**EX.NO:1 Solution to XOR problem using DNN**

**DATE:**

**AIM:**

To solve XOR problem using DNN

**PROCEDURE:**

* 1. Define the XOR input and output data
  2. Define the DNN architecture
  3. Compile the model
  4. Train the DNN
  5. Test the trained DNN

**PROGRAM**

import tensorflow as tf import numpy as np

# Define the XOR input and output data

x\_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float32)

y\_data = np.array([[0], [1], [1], [0]], dtype=np.float32)

# Define the DNN architecture model = tf.keras.Sequential([

tf.keras.layers.Dense(8, input\_dim=2, activation='relu'), tf.keras.layers.Dense(8, activation='relu'), tf.keras.layers.Dense(1, activation='sigmoid')

])

# Compile the model

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

# Train the DNN

model.fit(x\_data, y\_data, epochs=1000, verbose=0)

# Test the trained DNN

predictions = model.predict(x\_data) rounded\_predictions = np.round(predictions) print("Predictions:", rounded\_predictions)

**OUTPUT**

Predictions: [[0.]

[1.][1.][0.]]

**EX.NO:2 Character recognition using CNN**

**DATE:**

**AIM:**

To implement character Recognition using CNN

**PROCEDURE:**

|  |  |
| --- | --- |
| **1** | Load the MNIST dataset |
| **2** | Preprocess the data |
| **3** | Define the CNN architecture |
| **4** | Compile, train and evaluate the model |

**PROGRAM**

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

# Load the MNIST dataset

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

# Preprocess the data

x\_train = x\_train.reshape(-1, 28, 28, 1).astype('float32') / 255.0

x\_test = x\_test.reshape(-1, 28, 28, 1).astype('float32') / 255.0

# Define the CNN architecture model = Sequential()

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

model.add(MaxPooling2D((2, 2)))

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

model.add(MaxPooling2D((2, 2))) model.add(Flatten()) model.add(Dense(64, activation='relu'))

model.add(Dense(10, activation='softmax'))

# Compile the model

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

# Train the CNN

model.fit(x\_train, y\_train, epochs=5, batch\_size=64, validation\_data=(x\_test, y\_test))

# Evaluate the model

loss, accuracy = model.evaluate(x\_test, y\_test) print('Test Loss:', loss)

print('Test Accuracy:', accuracy)

**OUTPUT**

Epoch 1/5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 938/938 [==============================] - 21s | 21ms/step | - loss: | 0.1687 | - accuracy: | 0. |
| 9493 - val\_loss: 0.0792 - val\_accuracy: 0.9755 |  |  |  |  |  |
| Epoch 2/5 |  |  |  |  |  |
| 938/938 [==============================] - 19s | 21ms/step | - loss: | 0.0511 | - accuracy: | 0. |
| 9847 - val\_loss: 0.0426 - val\_accuracy: 0.9855 |  |  |  |  |  |
| Epoch 3/5 |  |  |  |  |  |
| 938/938 [==============================] - 20s | 21ms/step | - loss: | 0.0365 | - accuracy: | 0. |
| 9884 - val\_loss: 0.0308 - val\_accuracy: 0.9900 |  |  |  |  |  |
| Epoch 4/5 |  |  |  |  |  |
| 938/938 [==============================] - 20s | 21ms/step | - loss: | 0.0274 | - accuracy: | 0. |
| 9915 - val\_loss: 0.0319 - val\_accuracy: 0.9889 |  |  |  |  |  |
| Epoch 5/5 |  |  |  |  |  |
| 938/938 [==============================] - 20s | 21ms/step | - loss: | 0.0230 | - accuracy: | 0. |
| 9927 - val\_loss: 0.0353 - val\_accuracy: 0.9901 |  |  |  |  |  |

313/313 [==============================] - 1s 4ms/step - loss: 0.0353 - accuracy: 0.99

01

Test Loss: 0.03527578338980675

Test Accuracy: 0.9901000261306763

**RESULT**

Thus character Recognition using CNN is implemented.

