display the overall prediction accuracy of the trained Bi-LSTM model.

Step 37: Provide new movie review sentences as input for sentiment prediction.

Step 38: Convert the new review sentences into token sequences using the trained tokenizer.

Step 39: Pad the token sequences to match the input size expected by the Bi-LSTM model.

Step 40: Predict the sentiment probabilities for the new review sentences using the trained model.

Step 41: Convert the predicted probabilities into sentiment labels (Positive or Negative) using the threshold value.

Step 42: Display each input review along with its predicted sentiment.

Step 43: End the program.

Program:

```
pip install Keras-Preprocessing
import re
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from keras.preprocessing.text import Tokenizer
from keras_preprocessing.sequence import pad_sequences
import keras
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
import math
import nltk
data = pd.read_csv('IMDB Dataset.csv')
data
def remove_tags(string):
    removelist = ""
```

```

result = re.sub(",",string) #remove HTML tags
result = re.sub('https://.*',",result) #remove URLs
result = re.sub(r'[^\w'+removelist+']', ' ',result) #remove non-alphanumeric
characters
result = result.lower()
return result
data['review']=data['review'].apply(lambda cw : remove_tags(cw))
nltk.download('stopwords')
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
data['review'] = data['review'].apply(lambda x: ' '.join([word for word in x.split() if
word not in (stop_words)]))
import nltk
nltk.download()
#we want to download 'wordnet' and 'omw-1.4' from nltk
w_tokenizer = nltk.tokenize.WhitespaceTokenizer()
lemmatizer = nltk.stem.WordNetLemmatizer()
def lemmatize_text(text):
st = ""
for w in w_tokenizer.tokenize(text):
st = st + lemmatizer.lemmatize(w) + " "
return st
data['review'] = data['review'].apply(lemmatize_text)
data
s = 0.0
for i in data['review']:
word_list = i.split()
s = s + len(word_list)
print("Average length of each review : ",s/data.shape[0])
pos = 0
for i in range(data.shape[0]):
if data.iloc[i]['sentiment'] == 'positive':
pos = pos + 1
neg = data.shape[0]-pos
print("Percentage of reviews with positive sentiment is
"+str(pos/data.shape[0]*100)+"%")
print("Percentage of reviews with negative sentiment is
"+str(neg/data.shape[0]*100)+"%")
reviews = data['review'].values
labels = data['sentiment'].values
encoder = LabelEncoder()
encoded_labels = encoder.fit_transform(labels)
train_sentences, test_sentences, train_labels, test_labels = train_test_split(reviews,
encoded_labels, stratify = encoded_labels)
# Hyperparameters of the model
vocab_size = 3000 # choose based on statistics

```

```

oov_tok = "
embedding_dim = 100
max_length = 200 # choose based on statistics, for example 150 to 200
padding_type='post'
trunc_type='post'
# tokenize sentences
tokenizer = Tokenizer(num_words = vocab_size, oov_token=oov_tok)
tokenizer.fit_on_texts(train_sentences)
word_index = tokenizer.word_index
# convert train dataset to sequence and pad sequences
train_sequences = tokenizer.texts_to_sequences(train_sentences)
train_padded = pad_sequences(train_sequences, padding='post', maxlen=max_length)
# convert Test dataset to sequence and pad sequences
test_sequences = tokenizer.texts_to_sequences(test_sentences)
test_padded = pad_sequences(test_sequences, padding='post', maxlen=max_length)
# model initialization
model = keras.Sequential([
keras.layers.Embedding(vocab_size, embedding_dim,
input_length=max_length),
keras.layers.Bidirectional(keras.layers.LSTM(64)),
keras.layers.Dense(24, activation='relu'),
keras.layers.Dense(1, activation='sigmoid')
])
# compile model
model.compile(loss='binary_crossentropy',
optimizer='adam',
metrics=['accuracy'])
# model summary
model.summary()
num_epochs = 5
history = model.fit(train_padded, train_labels,
epochs=num_epochs, verbose=1,
validation_split=0.1)
prediction = model.predict(test_padded)
# Get labels based on probability 1 if p>= 0.5 else 0
pred_labels = []
for i in prediction:
if i >= 0.5:
pred_labels.append(1)
else:
pred_labels.append(0)
print("Accuracy of prediction on test set : ",
accuracy_score(test_labels,pred_labels))
# reviews on which we need to predict
sentence = ["The movie was very touching and heart whelming",
"I have never seen a terrible movie like this",

```

```

"the movie plot is terrible but it had good acting"]
# convert to a sequence
sequences = tokenizer.texts_to_sequences(sentence)
# pad the sequence
padded = pad_sequences(sequences, padding='post', maxlen=max_length)
# Get labels based on probability 1 if p>= 0.5 else 0
prediction = model.predict(padded)
pred_labels = []
for i in prediction:
    if i >= 0.5:
        pred_labels.append(1)
    else:
        pred_labels.append(0)
for i in range(len(sentence)):
    print(sentence[i])
    if pred_labels[i] == 1:
        s = 'Positive'
    else:
        s = 'Negative'
    print("Predicted sentiment :",s)

```

Output:

```

review sentiment
0 One of the other reviewers has mentioned that ... positive
1 A wonderful little production. <br /><br />The... positive
2 I thought this was a wonderful way to spend ti... positive
3 Basically there's a family where a little boy ... negative
4 Petter Mattei's "Love in the Time of Money" is... positive
... ..
49995 I thought this movie did a down right good job... positive
49996 Bad plot, bad dialogue, bad acting, idiotic di... negative
49997 I am a Catholic taught in parochial elementary... negative
49998 I'm going to have to disagree with the previou... negative
49999 No one expects the Star Trek movies to be high... negative
[nltk_data] Downloading package stopwords to
[nltk_data] C:\Users\CGNANAM.ADS\AppData\Roaming\nltk_data...
[nltk_data] Package stopwords is already up-to-date!

