**RAMCO INSTITUTE OF TECHNOLOGY**

**RAJAPALAYAM-626117.**



**III YEAR / V SEMESTER**

**AD3511 - DEEP LEARNING LABARATORY**

**Prepared by,**

**B. REVATHI**

**Assistant Professor**

**AI & DS**

**Index**

|  |  |
| --- | --- |
| **Ex.No** | **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 and Decoder Model |
| 8 | Image Augumentation using GANS |
| 9 | Mini Project on Real World Applications |

|  |  |
| --- | --- |
| **Ex:No1** | **Solving XOR problem using DNN** |

**AIM:**

To write a python program of Solving XOR problem using DNN

**Algorithm:**

Step1: Start

Step 2: Data Preparation Generate the XOR truth table

Step 3: Create a deep neural network (DNN) with an input layer, one or more hidden layers, and an output layer.

Step 4: Define the number of neurons in each layer and choose an activation function for each layer.

Step 5: Train the DNN using the XOR truth table as training data and labels.

Step 6: Test the trained model with new XOR input combinations to evaluate its performance.

Step7: Stop

**Program:**

import numpy as np

from keras.models import Sequential

from keras.layers import Dense

# Input and output data for XOR

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

y = np.array([[0], [1], [1], [0]])

# Create a Sequential model

model = Sequential()

# Add layers to the model

model.add(Dense(8, input\_dim=2, activation='relu'))

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

# Compile the model

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model

model.fit(X, y, epochs=10000, verbose=0)

# Evaluate the model

loss, accuracy = model.evaluate(X, y)

print(f"Loss: {loss:.4f}, Accuracy: {accuracy\*100:.2f}%")

# Make predictions

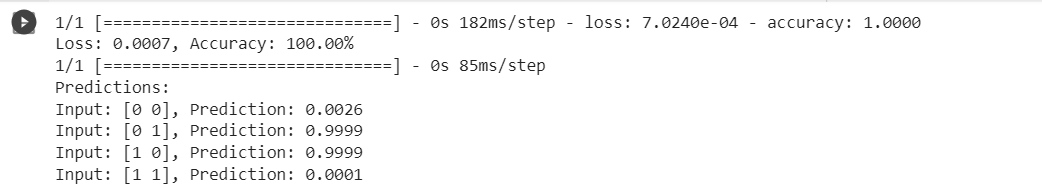
predictions = model.predict(X)

print("Predictions:")

for i in range(len(predictions)):

    print(f"Input: {X[i]}, Prediction: {predictions[i][0]:.4f}")

**Output:**



**Result:**

Thus the program of Solving XOR problem using DNN was executed and written successfully.

|  |  |
| --- | --- |
| **Ex:No:2** | **Character recognition using CNN** |

**AIM:**

To write a python program Character recognition using CNN

**Algorithm:**

Step1: Start

Step 1: Gather a labeled dataset of handwritten characters, such as MNIST or a custom dataset.

Step 2: Model Architecture

Design a CNN architecture suitable for character recognition.

Step 3: Define Model Parameters

Specify hyperparameters like learning rate, batch size, number of epochs, and regularization techniques (e.g., dropout).

Step 4: Build the CNN Model

Implement the CNN architecture using a deep learning framework like TensorFlow or PyTorch.

Step 5: Compile the Model

Compile the CNN model using the specified loss function, optimizer, and metrics (e.g., accuracy).

Step 6: Train the Model

Train the CNN model using the preprocessed training data.

Step 7: Evaluate the Model

Evaluate the trained model using the preprocessed testing data.

Step 8: Stop

**Program:**

import numpy as np

import tensorflow as tf

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

from tensorflow.keras.utils import to\_categorical

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

x\_train = np.expand\_dims(x\_train, axis=-1)

x\_test = np.expand\_dims(x\_test, axis=-1)

y\_train = to\_categorical(y\_train, num\_classes=10)

y\_test = to\_categorical(y\_test, num\_classes=10)

x\_train = np.expand\_dims(x\_train, axis=-1)

x\_test = np.expand\_dims(x\_test, axis=-1)

y\_train = to\_categorical(y\_train, num\_classes=10)

y\_test = to\_categorical(y\_test, num\_classes=10)

import matplotlib.pyplot as plt

# Visualize some of the train images

num\_samples\_to\_visualize = 5

plt.figure(figsize=(15, 3))

for i in range(num\_samples\_to\_visualize):

    plt.subplot(1, num\_samples\_to\_visualize, i+1)

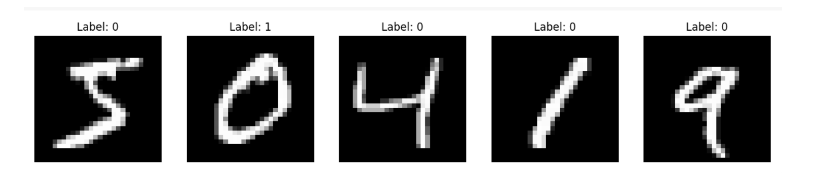
    plt.imshow(x\_train[i].reshape(28, 28), cmap='gray')

    plt.title(f"Label: {np.argmax(y\_train[i])}")

    plt.axis('off')

plt.show()

**Output:**



model = Sequential([

    Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),

    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3), activation='relu'),

    MaxPooling2D((2, 2)),

    Flatten(),

    Dense(128, activation='relu'),

    Dropout(0.5),

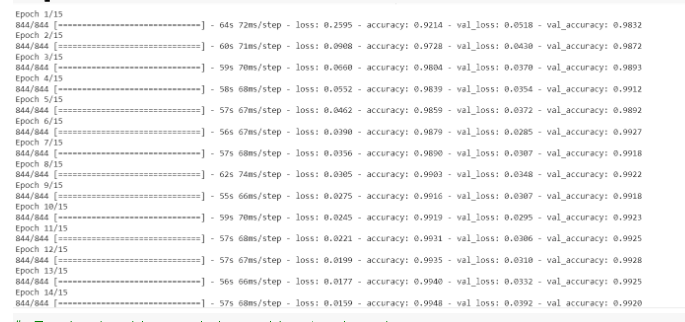
    Dense(10, activation='softmax')])

    model.compile(optimizer='adam', loss='categorical\_crossentropy',

    metrics=['accuracy'])

    model.fit(x\_train, y\_train, epochs=15, batch\_size=64,                                                validation\_split=0.1)

**Output:**



# Evaluate the model on the test set

test\_loss, test\_accuracy = model.evaluate(x\_test, y\_test)

print(f"Test accuracy: {test\_accuracy}")

**Output:**



# Choose an index from the test set

import matplotlib.pyplot as plt

index = 0

# Get the image and its label

test\_image = x\_test[index]

true\_label = np.argmax(y\_test[index])