**EX.NO:3**  **Face recognition using CNN**

**DATE:**

**AIM:**

To Implement Face recognition using CNN

**PROCEDURE:**

|  |  |
| --- | --- |
| **1** | Set the path to the directory containing the face images |
| **2** | Load the face images and labels & Iterate over the face image directory and load the images |
| **3** | Convert the data to numpy arrays, Preprocess labels to extract numeric part And Convert labels to one-hot encoded vectors |
| **4** | Split the data into training and validation sets |
| **5** | Compile, Train the CNN model and Save the trained model |

**PROGRAM**

import os

import numpy as np import tensorflow as tf

from tensorflow.keras.preprocessing.image import load\_img, img\_to\_array from sklearn.model\_selection import train\_test\_split

# Set the path to the directory containing the face images faces\_dir = "D:/R2021 DL LAB/Faces/Faces"

# Load the face images and labels x\_data = []

y\_data = []

# Iterate over the face image directory and load the images for filename in os.listdir(faces\_dir):

if filename.endswith(".jpg"):

img\_path = os.path.join(faces\_dir, filename)

img = load\_img(img\_path, target\_size=(64, 64)) # Resize images to 64x64 pixels

img\_array = img\_to\_array(img) x\_data.append(img\_array)

label = filename.split(".")[0]

# Assuming the filename format is label.jpg y\_data.append(label)

# Convert the data to numpy arrays

x\_data = np.array(x\_data) y\_data = np.array(y\_data)

# Preprocess labels to extract numeric part

y\_data\_numeric = np.array([int(label.split("\_")[1]) for label in y\_data])

# Convert labels to one-hot encoded vectors num\_classes = len(np.unique(y\_data\_numeric))

y\_data\_encoded = tf.keras.utils.to\_categorical(y\_data\_numeric, num\_classes)

# Split the data into training and validation sets

x\_train, x\_val, y\_train, y\_val = train\_test\_split(x\_data, y\_data\_encoded, test\_size=0.2, random\_state=42)

# Define the CNN architecture for face recognition model = tf.keras.models.Sequential([

tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(64, 64,

3)),

])

tf.keras.layers.MaxPooling2D((2, 2)), tf.keras.layers.Flatten(), tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(num\_classes, activation='softmax')

# Compile the model

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

# Train the CNN model

model.fit(x\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(x\_val, y\_val))

# Save the trained model model.save("face\_recognition\_model.keras")

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **OUTPUT**  Epoch 1/10  65/65 [==============================] | - 5s 68ms/step | | - | loss: | 215.6209 | | - accuracy: 0.0 | |
| 098 - val\_loss: 4.7830 - val\_accuracy:  Epoch 2/10 | 0.0039 | |  |  |  | |  | |
| 65/65 [==============================] | - 4s | 66ms/step | - | loss: | 4.7793 | - | accuracy: | 0.011 |
| 2 - val\_loss: 4.7757 - val\_accuracy: 0.0039 Epoch 3/10  65/65 [==============================] - 4s | | 66ms/step | - loss: | | 4.7717 | - accuracy: | | 0.012 |
| 2 - val\_loss: 4.7694 - val\_accuracy: 0.0039 Epoch 4/10  65/65 [==============================] - 4s | | 66ms/step | - loss: | | 4.7646 | - accuracy: | | 0.010 |
| 7 - val\_loss: 4.7634 - val\_accuracy: 0.0039  Epoch 5/10 | |  |  | |  |  | |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 65/65 [==============================] - 4s | 68ms/step | - loss: | 4.7579 | - accuracy: | 0.010 |
| 2 - val\_loss: 4.7577 - val\_accuracy: 0.0039 |  |  |  |  |  |
| Epoch 6/10 |  |  |  |  |  |
| 65/65 [==============================] - 5s | 70ms/step | - loss: | 4.7516 | - accuracy: | 0.010 |
| 7 - val\_loss: 4.7525 - val\_accuracy: 0.0039 |  |  |  |  |  |
| Epoch 7/10 |  |  |  |  |  |
| 65/65 [==============================] - 4s | 66ms/step | - loss: | 4.7457 | - accuracy: | 0.009 |
| 8 - val\_loss: 4.7476 - val\_accuracy: 0.0039 |  |  |  |  |  |
| Epoch 8/10 |  |  |  |  |  |
| 65/65 [==============================] - 4s | 67ms/step | - loss: | 4.7402 | - accuracy: | 0.010 |
| 7 - val\_loss: 4.7432 - val\_accuracy: 0.0039 |  |  |  |  |  |
| Epoch 9/10 |  |  |  |  |  |
| 65/65 [==============================] - 4s | 66ms/step | - loss: | 4.7349 | - accuracy: | 0.013 |
| 2 - val\_loss: 4.7392 - val\_accuracy: 0.0078 |  |  |  |  |  |
| Epoch 10/10 |  |  |  |  |  |
| 65/65 [==============================] - 4s | 65ms/step | - loss: | 4.7300 | - accuracy: | 0.010 |
| 2 - val\_loss: 4.7354 - val\_accuracy: 0.0078  **RESULT**  Face recognition using CNN is analysed and implemented. |  |  |  |  |  |