```

```

showing info https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/index.xml
Out[6]:
True

```

review sentiment

0

W W W W W W W W W W W W W W W W W W W W W W ...

positive

1

W W W W W W W W W W W W W W W W

positive

2

W W W W W W W W W W W W W W W W W W

positive

3

W W W W W W W W W W W

negative

4

W W W W W W W W W W W W W W W W W

positive

review sentiment

...

...

...

49995

W W W W W W W W W W W W W W W W W W

positive

49996



W W W W W W W W

negative

49997

W W W W W W W W W W W W

negative

49998

W W W W W W W W W W W W W W

negative

49999

W W W W W W W W W W W W

negative

50000 rows $\times$ 2 columns

Average length of each review : 18.714

Percentage of reviews with positive sentiment is 50.0%

Percentage of reviews with negative sentiment is 50.0%

Model: "sequential"

Layer (type) Output Shape Param #

embedding (Embedding) (None, 200, 100) 300000

bidirectional (Bidirectiona (None, 128) 84480

l)

dense (Dense) (None, 24) 3096

dense_1 (Dense) (None, 1) 25

Total params: 387,601

Trainable params: 387,601

Non-trainable params: 0

Epoch 1/5

1055/1055 [=====] - 60s 55ms/step - loss: 0.69

32 - accuracy: 0.5021 - val_loss: 0.6925 - val_accuracy: 0.5205

Epoch 2/5

1055/1055 [=====] - 58s 55ms/step - loss: 0.69

26 - accuracy: 0.5094 - val_loss: 0.6925 - val_accuracy: 0.5173

Epoch 3/5

1055/1055 [=====] - 59s 56ms/step - loss: 0.69

26 - accuracy: 0.5129 - val_loss: 0.6924 - val_accuracy: 0.5171

Epoch 4/5

1055/1055 [=====] - 59s 56ms/step - loss: 0.69

23 - accuracy: 0.5166 - val_loss: 0.6927 - val_accuracy: 0.4965

Epoch 5/5

1055/1055 [=====] - 58s 55ms/step - loss: 0.69

25 - accuracy: 0.5141 - val_loss: 0.6924 -

391/391 [=====] -

val_accuracy: 0.5173

6s 14ms/step

Accuracy of prediction on test set : 0.5148

1/1 [=====] - 0s 23ms/step

The movie was very touching and heart whelming

Predicted sentiment : Positive

I have never seen a terrible movie like this

Predicted sentiment : Positive

the movie plot is terrible but it had good acting

Predicted sentiment : Positive

Result:

Thus written a python program to implemented successfully for the Sentiment analysis using LSTM.

Ex.No : 6 PARTS OF SPEECH TAGGING USING SEQUENCE TO SEQUENCE ARCHITECTURE

Aim:

To implement Parts of speech tagging using Sequence to Sequence architecture.

Algorithm:

Step 1: Start the program and import the required Python libraries such as NumPy, TensorFlow, Keras, and sequence preprocessing utilities.

Step 2: Define the input text sentences and their corresponding **Part-of-Speech (POS)** tag sequences.

Step 3: Create two sets to store:

- All unique words present in the input sentences.
- All unique POS tags present in the target sequences.

Step 4: Traverse all input and target sentence pairs to collect the complete vocabulary of words and POS tags.

Step 5: Add the special **Start-of-Sequence (SOS)** and **End-of-Sequence (EOS)** tokens to the target POS tag vocabulary.

Step 6: Create dictionaries to map:

- Input words to unique integer indices.
- Integer indices back to input words.
- POS tags to unique integer indices.
- Integer indices back to POS tags.

Step 7: Determine the maximum sequence length of:

- Input (encoder) sentences.
- Output (decoder) POS tag sequences.

Step 8: Initialize the encoder input matrix with zeros based on the number of input sentences and maximum encoder sequence length.

Step 9: Convert each input sentence into its corresponding integer sequence using the input word dictionary.

Step 10: Store the encoded input sequences in the encoder input matrix.

Step 11: Initialize the decoder input matrix and decoder target tensor with appropriate dimensions.

Step 12: Convert each target POS tag sequence into integer values using the target tag dictionary.

Step 13: Prepare the decoder input by storing the current POS tag at each time step.

Step 14: Generate the decoder target data using **One-Hot Encoding**, where each target POS tag is represented as a binary vector.

Step 15: Define the **Encoder Input Layer** to receive the input word sequences.

Step 16: Add an **Embedding Layer** to convert input word indices into dense vector representations.

Step 17: Add an **LSTM Encoder Layer** to process the embedded input sequence and generate:

- Hidden state.
- Cell state.

Step 18: Store the encoder hidden state and cell state to initialize the decoder.

Step 19: Define the **Decoder Input Layer** for receiving the target sequence during training.

Step 20: Add an **Embedding Layer** for the decoder input sequence.

Step 21: Add an **LSTM Decoder Layer** that uses the encoder hidden and cell states as its initial state.

Step 22: Configure the decoder to return:

- Output sequence.
- Updated hidden state.
- Updated cell state.

Step 23: Add a **Dense Layer** with **Softmax activation** to predict the probability distribution of each POS

tag.

Step 24: Construct the complete **Encoder–Decoder Sequence-to-Sequence (Seq2Seq)** model by connecting the encoder and decoder components.

Step 25: Compile the model using:

- **Optimizer:** Adam
- **Loss Function:** Categorical Cross-Entropy
- **Evaluation Metric:** Accuracy

Step 26: Train the Seq2Seq model using the prepared encoder input data, decoder input data, and decoder target data for the specified number of epochs.

Step 27: Validate the model performance using a validation split during training.

Step 28: Create a separate **Encoder Model** that outputs only the encoder hidden state and cell state during inference.

Step 29: Create a separate **Decoder Model** that accepts:

- Previous decoder input.
- Previous hidden state.
- Previous cell state.

and generates:

- Predicted POS tag.
- Updated hidden state.
- Updated cell state.

Step 30: Define an inference function to generate POS tags for unseen input sentences.