# Make a prediction

prediction = model.predict(np.expand\_dims(test\_image, axis=0))

predicted\_label = np.argmax(prediction)

# Calculate accuracy for this individual image

accuracy = 100 \* (predicted\_label == true\_label)

# Display the test image

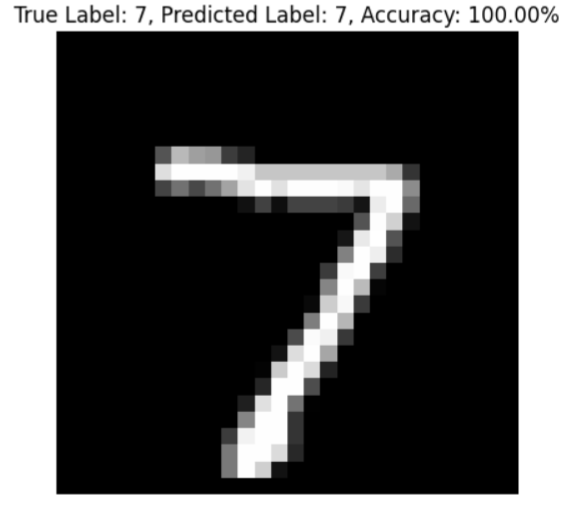
plt.imshow(test\_image.reshape(28, 28), cmap='gray')

plt.title(f"True Label: {true\_label}, Predicted Label: {predicted\_label}, Accuracy: {accuracy:.2f}%")

plt.axis('off')

plt.show()

**Output:**



**Result:**

Thus the program of Character recognition using CNN was executed and written successfully.

|  |  |
| --- | --- |
| **EX.NO 3** | **Face Recognition using CNN** |

**AIM:**

To write a python program Face Recognition using CNN

**Algorithm:**

Step 1: Collect a dataset of facial images with labels.

Step 2: Data Pre-processing

Resize all facial images to a consistent size.

Normalize pixel values, Augment the dataset.

Step 3: Divide the dataset into training, validation, and test sets.

Step 4: Build the CNN Model. Design the CNN architecture for face recognition. Add convolutional layers, pooling layers, and fully connected layers. Use activation functions like ReLU.

Step 5: Model Compilation

Step 6: Model Train the CNN model on the training dataset.

Use the validation dataset to monitor performance and prevent overfitting.

Step 7: Model Evaluation

Evaluate the trained model on the test dataset.Calculate metrics like accuracy, precision, recall, and F1-score.

**Program:**

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.preprocessing import image

from tensorflow.keras.optimizers import RMSprop

import tensorflow as tf

import matplotlib.pyplot as plt

import cv2

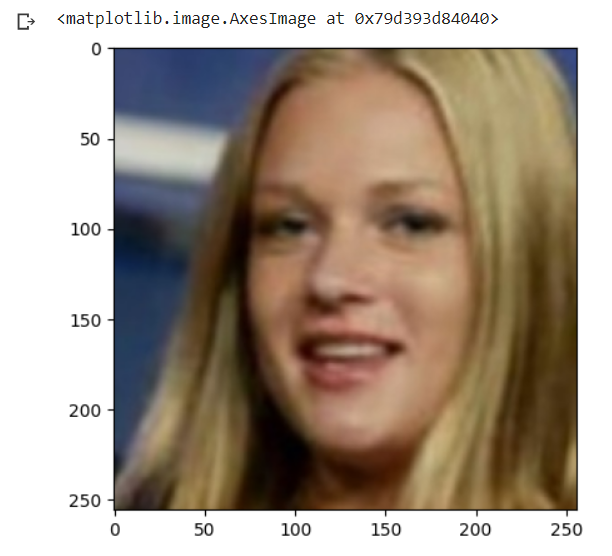
import os

import numpy as np

img = image.load\_img("/content/drive/MyDrive/deep learning/archive (1)/basedata/training/face/person\_0000.jpg")

plt.imshow(img)

**Output:**



cv2.imread("/content/drive/MyDrive/deep learning/archive (1)/basedata/training/face/person\_0000.jpg").shape

**Output:**



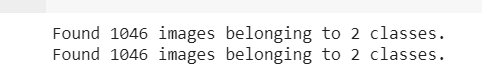
train = ImageDataGenerator(rescale = 1/255)

validation =ImageDataGenerator(rescale = 1/255)

trained\_dataset = train.flow\_from\_directory("/content/drive/MyDrive/deep learning/archive (1)/basedata/training",target\_size=(200,200),batch\_size= 3, class\_mode ='binary')

validation\_dataset = train.flow\_from\_directory("/content/drive/MyDrive/deep learning/archive (1)/basedata/validation",target\_size=(200,200),batch\_size= 3,class\_mode ='binary')

**Output:**



trained\_dataset.class\_indices

**Output:**



trained\_dataset. classes

Output:



model=tf.keras.models.Sequential([tf.keras.layers.Conv2D(16,(3,3),activation='relu',input\_shape=(200,200,3)),

tf.keras.layers.MaxPool2D(2,2),

tf.keras.layers.Conv2D(32,(3,3),activation='relu',input\_shape=(200,200,3)),

tf.keras.layers.MaxPool2D(2,2),

tf.keras.layers.Conv2D(64,(3,3),activation='relu',input\_shape=(200,200,3)),

                                    tf.keras.layers.MaxPool2D(2,2),

                                    tf.keras.layers.Flatten(),

                                    tf.keras.layers.Dense(512,activation='relu'),

                                    tf.keras.layers.Dense(1,activation='sigmoid')

                                   ])

model.compile(loss ='binary\_crossentropy',

             optimizer = RMSprop(learning\_rate=0.001),

             metrics=['accuracy'])