**EX.NO:4 Language modeling using RNN**

**DATE:**

**AIM:**

To implement Language modeling using RNN

**PROCEDURE:**

|  |  |
| --- | --- |
| **1** | Create a set of unique characters in the text |
| **2** | Convert text to a sequence of character indices |
| **3** | Create input-output pairs for training |
| **4** | Convert sequences and next\_char to numpy arrays |
| **5** | Train the model and Generate text using the trained model |

**PROGRAM:**

import numpy as np import tensorflow as tf

from tensorflow.keras.models import Sequential

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

# Sample text data

text = "This is a sample text for language modeling using RNN."

# Create a set of unique characters in the text chars = sorted(set(text))

char\_to\_index = {char: index for index, char in enumerate(chars)} index\_to\_char = {index: char for index, char in enumerate(chars)}

# Convert text to a sequence of character indices text\_indices = [char\_to\_index[char] for char in text]

# Create input-output pairs for training seq\_length = 20

sequences = [] next\_char = []

for i in range(0, len(text\_indices) - seq\_length): sequences.append(text\_indices[i : i + seq\_length]) next\_char.append(text\_indices[i + seq\_length])

# Convert sequences and next\_char to numpy arrays X = np.array(sequences)

y = np.array(next\_char)

# Build the RNN model

model = Sequential([

Embedding(input\_dim=len(chars), output\_dim=50, input\_length=seq\_length), SimpleRNN(100, return\_sequences=False),

Dense(len(chars), activation="softmax")

])

model.compile(loss="sparse\_categorical\_crossentropy", optimizer="adam") # Train the model

model.fit(X, y, batch\_size=64, epochs=50)

# Generate text using the trained model seed\_text = "This is a sample te" generated\_text = seed\_text num\_chars\_to\_generate = 100

for \_ in range(num\_chars\_to\_generate):

seed\_indices = [char\_to\_index[char] for char in seed\_text]

# Check if the seed sequence length matches the model's input length if len(seed\_indices) < seq\_length:

diff = seq\_length - len(seed\_indices) seed\_indices = [0] \* diff + seed\_indices

seed\_indices = np.array(seed\_indices).reshape(1, -1) next\_index = model.predict(seed\_indices).argmax() next\_char = index\_to\_char[next\_index]

generated\_text += next\_char

seed\_text = seed\_text[1:] + next\_char print(generated\_text)