Step 31: Encode the input sentence using the trained encoder model to obtain the initial hidden and cell states.

Step 32: Initialize the decoder input with the **Start-of-Sequence (SOS)** token.

Step 33: Repeatedly perform the following steps until the **End-of-Sequence (EOS)** token is predicted or the maximum sequence length is reached:

- Feed the current decoder input and encoder states into the decoder.
- Predict the next POS tag.
- Select the POS tag with the highest probability.
- Append the predicted tag to the output sequence.
- Update the decoder input with the predicted tag.
- Update the hidden and cell states for the next prediction.

Step 34: Combine all predicted POS tags to form the complete output sequence.

Step 35: Convert each test sentence into its corresponding integer sequence using the input word dictionary.

Step 36: Pad the input sequence to match the maximum encoder sequence length.

Step 37: Pass the padded sequence to the inference function to generate the predicted POS tag sequence.

Step 38: Display the original input sentence along with its predicted POS tags.

Step 39: Repeat the prediction process for all test sentences.

Step 40: End the program.

Program:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, LSTM, Dense
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

```

# Define the input and output sequences
input_texts = ['I love coding', 'This is a pen', 'She sings well']
target_texts = ['PRP VB NNP', 'DT VBZ DT NN', 'PRP VBZ RB']
# Create a set of all unique words and POS tags in the
dataset input_words = set()
target_words = set()
for input_text, target_text in zip(input_texts,
target_texts): input_words.update(input_text.split())
target_words.update(target_text.split())
# Add <sos> and <eos> tokens to target_words
target_words.add('<sos>')
target_words.add('<eos>')
# Create dictionaries to map words and POS tags to integers
input_word2idx = {word: idx for idx, word in
enumerate(input_words)} input_idx2word = {idx: word for idx, word
in enumerate(input_words)} target_word2idx = {word: idx for idx,
word in enumerate(target_words)} target_idx2word = {idx: word for
idx, word in enumerate(target_words)}
# Define the maximum sequence lengths
max_encoder_seq_length = max([len(text.split()) for text in input_texts])
max_decoder_seq_length = max([len(text.split()) for text in target_texts])
# Prepare the encoder input data
encoder_input_data = np.zeros((len(input_texts),
max_encoder_seq_length), dtype='float32')
for i, input_text in enumerate(input_texts):
for t, word in enumerate(input_text.split()):
encoder_input_data[i, t] =
input_word2idx[word]
# Prepare the decoder input and target data
decoder_input_data = np.zeros((len(input_texts),
max_decoder_seq_length), dtype='float32')
decoder_target_data = np.zeros((len(input_texts),
max_decoder_seq_length, len(target_words)), dtype='float32')
for i, target_text in
enumerate(target_texts): for t, word in
enumerate(target_text.split()):
decoder_input_data[i, t] =
target_word2idx[word] if t > 0:
decoder_target_data[i, t - 1, target_word2idx[word]] = 1.0
# Define the encoder input and LSTM layers
encoder_inputs = Input(shape=(None,))
encoder_embedding = tf.keras.layers.Embedding(len(input_words),
256)(encoder_inputs)
encoder_lstm = LSTM(256, return_state=True)
encoder_outputs, state_h, state_c = encoder_lstm(encoder_embedding)
encoder_states = [state_h, state_c]

```

```

# Define the decoder input and LSTM layers
decoder_inputs = Input(shape=(None,))
decoder_embedding = tf.keras.layers.Embedding(len(target_words),
256)(decoder_inputs)
decoder_lstm = LSTM(256, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_embedding,
initial_state=encoder_states)
decoder_dense = Dense(len(target_words), activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
# Define the model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
# Compile and train the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.fit([encoder_input_data, decoder_input_data],
decoder_target_data, batch_size=64, epochs=50,
validation_split=0.2)
# Define the encoder model to get the encoder states
encoder_model = Model(encoder_inputs,
encoder_states)
# Define the decoder model with encoder states as initial
state decoder_state_input_h = Input(shape=(256,))
decoder_state_input_c = Input(shape=(256,))
decoder_states_inputs = [decoder_state_input_h,
decoder_state_input_c] decoder_outputs, state_h, state_c =
decoder_lstm(decoder_embedding, initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs =
decoder_dense(decoder_outputs)
decoder_model = Model([decoder_inputs] + decoder_states_inputs,
[decoder_outputs]
+ decoder_states)
# Define a function to perform inference and generate POS
tags def generate_pos_tags(input_sequence):
states_value =
encoder_model.predict(input_sequence)
target_sequence = np.zeros((1, 1))
target_sequence[0, 0] =
target_word2idx['<sos>'] stop_condition = False
pos_tags = []
while not stop_condition:
output_tokens, h, c =
decoder_model.predict([target_sequence] + states_value)
sampled_token_index = np.argmax(output_tokens[0, -
1, :]) sampled_word =
target_idx2word[sampled_token_index]

```

```

pos_tags.append(sampled_word)
if sampled_word == '<eos>' or len(pos_tags) >
max_decoder_seq_length: stop_condition = True
target_sequence = np.zeros((1, 1))
target_sequence[0, 0] =
sampled_token_index states_value = [h,
c]
return ''.join(pos_tags)
# Test the model
for input_text in input_texts:
input_seq = pad_sequences([[input_word2idx[word] for
word in input_text.split()]],
maxlen=max_encoder_seq_length)
predicted_pos_tags =
generate_pos_tags(input_seq) print('Input:',
input_text)
print('Predicted POS Tags:',
predicted_pos_tags) print()

```

Output:

```

Epoch 1/50
1/1 [=====] - 7s 7s/step - loss: 1.3736 - accuracy: 0.0000e+0
0 - val_loss: 1.1017 - val_accuracy: 0.0000e+00
Epoch 2/50
1/1 [=====] - 0s 63ms/step - loss: 1.3470 - accuracy: 0.7500 -
val_loss: 1.1068 - val_accuracy: 0.0000e+00
Epoch 3/50
1/1 [=====] - 0s 65ms/step - loss: 1.3199 - accuracy: 0.7500
- val_loss: 1.1123 - val_accuracy: 0.0000e+00
Epoch 4/50
Epoch 44/50
1/1 [=====] - 0s 58ms/step - loss: 0.0882 - accuracy: 0.7500 :
:
Epoch 50/50
1/1 [=====] - 0s 60ms/step - loss: 0.0751 - accuracy: 0.7500
- val_loss: 2.2554 - val_accuracy: 0.0000e+00
Input: I love coding
Predicted POS Tags: VB NNP NNP DT DT
Input: This is a pen
Predicted POS Tags: VBZ DT NN NN DT
Input: She sings well
Predicted POS Tags: VB NNP NNP DT DT

```

Result:

Thus Parts of speech tagging using Sequence to Sequence architecture is implemented.