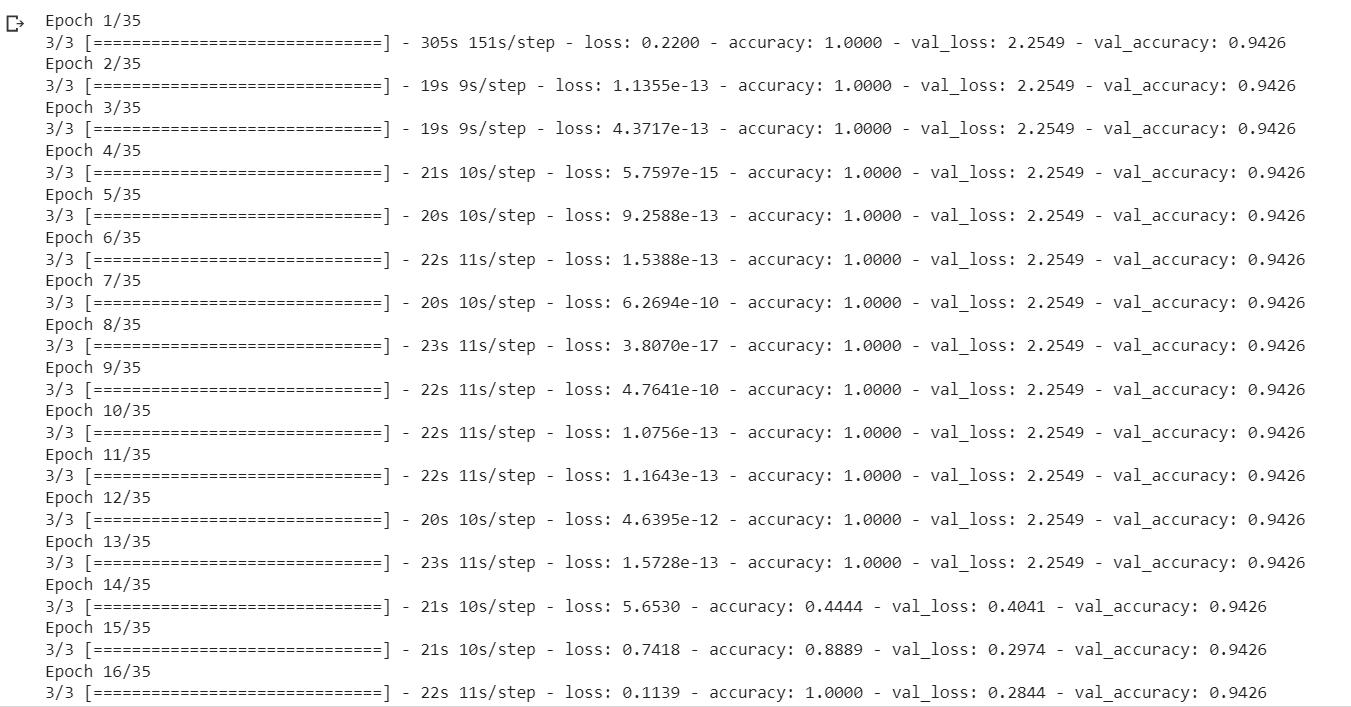
model\_fit = model.fit(trained\_dataset,

                     steps\_per\_epoch = 3,

                     epochs =35,

                     validation\_data = validation\_dataset)

**Output:**



dir\_path = "/content/drive/MyDrive/deep learning/testing"

for i in os.listdir(dir\_path):

    img = image.load\_img(dir\_path + '//' + i,target\_size=(200,200))

    plt.imshow(img)

    plt.show()

    X= image.img\_to\_array(img)

    X = np.expand\_dims(X,axis=0)

    images = np.vstack([X])

    val = model.predict(images)

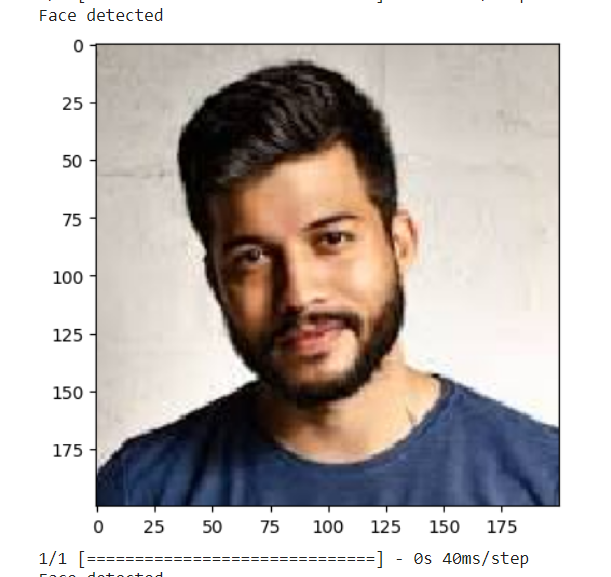
    if val == 0:

        print("Face detected")

    else:

        print("Face not detected")

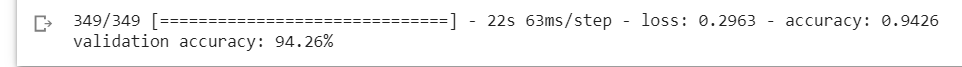
**Output:**



validation\_accuracy = model.evaluate(validation\_dataset)

print(f"validation accuracy: {validation\_accuracy[1] \* 100:.2f}%")

**Output:**



import tensorflow.keras.models

model.save('detection\_model.h5')

loaded\_model = tf.keras.models.load\_model('detection\_model.h5')

loaded\_model

**Result:**

Thus the program Face Recognition using CNN was executed and written successfully.

|  |  |
| --- | --- |
| **EX:NO.4** | **Language Modeling using RNN** |

**Aim:**

To Write a python Program Language modeling using RNN

**Algorithm:**

Step 1: Data Collection

Gather a large corpus of text data for training the language model. This text data can be from books, articles, websites, or any source relevant to your specific task.

Step 2: Data Preprocessing

Tokenize the text into words or subword units (e.g., using tools like NLTK or spaCy).

Step 3: Data Sequencing

Divide the text data into sequences or chunks of a fixed length (e.g., sentences or paragraphs).

Step 4: Model Architecture

Choose an RNN architecture, such as LSTM (Long Short-Term Memory) or GRU (Gated Recurrent Unit), or a combination of multiple layers.

Step 5: Embedding Layer (Optional)

If needed, include an embedding layer to convert integer IDs to dense vector representations.

Step 6: Model Compilation

Choose a loss function, typically categorical cross-entropy for language modeling tasks.

Step 7: Model Training

Train the RNN model using the preprocessed data.

Step 8: Model Evaluation

Evaluate the trained model's performance on a separate test dataset.

**Program:**

import numpy as np

import pandas as pd

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Embedding, LSTM, Dense

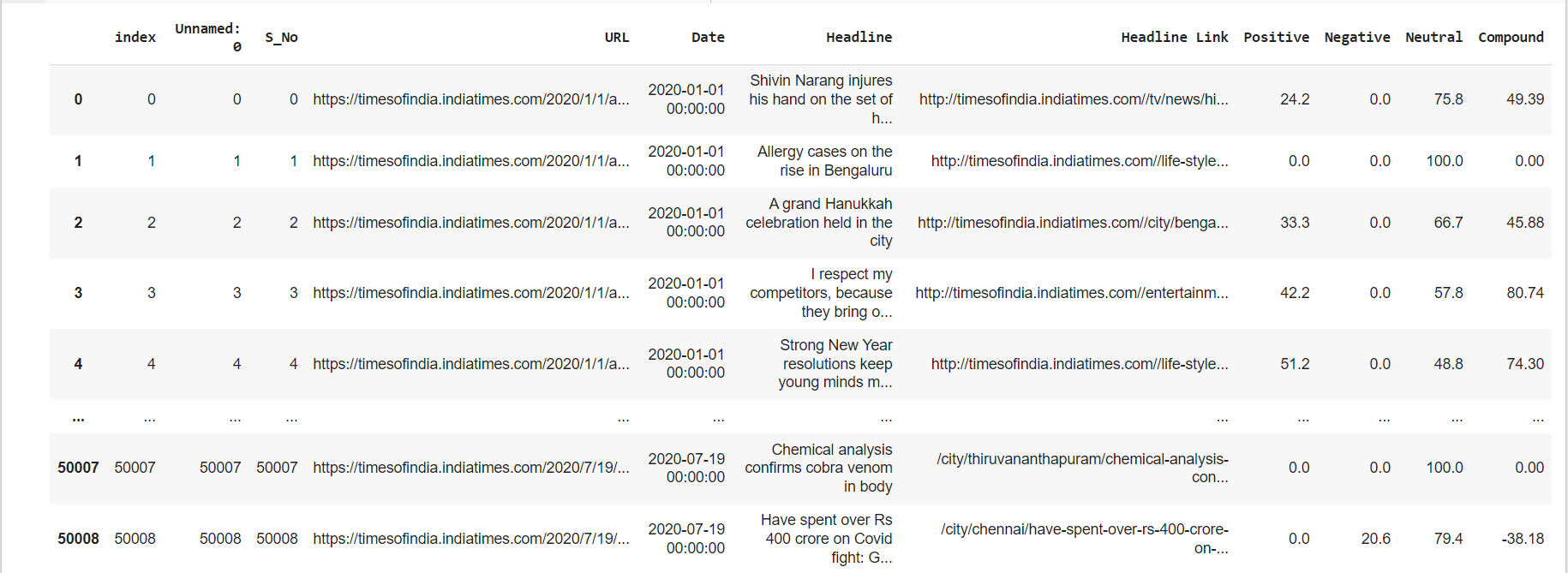
from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

df = pd.read\_csv('/content/drive/MyDrive/deep learning/Times\_of\_India\_Healines\_since\_jan\_2020\_score.csv')

df

**Output:**



headlines = df['Headline'].tolist()

# Assuming headlines is a list of strings with some non-text values

headlines = [str(headline) for headline in headlines]

headlines = headlines[:20]

tokenizer = Tokenizer()

tokenizer.fit\_on\_texts(headlines)

total\_words = len(tokenizer.word\_index) + 1

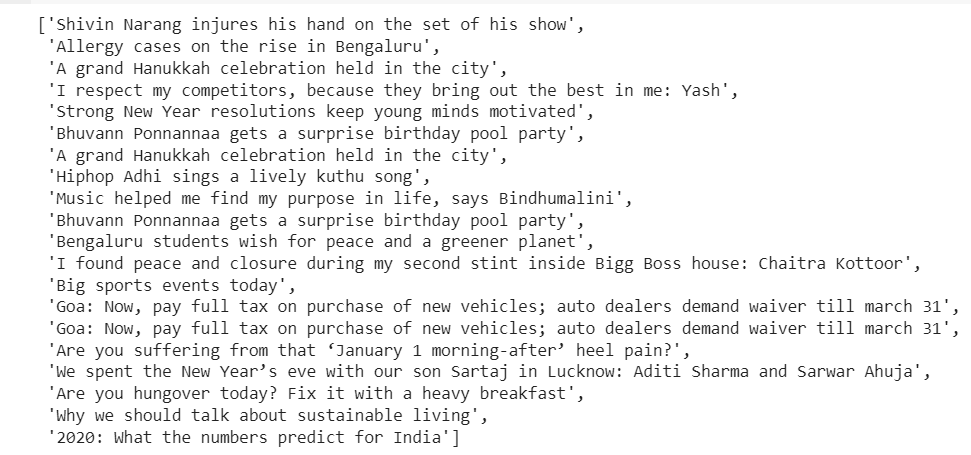
len(headlines)

**Output:**



Headlines

**Output:**



input\_sequences = []

for line in headlines:

    token\_list = tokenizer.texts\_to\_sequences([line])[0]

    for i in range(1, len(token\_list)):

        n\_gram\_sequence = token\_list[:i+1]

        input\_sequences.append(n\_gram\_sequence)

max\_sequence\_length = max([len(seq) for seq in input\_sequences])

input\_sequences = pad\_sequences(input\_sequences, maxlen=max\_sequence\_length, padding='pre')

predictors, label = input\_sequences[:, :-1], input\_sequences[:, -1]

label = tf.keras.utils.to\_categorical(label, num\_classes=total\_words)

model = Sequential()

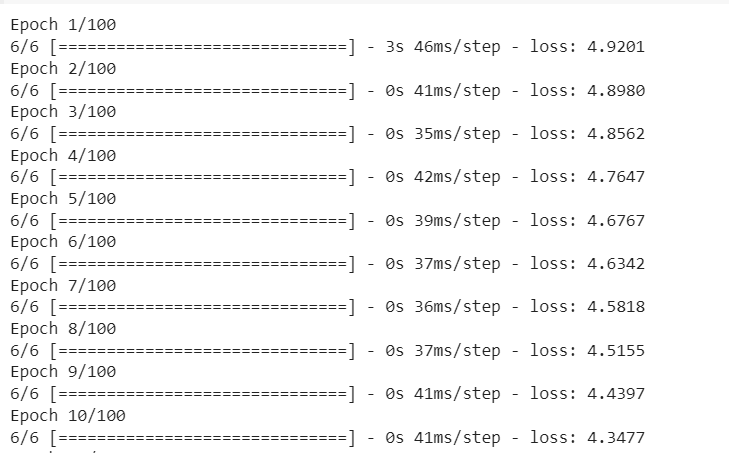
model.add(Embedding(total\_words, 100, input\_length=max\_sequence\_length-1))  # Reduced embedding dimension

model.add(LSTM(150))  # Reduced LSTM units

model.add(Dense(total\_words, activation='softmax'))

model.compile(loss='categorical\_crossentropy', optimizer='adam')

model.fit(predictors, label, epochs=100, verbose=1)



import numpy as np

def generate\_text(seed\_text, next\_words, model, max\_sequence\_length, tokenizer, temperature=1.0):

    generated\_text = seed\_text

    for \_ in range(next\_words):

        token\_list = tokenizer.texts\_to\_sequences([seed\_text])[0]

        token\_list = pad\_sequences([token\_list], maxlen=max\_sequence\_length-1, padding='pre')

        predicted\_probabilities = model.predict(token\_list, verbose=0)[0]