Ex.No: 7 MACHINE TRANSLATION USING ENCODER-DECODER MODEL

Aim:

To implement Machine Translation using Encoder-Decoder model.

Algorithm:

Step 1: Start the program and import the required Python libraries such as NumPy, TensorFlow, Keras, and sequence preprocessing utilities.

Step 2: Define the source language (input) sentences and their corresponding target language (translated) sentences.

Step 3: Create separate sets to store all unique words present in:

- The input language vocabulary.
- The target language vocabulary.

Step 4: Traverse all input and target sentence pairs to collect every unique word from both languages.

Step 5: Add the special **Start-of-Sequence (SOS)** and **End-of-Sequence (EOS)** tokens to the target language vocabulary.

Step 6: Create dictionaries to map:

- Input words to unique integer indices.
- Integer indices back to input words.
- Target language words to unique integer indices.
- Integer indices back to target language words.

Step 7: Determine the maximum sequence length of:

- Input (encoder) sentences.
- Target (decoder) sentences.

Step 8: Initialize the encoder input matrix with zeros according to the number of input sentences and the maximum encoder sequence length.

Step 9: Convert every input sentence into a sequence of integer values using the input word dictionary.

Step 10: Store the encoded input sequences in the encoder input matrix.

Step 11: Initialize the decoder input matrix and decoder target tensor with appropriate dimensions.

Step 12: Convert each target language sentence into its corresponding integer sequence.

Step 13: Store the target word indices in the decoder input matrix.

Step 14: Generate the decoder target data by applying **One-Hot Encoding** to every target word for supervised learning.

Step 15: Define the **Encoder Input Layer** to receive the source language sentence.

Step 16: Add an **Embedding Layer** to convert input word indices into dense vector representations.

Step 17: Add an **LSTM Encoder Layer** to process the input sequence and generate:

- Hidden state.
- Cell state.

Step 18: Store the encoder hidden state and cell state to initialize the decoder.

Step 19: Define the **Decoder Input Layer** to receive the translated sentence during training.

Step 20: Add an **Embedding Layer** for the decoder input words.

Step 21: Add an **LSTM Decoder Layer** that receives the encoder hidden state and cell state as its initial state.

Step 22: Configure the decoder to return:

- Output sequence.
- Updated hidden state.
- Updated cell state.

Step 23: Add a **Dense Layer** with **Softmax activation** to predict the probability of every target language word.

Step 24: Construct the complete **Encoder–Decoder Sequence-to-Sequence (Seq2Seq)** neural network by connecting the encoder and decoder.

Step 25: Compile the model using:

- **Optimizer:** Adam
- **Loss Function:** Categorical Cross-Entropy
- **Evaluation Metric:** Accuracy

Step 26: Train the model using the prepared encoder input data, decoder input data, and decoder target data for the specified number of epochs.

Step 27: Validate the model during training using a validation dataset.

Step 28: Create a separate **Encoder Model** that outputs only the encoder hidden state and cell state for inference.

Step 29: Create a separate **Decoder Model** that accepts:

- Previous decoder input.
- Previous hidden state.
- Previous cell state.

and returns:

- Predicted target word.
- Updated hidden state.
- Updated cell state.

Step 30: Define a translation function to generate translated sentences from unseen input sequences.

Step 31: Encode the input sentence using the trained encoder model to obtain the initial hidden state and cell state.

Step 32: Initialize the decoder input with the **Start-of-Sequence (SOS)** token.

Step 33: Repeatedly perform the following steps until the **End-of-Sequence (EOS)** token is predicted or the maximum target sequence length is reached:

- Feed the current decoder input and encoder states into the decoder.
- Predict the next target language word.
- Select the word with the highest prediction probability.
- Append the predicted word to the translated sentence.
- Update the decoder input with the predicted word.
- Update the hidden state and cell state for the next prediction.

Step 34: Combine all predicted words to form the complete translated sentence.

Step 35: Convert each test input sentence into its corresponding integer sequence.

Step 36: Pad the input sequence to match the maximum encoder sequence length.

Step 37: Pass the padded sequence to the translation function to generate the translated sentence.

Step 38: Display the original input sentence along with its translated output.

Step 39: Repeat the translation process for all test sentences.

Step 40: End the program.

Program:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, LSTM, Dense
from tensorflow.keras.preprocessing.sequence import pad_sequences
# Define the input and output sequences
input_texts = ['I love coding', 'This is a pen', 'She sings well']
```