  # Apply temperature to the predicted probabilities

        predicted\_probabilities = np.log(predicted\_probabilities) / temperature

        predicted\_probabilities = np.exp(predicted\_probabilities)

        predicted\_probabilities /= np.sum(predicted\_probabilities)

# Sample a word based on the adjusted probabilities

        predicted = np.random.choice(len(predicted\_probabilities), size=1, p=predicted\_probabilities)[0]

      output\_word = ""

        for word, index in tokenizer.word\_index.items():

            if index == predicted:

                output\_word = word

                break

        seed\_text += " " + output\_word

        generated\_text += " " + output\_word

    return generated\_text

seed\_text = "Music helped me find my purpose in life"

next\_words = 20  # Number of words to generate

generated\_text = generate\_text(seed\_text, next\_words, model, max\_sequence\_length, tokenizer)

print(generated\_text)

**Output:**



**Result:**

Thus the program Language modeling using RNN was executed and written successfully.

|  |  |
| --- | --- |
| **EX:NO:5** | **Sentiment analysis using LSTM** |

**AIM:**

To write a python program Sentiment analysis using LSTM

**Algorithm:**

Step 1: Data Collection

Gather a dataset with labeled text examples and their corresponding sentiment labels (e.g., positive, negative, or neutral).

Step 2: Data Preprocessing

Tokenize the text: Split the text into individual words or subword tokens.

Convert text to numerical form: Assign a unique numerical ID to each word/token (word embedding).

Pad sequences: Ensure all sequences have the same length by padding or truncating them as needed.

Step 3: Split the Dataset

Divide your dataset into three subsets: training, validation, and test sets.

Step 4: LSTM Model Architecture

Define an LSTM model architecture with layers such as embedding, LSTM, and dense layers.

Configure input sequence length, embedding dimension, LSTM units, and the number of output units corresponding to sentiment classes.

Step 5: Compile the Model

Choose a loss function, an optimizer, and evaluation metrics.

Step 6: Training

Train the LSTM model on the training dataset.

Monitor training progress with validation data.

Use techniques like early stopping to prevent overfitting.

Step 7: Model Evaluation

Evaluate the model's performance on the test dataset using metrics like accuracy and F1 score.

**Program:**

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

import re

import pickle

import numpy as np

import pandas as pd

# Plot libraries

import seaborn as sns

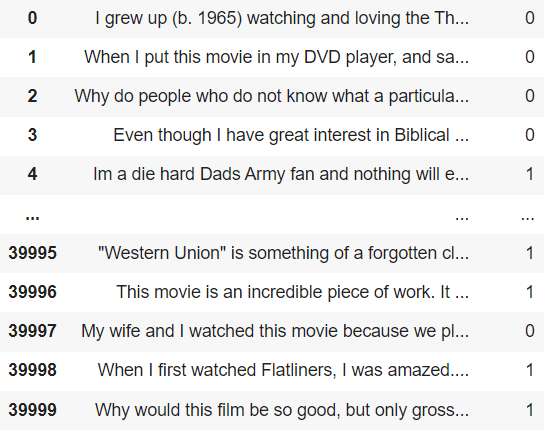
from wordcloud import WordCloud

import matplotlib.pyplot as plt

df = pd.read\_csv("/content/drive/MyDrive/deep learning/movie.csv")

df

**Output:**



# Tokenize the text

tokenizer = Tokenizer(num\_words=5000, oov\_token='<OOV>')

tokenizer.fit\_on\_texts(texts)

# Separate text and labels

texts = df['text'].tolist()

labels = df['label'].tolist()

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from tensorflow.keras.utils import to\_categorical

# Convert text to sequences

sequences = tokenizer.texts\_to\_sequences(texts)

# Pad sequences to have the same length

max\_length = 100  # Choose an appropriate value

padded\_sequences = pad\_sequences(sequences, maxlen=max\_length, padding='post', truncating='post')

labels = to\_categorical(labels, num\_classes=2)

from sklearn.model\_selection import train\_test\_split

train\_texts, test\_texts, train\_labels, test\_labels = train\_test\_split(padded\_sequences, labels, test\_size=0.2, random\_state=42)

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Embedding, LSTM, Dense

model = Sequential()

model.add(Embedding(input\_dim=5000, output\_dim=16, input\_length=max\_length))

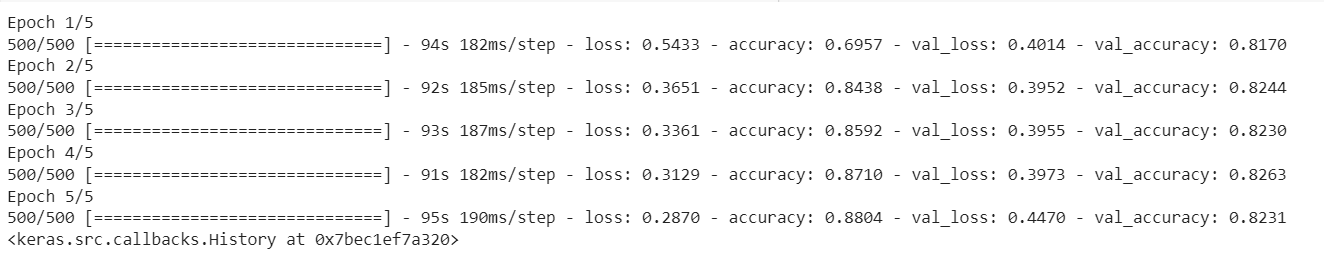
model.add(LSTM(128))

model.add(Dense(2, activation='softmax'))

model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

model.fit(train\_texts, train\_labels, epochs=5, batch\_size=64, validation\_data=(test\_texts, test\_labels))

**Output:**

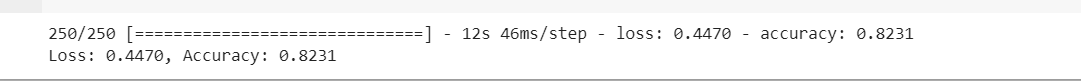


model.save('your\_lstm\_model.h5')

loss, accuracy = model.evaluate(test\_texts, test\_labels)

print(f'Loss: {loss:.4f}, Accuracy: {accuracy:.4f}')

**Output:**



# Example sentiment prediction

new\_texts = ["This movie is great!"]

new\_sequences = tokenizer.texts\_to\_sequences(new\_texts)

new\_padded = pad\_sequences(new\_sequences, maxlen=max\_length, padding='post', truncating='post')

predictions = model.predict(new\_padded)

print(predictions)

sentiment = model.predict(new\_padded,batch\_size=1,verbose=2)[0]

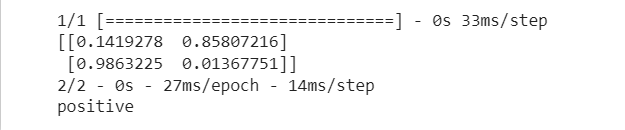
if np.argmax(sentiment) == 0:

  print("negetive")

elif np.argmax(sentiment) == 1:

  print("positive")

**Output:**



**Result:**

Thus the program Sentiment analysis using LSTM was executed and written successfully.

|  |  |
| --- | --- |
| **EX.No.6** | **Parts of speech Tagging using Sequence to sequence Architecture** |

**Aim:**

To write a python program Parts of speech Tagging using Sequence to sequence Architecture

**Algorithm:**

Step 1: Data Preparation

Data preprocessing: Tokenize sentences into words and tags, and create a vocabulary of words and part-of-speech tags. You may need to pad or truncate sentences to a fixed length.