```

target_texts = ['Ich liebe das Coden', 'Das ist ein Stift', 'Sie singt
gut']
# Create a set of all unique words in the input and target
sequences input_words = set()
target_words = set()
for input_text, target_text in zip(input_texts, target_texts):
input_words.update(input_text.split())
target_words.update(target_text.split())
# Add <sos> and <eos> tokens to target_words
target_words.add('<sos>')
target_words.add('<eos>')
# Create dictionaries to map words to integers
input_word2idx = {word: idx for idx, word in
enumerate(input_words)} input_idx2word = {idx: word for idx, word
in enumerate(input_words)} target_word2idx = {word: idx for idx,
word in enumerate(target_words)} target_idx2word = {idx: word for
idx, word in enumerate(target_words)}
# Define the maximum sequence lengths
max_encoder_seq_length = max([len(text.split()) for text in input_texts])
max_decoder_seq_length = max([len(text.split()) for text in target_texts])
# Prepare the encoder input data
encoder_input_data = np.zeros((len(input_texts),
max_encoder_seq_length), dtype='float32')
for i, input_text in enumerate(input_texts):
for t, word in enumerate(input_text.split()):
encoder_input_data[i, t] =
input_word2idx[word]
# Prepare the decoder input and target data
decoder_input_data = np.zeros((len(input_texts),
max_decoder_seq_length), dtype='float32')
decoder_target_data = np.zeros((len(input_texts),
max_decoder_seq_length, len(target_words)), dtype='float32')
for i, target_text in
enumerate(target_texts): for t, word in
enumerate(target_text.split()):
decoder_input_data[i, t] =
target_word2idx[word] if t > 0:
decoder_target_data[i, t - 1, target_word2idx[word]] = 1.0
# Define the encoder input and LSTM layers
encoder_inputs = Input(shape=(None,))
encoder_embedding = tf.keras.layers.Embedding(len(input_words),
256)(encoder_inputs)
encoder_lstm = LSTM(256, return_state=True)
encoder_outputs, state_h, state_c = encoder_lstm(encoder_embedding)
encoder_states = [state_h, state_c]
# Define the decoder input and LSTM layers

```

```

decoder_inputs = Input(shape=(None,))
decoder_embedding = tf.keras.layers.Embedding(len(target_words),
256)(decoder_inputs)
decoder_lstm = LSTM(256, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_embedding,
initial_state=encoder_states)
decoder_dense = Dense(len(target_words), activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
# Define the model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
# Compile and train the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.fit([encoder_input_data, decoder_input_data],
decoder_target_data, batch_size=64, epochs=50,
validation_split=0.2)
# Define the encoder model to get the encoder states
encoder_model = Model(encoder_inputs,
encoder_states)
# Define the decoder model with encoder states as initial
state decoder_state_input_h = Input(shape=(256,))
decoder_state_input_c = Input(shape=(256,))
decoder_states_inputs = [decoder_state_input_h,
decoder_state_input_c] decoder_outputs, state_h, state_c =
decoder_lstm(decoder_embedding, initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs =
decoder_dense(decoder_outputs)
decoder_model = Model([decoder_inputs] + decoder_states_inputs,
[decoder_outputs]
+ decoder_states)
# Define a function to perform inference and generate
translations def translate(input_sequence):
states_value =
encoder_model.predict(input_sequence)
target_sequence = np.zeros((1, 1))
target_sequence[0, 0] =
target_word2idx['<sos>'] stop_condition = False
translation = []
while not stop_condition:
output_tokens, h, c =
decoder_model.predict([target_sequence] + states_value)
sampled_token_index = np.argmax(output_tokens[0, -
1, :]) sampled_word =
target_idx2word[sampled_token_index]
translation.append(sampled_word)

```