Step 2: Model Architecture

Sequence-to-Sequence Model: Create a sequence-to-sequence model with an encoder-decoder architecture. You can use a recurrent neural network (RNN), a long short-term memory (LSTM) network, or a transformer-based model. Encoder: The encoder processes the input sentence (sequence of words). It can be a bidirectional RNN or a transformer encoder. The encoder's output will be a representation of the input sentence.

Decoder: The decoder takes the encoder's output and generates a sequence of part-of-speech tags. It can be a unidirectional RNN or a transformer decoder.

Output Layer: Use a softmax layer at the output of the decoder to predict the part-of-speech tag for each word in the input sentence. The number of output units in the softmax layer should match the number of unique part-of-speech tags in your dataset.

Step 3: Training

Loss Function: Use a categorical cross-entropy loss to compute the error between predicted part-of-speech tags and the ground truth tags.

Optimizer: Utilize an optimizer like Adam or SGD to minimize the loss during training.

Training Loop: Train the model on the training data. Monitor the loss on the validation set to prevent overfitting.

Hyperparameter Tuning: Experiment with hyperparameters such as learning rate, batch size, and model architecture to optimize performance.

Step 4: Evaluation

Testing: Evaluate the trained model on the test dataset to assess its accuracy and performance.

Performance Metrics: Use metrics like accuracy, precision, recall, and F1-score to measure the quality of part-of-speech tagging.

**Program:**

import torch

import torch.nn as nn

import torch.optim as optim

input\_sequence = [&quot;She&quot;, &quot;reads&quot;, &quot;a&quot;, &quot;book&quot;]

output\_sequence = [&quot;PRP&quot;, &quot;VBZ&quot;, &quot;T&quot;, &quot;N&quot;]

input\_vocab = set(input\_sequence)

output\_vocab = set(output\_sequence)

input\_word2index = {word: i for i, word in enumerate(input\_vocab)}

input\_index2word = {i: word for i, word in enumerate(input\_vocab)}

output\_word2index = {tag: i for i, tag in enumerate(output\_vocab)}

output\_index2word = {i: tag for i, tag in enumerate(output\_vocab)}

input\_indices = [input\_word2index[word] for word in input\_sequence]

output\_indices = [output\_word2index[tag] for tag in output\_sequence]

class Seq2Seq(nn.Module):

    def \_\_init\_\_(self, input\_size, output\_size, hidden\_size):

        super(Seq2Seq, self).\_\_init\_\_()

        self.embedding = nn.Embedding(input\_size, hidden\_size)

        self.gru = nn.GRU(hidden\_size, hidden\_size)

        self.out = nn.Linear(hidden\_size, output\_size)

    def forward(self, input\_seq):

        embedded = self.embedding(input\_seq)

        output, hidden = self.gru(embedded)

        output = self.out(output)

        return output

input\_size = len(input\_vocab)

output\_size = len(output\_vocab)

hidden\_size = 256

learning\_rate = 0.01

epochs = 100

model = Seq2Seq(input\_size, output\_size, hidden\_size)

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(model.parameters(), lr=learning\_rate)

for epoch in range(epochs):

    optimizer.zero\_grad()

    input\_tensor = torch.tensor(input\_indices, dtype=torch.long)

    output\_tensor = torch.tensor(output\_indices, dtype=torch.long)

    output = model(input\_tensor)

    loss = criterion(output.view(-1, output\_size), output\_tensor.view(-1))

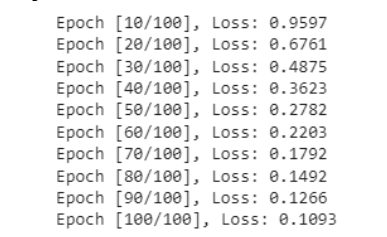
    loss.backward()

    optimizer.step()

    if (epoch + 1) % 10 == 0:

        print(f&#39;Epoch [{epoch + 1}/{epochs}], Loss: {loss.item():.4f}&#39;)

**Output:**



# Create word-to-index and index-to-word mappings for input

input\_word2index = {word: i for i, word in enumerate(input\_vocab)}

input\_word2index[&#39;&lt;UNK&gt;&#39;] = len(input\_vocab)  # Add an unknown word token

input\_index2word = {i: word for i, word in enumerate(input\_vocab)}

input\_index2word[len(input\_vocab)] = &#39;&lt;UNK&gt;&#39;

# Inference

with torch.no\_grad():

    input\_test = [&quot;She&quot;, &quot;reads&quot;, &quot;a&quot;, &quot;book&quot;]

    input\_test\_indices = [input\_word2index.get(word, input\_word2index[&#39;&lt;UNK&gt;&#39;]) for word

in input\_test]

    input\_test\_tensor = torch.tensor(input\_test\_indices, dtype=torch.long).unsqueeze(1)  # Add

a batch dimension

    # Initialize the hidden state for the model

    hidden = torch.zeros(1, 1, hidden\_size)

    predicted\_indices = model(input\_test\_tensor)

  # Extract the most likely tag for each word in the sequence

    predicted\_indices = predicted\_indices.argmax(dim=2)

    # Convert to a list of tag indices

    predicted\_indices = predicted\_indices.view(-1).tolist()

    # Convert the predicted indices to tags

    predicted\_tags = [output\_index2word[i] for i in predicted\_indices]

print(&quot;Input sentence:&quot;, input\_test)

    print(&quot;Predicted tags:&quot;, predicted\_tags)

**Output:**



**Result:**

Thus the program of Parts of speech Tagging using Sequence to sequence Architecture was executed and written successfully.

|  |  |
| --- | --- |
| **EX.NO. 7** | **Machine Translation Using Encoder – Decoder model** |

**Aim:**

To write a python program of Machine Translation Using Encode – Decoder model

**Algorithm:**

**Step 1: Data Preparation**

Data preprocessing:

Tokenize sentences into words and tags.

Create a vocabulary of words and part-of-speech tags.

Perform padding or truncation of sentences to a fixed length if necessary.

**Step 2: Model Architecture**

Sequence-to-Sequence Model:

Create a sequence-to-sequence model with an encoder-decoder architecture.

Choose the architecture for the encoder and decoder:

Encoder: (Specify the encoder type) processes the input sentence, and it can be a bidirectional RNN or a transformer encoder.

Decoder: (Specify the decoder type) generates a sequence of part-of-speech tags, and it can be a unidirectional RNN or a transformer decoder.

Output Layer: Use a softmax layer at the output of the decoder to predict the part-of-speech tag for each word in the input sentence. The number of output units should match the number of unique part-of-speech tags in your dataset.

**Step 3: Training**

Loss Function:

Use categorical cross-entropy loss to compute the error between predicted part-of-speech tags and ground truth tags.

Optimizer:

Utilize an optimizer, such as Adam or SGD, to minimize the loss during training.

**Step 4: Evaluation**

Testing:

Evaluate the trained model on the test dataset to assess its accuracy and performance.

Performance Metrics:

Use metrics like accuracy, precision, recall, and F1-score to measure the quality of part-of-speech tagging.

**Program:**

Machine Translation Using Encode – Decoder model:

Program:

import pandas as pd

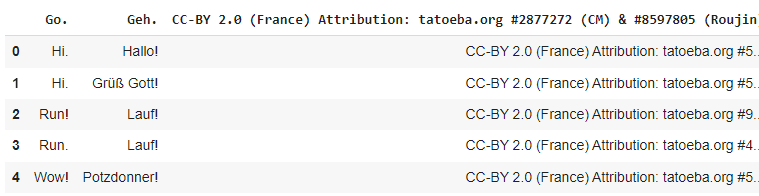
import numpy as np

import matplotlib.pyplot as plt

df = pd.read\_csv('/content/drive/MyDrive/deep learning/deu.txt',sep='\t')

df.head(5)

**Output:**



data\_path = '/content/drive/MyDrive/deep learning/deu.txt'

df.columns = ["English", "French","text"]

# Print the first 10 rows

df.head(10)

**Output:**



df.drop('text',axis=1,inplace=True)

df.head(10)

**Output:**



from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input,LSTM,Dense

batch\_size=64

epochs=100

latent\_dim=256 # here latent dim represent hidden state or cell state

num\_samples=10000

# Vectorize the data.

input\_texts = []

target\_texts = []

input\_characters = set()

target\_characters = set()

with open(data\_path, 'r', encoding='utf-8') as f:

    lines = f.read().split('\n')

for line in lines[: min(num\_samples, len(lines) - 1)]:

    input\_text, target\_text, \_ = line.split('\t')

    # We use "tab" as the "start sequence" character

    # for the targets, and "\n" as "end sequence" character.

    target\_text = '\t' + target\_text + '\n'

    input\_texts.append(input\_text)

    target\_texts.append(target\_text)

    for char in input\_text:

        if char not in input\_characters:

            input\_characters.add(char)

    for char in target\_text:

        if char not in target\_characters:

            target\_characters.add(char)

input\_characters=sorted(list(input\_characters))

target\_characters=sorted(list(target\_characters))

num\_encoder\_tokens=len(input\_characters)

num\_decoder\_tokens=len(target\_characters)

max\_encoder\_seq\_length=max([len(txt) for txt in input\_texts])

max\_decoder\_seq\_length=max([len(txt) for txt in target\_texts])

print('Number of samples:', len(input\_texts))

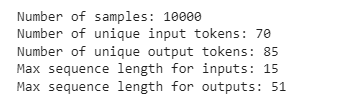
print('Number of unique input tokens:', num\_encoder\_tokens)

print('Number of unique output tokens:', num\_decoder\_tokens)

print('Max sequence length for inputs:', max\_encoder\_seq\_length)

print('Max sequence length for outputs:', max\_decoder\_seq\_length)

**Output:**



input\_token\_index=dict([(char,i) for i, char in enumerate(input\_characters)])

target\_token\_index=dict(

[(char,i) for i, char in enumerate(target\_characters)])

encoder\_input\_data = np.zeros(

    (len(input\_texts), max\_encoder\_seq\_length, num\_encoder\_tokens),

    dtype='float32')

decoder\_input\_data = np.zeros(

    (len(input\_texts), max\_decoder\_seq\_length, num\_decoder\_tokens),

    dtype='float32')

decoder\_target\_data = np.zeros(

    (len(input\_texts), max\_decoder\_seq\_length, num\_decoder\_tokens),

    dtype='float32')

for i, (input\_text, target\_text) in enumerate(zip(input\_texts, target\_texts)):

    for t, char in enumerate(input\_text):

        encoder\_input\_data[i, t, input\_token\_index[char]] = 1.

    encoder\_input\_data[i, t + 1:, input\_token\_index[' ']] = 1.

    for t, char in enumerate(target\_text):

        # decoder\_target\_data is ahead of decoder\_input\_data by one timestep

        decoder\_input\_data[i, t, target\_token\_index[char]] = 1.

        if t > 0:

            # decoder\_target\_data will be ahead by one timestep

            # and will not include the start character.

            decoder\_target\_data[i, t - 1, target\_token\_index[char]] = 1.

    decoder\_input\_data[i, t + 1:, target\_token\_index[' ']] = 1.

    decoder\_target\_data[i, t:, target\_token\_index[' ']] = 1.

encoder\_inputs = Input(shape=(None, num\_encoder\_tokens))

encoder = LSTM(latent\_dim, return\_state=True)

encoder\_outputs, state\_h, state\_c = encoder(encoder\_inputs)

encoder\_states = [state\_h, state\_c]

decoder\_inputs = Input(shape=(None, num\_decoder\_tokens))

decoder\_lstm = LSTM(latent\_dim, return\_sequences=True, return\_state=True)

decoder\_outputs, \_, \_ = decoder\_lstm(decoder\_inputs,

                                     initial\_state=encoder\_states)

decoder\_dense = Dense(num\_decoder\_tokens, activation='softmax')

decoder\_outputs = decoder\_dense(decoder\_outputs)

model = Model([encoder\_inputs, decoder\_inputs], decoder\_outputs)

model.compile(optimizer='rmsprop', loss='categorical\_crossentropy',

              metrics=['accuracy'])