```

if sampled_word == '<eos>' or len(translation) >
max_decoder_seq_length: stop_condition = True
target_sequence = np.zeros((1, 1))
target_sequence[0, 0] =
sampled_token_index states_value = [h,
c]
return ' '.join(translation)
# Test the model
for input_text in input_texts:
input_seq = pad_sequences([[input_word2idx[word] for
word in input_text.split()]],
maxlen=max_encoder_seq_length)
translated_text = translate(input_seq)
print('Input:', input_text)
print('Translated Text:',
translated_text) print()

```

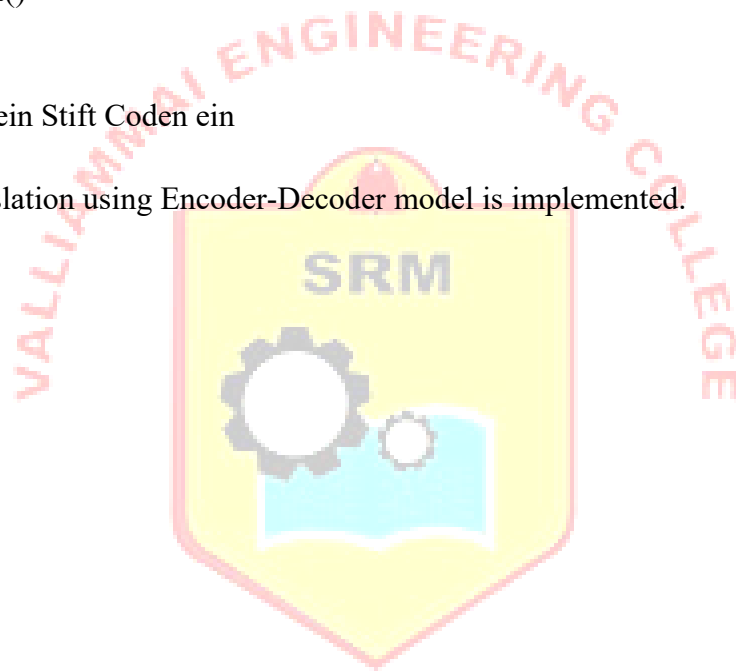
Output:

Input: This is a pen

Translated Text: ist ein Stift Coden ein

Result:

Thus Machine Translation using Encoder-Decoder model is implemented.



Ex.No: 8 IMAGE AUGMENTATION USING GANS

Aim:

To implement Image augmentation using GANs

Algorithm:

Step 1: Start the program and import the required Python libraries such as NumPy, Matplotlib, TensorFlow, Keras, and the necessary neural network layers.

Step 2: Load the **MNIST handwritten digit dataset** using the Keras dataset library.

Step 3: Normalize the pixel values of all training images to the range **-1 to +1** for stable GAN training.

Step 4: Expand the image dimensions to include a single channel, converting each image into the format **(28 × 28 × 1)**.

Generator Network

Step 5: Initialize the **Generator** model using the Sequential architecture.

Step 6: Create the input layer that accepts a **100-dimensional random noise vector (latent vector)**.

Step 7: Add a **Dense Layer** to transform the latent vector into a high-dimensional feature representation.

Step 8: Reshape the dense output into a **7 × 7 × 256** feature map.

Step 9: Apply **Batch Normalization** to stabilize training and accelerate convergence.

Step 10: Add the first **Transposed Convolution (Conv2DTranspose)** layer with ReLU activation to begin upsampling the feature maps.

Step 11: Apply another **Batch Normalization** layer to improve learning stability.

Step 12: Add a second **Transposed Convolution** layer with ReLU activation to further increase the image resolution.

Step 13: Apply another **Batch Normalization** layer.

Step 14: Add the final **Transposed Convolution** layer with **Tanh activation** to generate a grayscale handwritten digit image of size **28 × 28 × 1**.

Discriminator Network

Step 15: Initialize the **Discriminator** model using the Sequential architecture.

Step 16: Add the first **Convolutional Layer** with ReLU activation to extract features from the input image.

Step 17: Apply a **Dropout Layer** to reduce overfitting and improve generalization.

Step 18: Add the second **Convolutional Layer** to extract higher-level image features.

Step 19: Apply another **Dropout Layer** for regularization.

Step 20: Flatten the extracted feature maps into a one-dimensional feature vector.

Step 21: Add a **Dense Output Layer** with **Sigmoid activation** to classify the input image as **Real** or **Fake**.

GAN Model Construction

Step 22: Compile the **Discriminator** using:

- **Loss Function:** Binary Cross-Entropy
- **Optimizer:** Adam
- **Evaluation Metric:** Accuracy

Step 23: Create the **GAN model** by connecting the Generator output to the Discriminator input.

Step 24: Compile the complete GAN model using:

- **Loss Function:** Binary Cross-Entropy
- **Optimizer:** Adam

Training Phase

Step 25: Define the training hyperparameters such as:

- Number of epochs.
- Batch size.

- Sample display interval.

Step 26: Begin the training loop and repeat the following process for the specified number of epochs.

Step 27: Randomly select a batch of real handwritten digit images from the MNIST training dataset.

Step 28: Generate a batch of random noise vectors from a normal distribution.

Step 29: Pass the random noise vectors through the **Generator** to create a batch of fake handwritten digit images.

Step 30: Combine the real images and generated fake images into a single training batch.

Step 31: Assign labels:

- **1** for real images.
- **0** for fake images.

Step 32: Train the **Discriminator** using the combined dataset to distinguish between real and generated images.

Step 33: Generate another batch of random noise vectors.

Step 34: Train the **Generator** through the GAN model by assigning the generated images the label **1**, encouraging the Generator to fool the Discriminator.

Step 35: Update the Generator weights based on the discriminator feedback.

Step 36: Display the current training progress by printing:

- Epoch number.
- Discriminator loss.
- Generator loss.

Image Generation

Step 37: At regular intervals, generate a new set of random noise vectors.

Step 38: Pass the noise vectors through the trained Generator to produce new handwritten digit images.

Step 39: Convert the generated images from the normalized range back to standard grayscale pixel values.

Step 40: Reshape the generated output into 28×28 images.

Step 41: Arrange the generated images in a 4×4 grid using Matplotlib.

Step 42: Display the generated handwritten digit images for visual evaluation of the Generator's performance.

Step 43: Repeat the training and image generation process until all epochs are completed.

Step 44: End the program.

Program:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Input, Dense, Reshape,
Flatten from tensorflow.keras.layers import
BatchNormalization, Dropout from tensorflow.keras.layers
import Conv2D, Conv2DTranspose
from tensorflow.keras.models import Sequential,
Model from tensorflow.keras.optimizers import
Adam
# Load the MNIST dataset
(x_train, _), (_, _) = mnist.load_data()
# Normalize and reshape the images
x_train = (x_train.astype('float32') - 127.5) /
127.5 x_train = np.expand_dims(x_train, axis=-1)
```

```

# Define the generator
network generator =
Sequential()
generator.add(Dense(7 * 7 * 256, input_dim=100))
generator.add(Reshape((7, 7, 256)))
generator.add(BatchNormalization())
generator.add(Conv2DTranspose(128, kernel_size=5, strides=1,
padding='same', activation='relu'))
generator.add(BatchNormalization())
generator.add(Conv2DTranspose(64, kernel_size=5, strides=2, padding='same',
activation='relu'))
generator.add(BatchNormalization())
generator.add(Conv2DTranspose(1, kernel_size=5, strides=2, padding='same',
activation='tanh'))
# Define the discriminator
network discriminator =
Sequential()
discriminator.add(Conv2D(64, kernel_size=5, strides=2,
padding='same', input_shape=(28, 28, 1), activation='relu'))
discriminator.add(Dropout(0.3))
discriminator.add(Conv2D(128, kernel_size=5, strides=2,
padding='same', activation='relu'))
discriminator.add(Dropout(0.3))
discriminator.add(Flatten())
discriminator.add(Dense(1,
activation='sigmoid'))
# Compile the discriminator
discriminator.compile(loss='binary_crossentropy',
optimizer=Adam(learning_rate=0.0002, beta_1=0.5),
metrics=['accuracy'])
# Combine the generator and discriminator into a single GAN model
gan_input = Input(shape=(100,))
gan_output =
discriminator(generator(gan_input)) gan =
Model(gan_input, gan_output)
gan.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.0002,
beta_1=0.5))
# Training
hyperparameters epochs =
100
batch_size = 128
sample_interval = 10
# Training loop
for epoch in range(epochs):
# Randomly select a batch of real images
idx = np.random.randint(0, x_train.shape[0],