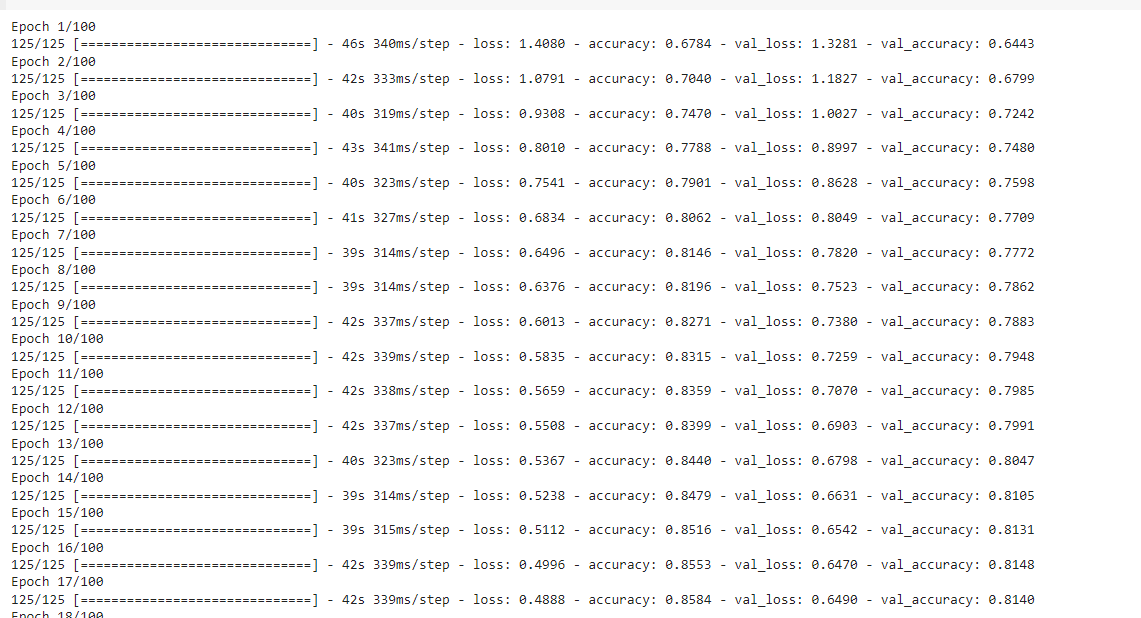
model.fit([encoder\_input\_data, decoder\_input\_data], decoder\_target\_data,

          batch\_size=batch\_size,

          epochs=epochs,

          validation\_split=0.2)

**Output:**



model.save('eng-german.h5')

encoder\_model = Model(encoder\_inputs, encoder\_states)

decoder\_state\_input\_h = Input(shape=(latent\_dim,))

decoder\_state\_input\_c = Input(shape=(latent\_dim,))

decoder\_states\_inputs = [decoder\_state\_input\_h, decoder\_state\_input\_c]

decoder\_outputs, state\_h, state\_c = decoder\_lstm(

    decoder\_inputs, 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)

reverse\_input\_char\_index = dict(

    (i, char) for char, i in input\_token\_index.items())

reverse\_target\_char\_index = dict(

    (i, char) for char, i in target\_token\_index.items())

for i in range(5):

  input\_seq = encoder\_input\_data[i:i+1]

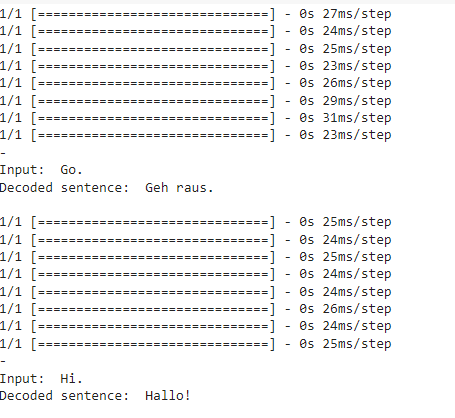
  decoded\_sentence = decode\_sequence(input\_seq)

  print('-')

  print("Input: ",input\_texts[i])

  print("Decoded sentence: ",decoded\_sentence)

**Output:**



**Result:**

Thus the program was Machine Translation Using Encode – Decoder model was executed and suceesfully.

|  |  |
| --- | --- |
| **EX.No. 8** | **Image augmentation using GANs** |

**Aim:**

To write a python program of Image augmentation using GANs

**Algorithm:**

**Step 1: Data Preparation**

Collect and preprocess your original dataset of images for training the GAN.

Define the target augmentation transformations you want to achieve (e.g., rotation, scaling, brightness changes, etc.).

**Step 2: GAN Training**

Generator Network:

Initialize a generator network (G) that will learn to generate augmented images. The architecture of G should be appropriate for generating images similar to your original dataset.

Discriminator Network:

Initialize a discriminator network (D) to distinguish between real and generated images.

**Step 3: Image Augmentation**

Use the trained generator (G) to generate augmented images. You can apply specific augmentation parameters (e.g., rotation angles, scaling factors) as input conditions to G to control the type of augmentation.

Apply the generated augmented images to your original dataset as needed.

**Step 4: Evaluation and Testing**

Evaluate the quality of the augmented images and assess their impact on the performance of your machine learning model (if used for model training).

**Step 5: Fine-Tuning (Optional)**

Optionally, fine-tune the GAN or retrain it with more specific augmentation requirements or on additional data.

**Step 6: Image Augmentation Using GANs Algorithm (Simplified)**

Prepare your dataset and define desired augmentation transformations.

Train a GAN with a generator (G) and discriminator (D).

**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 = 10

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):

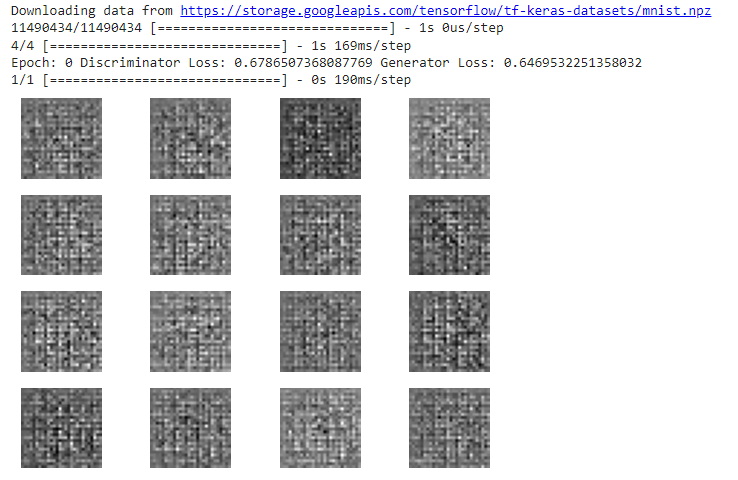
                axs[i, j].imshow(samples[count, :, :], cmap='gray')

                axs[i, j].axis('off')

                count += 1

        plt.show()

**Output:**





**Result:**

Thus the program was Machine Translation Using Encoder – Decoder model was executed and successfully.