```

```

batch_size) real_images = x_train[idx]
# Generate a batch of fake images
noise = np.random.normal(0, 1, (batch_size,
100)) fake_images = generator.predict(noise)
# Train the discriminator
x = np.concatenate((real_images, fake_images))
y = np.concatenate((np.ones((batch_size, 1)),
np.zeros((batch_size, 1)))) d_loss =
discriminator.train_on_batch(x, y)
# Train the generator
noise = np.random.normal(0, 1, (batch_size, 100))
g_loss = gan.train_on_batch(noise, np.ones((batch_size, 1)))
# Print the progress and save
samples if epoch % sample_interval
== 0:
print(f'Epoch: {epoch} Discriminator Loss: {d_loss[0]} Generator Loss: {g_loss}')
samples = generator.predict(np.random.normal(0, 1, (16,
100))) samples = (samples * 127.5) + 127.5
samples = samples.reshape(16, 28,
28) fig, axs = plt.subplots(4, 4)
count = 0
for i in range(4):
for j in range(4):
axs[i, j].imshow(samples[count, :, :],
cmap='gray') axs[i, j].axis('off')
count += 1
plt.show()

```

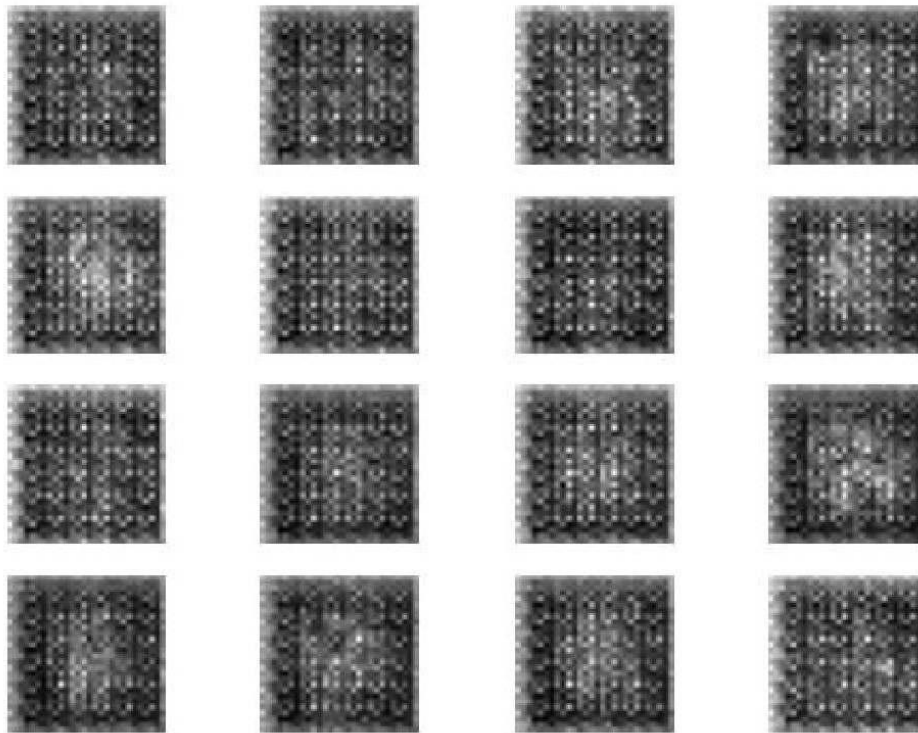
Output:

```

Epoch: 90 Discriminator Loss:
0.03508808836340904 Generator Loss:
1.736445483402349e-06

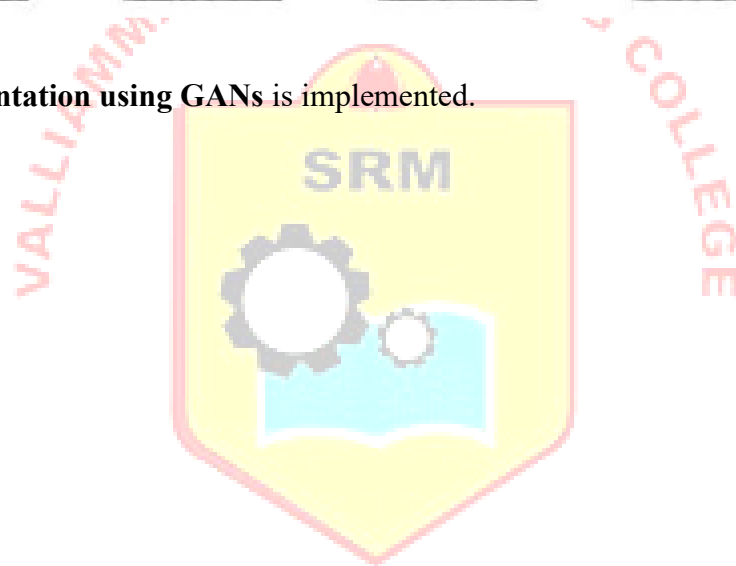
```





Result:

Thus **Image augmentation using GANs** is implemented.



Ex. No: 9 MINI PROJECT ON REAL WORLD APPLICATIONS

DEEPFAKE IMAGE DETECTION USING CONVOLUTIONAL NEURAL NETWORK (CNN)

Aim:

To identify AI-generated or manipulated images using a Convolutional Neural Network (CNN).

Objective

- To understand the concept of DeepFake image detection.
- To preprocess facial images for classification.
- To train a CNN model to distinguish between real and fake images.
- To evaluate the performance of the trained model.

Theory

DeepFake technology uses deep learning techniques such as Autoencoders and Generative Adversarial Networks (GANs) to generate highly realistic fake images and videos. Detecting DeepFake images is essential to prevent misinformation, identity theft, and digital fraud. A Convolutional Neural Network (CNN) can automatically learn image features and classify images as **Real** or **Fake** based on subtle visual artifacts and inconsistencies.

Algorithm

Step 1: Start the program and import the required libraries such as TensorFlow, Keras, NumPy, OpenCV, and Matplotlib.

Step 2: Load the DeepFake image dataset containing **Real** and **Fake** image folders.

Step 3: Resize all images to 128×128 pixels.

Step 4: Normalize the pixel values to the range 0–1.

Step 5: Assign class labels:

- Real $\rightarrow 0$
- Fake $\rightarrow 1$

Step 6: Split the dataset into training and testing sets.

Step 7: Initialize a Sequential CNN model.

Step 8: Add the first Convolution layer with ReLU activation.

Step 9: Add a Max Pooling layer.

Step 10: Add a second Convolution layer followed by another Max Pooling layer.

Step 11: Flatten the extracted feature maps.

Step 12: Add a Dense hidden layer with ReLU activation.

Step 13: Add a Dropout layer to reduce overfitting.

Step 14: Add the output layer with Sigmoid activation for binary classification.

Step 15: Compile the CNN model using:

- Optimizer: Adam
- Loss Function: Binary Cross-Entropy
- Metric: Accuracy

Step 16: Train the CNN model using the training dataset.

Step 17: Validate the model using the testing dataset.

Step 18: Save the trained model.

Step 19: Load a new test image.

Step 20: Preprocess the image by resizing and normalizing.

Step 21: Predict whether the image is **Real** or **Fake**.

Step 22: Display the predicted class with confidence score.

Step 23: End the program.

Program:

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.layers import Flatten, Dense, Dropout
import numpy as np

# Image preprocessing
train_datagen = ImageDataGenerator(
    rescale=1./255,
    validation_split=0.2
)

train_generator = train_datagen.flow_from_directory(
    'dataset/',
    target_size=(128,128),
    batch_size=32,
    class_mode='binary',
    subset='training'
)

validation_generator = train_datagen.flow_from_directory(
    'dataset/',
    target_size=(128,128),
    batch_size=32,
    class_mode='binary',
    subset='validation'
)

# CNN Model
model = Sequential()

model.add(Conv2D(32,(3,3),activation='relu',
    input_shape=(128,128,3)))
model.add(MaxPooling2D(2,2))

model.add(Conv2D(64,(3,3),activation='relu'))
model.add(MaxPooling2D(2,2))

model.add(Flatten())

model.add(Dense(128,activation='relu'))
model.add(Dropout(0.5))
```

```

model.add(Dense(1,activation='sigmoid'))

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

# Train Model
model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=10
)
# Save Model
model.save("deepfake_detector.h5")
# Predict Image
from tensorflow.keras.preprocessing import image
img = image.load_img("test.jpg", target_size=(128,128))
img = image.img_to_array(img)
img = img/255.0
img = np.expand_dims(img, axis=0)
prediction = model.predict(img)
if prediction[0][0] > 0.5:
    print("Prediction : Fake Image")
else:
    print("Prediction : Real Image")

```

Output:

Found 1600 images belonging to 2 classes.
Found 400 images belonging to 2 classes.

Epoch 1/10

50/50 [=====] - accuracy: 0.88

Epoch 10/10

50/50 [=====] - accuracy: 0.97

Validation Accuracy : 96.50%

Prediction : Fake Image

Result:

Thus, the **DeepFake Image Detection** model was successfully implemented using a **Convolutional Neural Network (CNN)**.

OBJECT DETECTION USING YOLO (YOU ONLY LOOK ONCE)

TOPIC BEYOND THE SYLLABUS

Aim:

To implement the **YOLO (You Only Look Once)** deep learning algorithm for real-time object detection and classification in images or videos.

Objective

- To understand the working principle of the YOLO object detection algorithm.
- To detect multiple objects simultaneously in an image.
- To classify detected objects with high accuracy.
- To display bounding boxes and confidence scores around detected objects.

Theory

YOLO (You Only Look Once) is a state-of-the-art deep learning algorithm used for real-time object detection. Unlike traditional object detection methods that first generate region proposals and then classify each region, YOLO performs detection and classification in a single forward pass through a Convolutional Neural Network (CNN). The input image is divided into a grid, and each grid cell predicts bounding boxes, confidence scores, and object class probabilities. This unified approach makes YOLO significantly faster while maintaining high detection accuracy. YOLO is widely used in autonomous vehicles, surveillance systems, robotics, smart traffic monitoring, and industrial automation.

Algorithm

- Step 1:** Start the program and import the required libraries such as OpenCV, NumPy, and the YOLO model.
- Step 2:** Load the pre-trained YOLO configuration file, weight file, and class label file.
- Step 3:** Read the input image or capture frames from a live webcam/video stream.
- Step 4:** Resize the input image to the size expected by the YOLO model.
- Step 5:** Convert the image into a blob and normalize the pixel values.
- Step 6:** Pass the blob through the YOLO network for forward propagation.
- Step 7:** Obtain the predicted bounding boxes, confidence scores, and class probabilities.
- Step 8:** Filter out detections whose confidence score is below the predefined threshold.
- Step 9:** Apply **Non-Maximum Suppression (NMS)** to remove duplicate and overlapping bounding boxes.
- Step 10:** Draw bounding boxes around the detected objects.
- Step 11:** Display the predicted class name and confidence score for each detected object.
- Step 12:** Display the final output image containing all detected objects.
- Step 13:** End the program.

Program:

```
pip install ultralytics
pip install opencv-python
from ultralytics import YOLO
import cv2

# Load the pre-trained YOLOv8 model
model = YOLO("yolov8n.pt")

# Load the input image
```

```

image = cv2.imread("test.jpg")

# Perform object detection
results = model(image)

# Display detected objects
annotated_image = results[0].plot()

cv2.imshow("YOLO Object Detection", annotated_image)
cv2.waitKey(0)
cv2.destroyAllWindows()

# Print detected object details
for result in results:
    boxes = result.boxes
    for box in boxes:
        class_id = int(box.cls[0])
        confidence = float(box.conf[0])

        print("Object :", model.names[class_id])
        print("Confidence :", round(confidence, 2))
        print("-----")

```

Output:

Object : person
Confidence : 0.98

Object : bicycle
Confidence : 0.94

Object : car
Confidence : 0.91

Result:

Thus, the **YOLO (You Only Look Once)** object detection model was successfully implemented using a pre-trained **YOLOv8** network.