**OUTPUT**

Epoch 1/50

1/1 [==============================] - 1s 1s/step - loss: 3.0885

Epoch 2/50

1/1 [==============================] - 0s 8ms/step - loss: 3.0053

Epoch 3/50

1/1 [==============================] - 0s 14ms/step - loss: 2.9234

Epoch 4/50

1/1 [==============================] - 0s 0s/step - loss: 2.8392

Epoch 5/50

1/1 [==============================] - 0s 17ms/step - loss: 2.7501

Epoch 6/50

1/1 [==============================] - 0s 0s/step - loss: 2.6545

Epoch 7/50

1/1 [==============================] - 0s 4ms/step - loss: 2.5519

Epoch 8/50

1/1 [==============================] - 0s 14ms/step - loss: 2.4425

Epoch 9/50

1/1 [==============================] - 0s 0s/step - loss: 2.3266

Epoch 10/50

1/1 [==============================] - 0s 18ms/step - loss: 2.2063

Epoch 11/50

1/1 [==============================] - 0s 8ms/step - loss: 2.0865

Epoch 12/50

1/1 [==============================] - 0s 5ms/step - loss: 1.9717

Epoch 13/50

1/1 [==============================] - 0s 0s/step - loss: 1.8622

Epoch 14/50

1/1 [==============================] - 0s 4ms/step - loss: 1.7552

Epoch 15/50

1/1 [==============================] - 0s 13ms/step - loss: 1.6493

Epoch 16/50

1/1 [==============================] - 0s 0s/step - loss: 1.5457

Epoch 17/50

1/1 [==============================] - 0s 17ms/step - loss: 1.4472

Epoch 18/50

1/1 [==============================] - 0s 0s/step - loss: 1.3554

Epoch 19/50

1/1 [==============================] - 0s 17ms/step - loss: 1.2678

Epoch 20/50

1/1 [==============================] - 0s 0s/step - loss: 1.1810

Epoch 21/50

1/1 [==============================] - 0s 17ms/step - loss: 1.0964

Epoch 22/50

1/1 [==============================] - 0s 14ms/step - loss: 1.0179

Epoch 23/50

1/1 [==============================] - 0s 1ms/step - loss: 0.9459

Epoch 24/50

1/1 [==============================] - 0s 16ms/step - loss: 0.8773

Epoch 25/50

1/1 [==============================] - 0s 0s/step - loss: 0.8107

Epoch 26/50

1/1 [==============================] - 0s 17ms/step - loss: 0.7473

Epoch 27/50

1/1 [==============================] - 0s 0s/step - loss: 0.6884

Epoch 28/50

1/1 [==============================] - 0s 17ms/step - loss: 0.6333

Epoch 29/50

1/1 [==============================] - 0s 0s/step - loss: 0.5809

Epoch 30/50

1/1 [==============================] - 0s 2ms/step - loss: 0.5318

Epoch 31/50

1/1 [==============================] - 0s 17ms/step - loss: 0.4871

Epoch 32/50

1/1 [==============================] - 0s 0s/step - loss: 0.4469

Epoch 33/50

1/1 [==============================] - 0s 18ms/step - loss: 0.4099

Epoch 34/50

1/1 [==============================] - 0s 0s/step - loss: 0.3753

Epoch 35/50

1/1 [==============================] - 0s 18ms/step - loss: 0.3430

Epoch 36/50

1/1 [==============================] - 0s 0s/step - loss: 0.3134

Epoch 37/50

1/1 [==============================] - 0s 15ms/step - loss: 0.2865

Epoch 38/50

1/1 [==============================] - 0s 0s/step - loss: 0.2621

Epoch 39/50

1/1 [==============================] - 0s 2ms/step - loss: 0.2399

Epoch 40/50

1/1 [==============================] - 0s 15ms/step - loss: 0.2200

Epoch 41/50

1/1 [==============================] - 0s 1ms/step - loss: 0.2021

Epoch 42/50

1/1 [==============================] - 0s 18ms/step - loss: 0.1860

Epoch 43/50

1/1 [==============================] - 0s 0s/step - loss: 0.1714

Epoch 44/50

1/1 [==============================] - 0s 16ms/step - loss: 0.1580

Epoch 45/50

1/1 [==============================] - 0s 0s/step - loss: 0.1460

Epoch 46/50

1/1 [==============================] - 0s 4ms/step - loss: 0.1353

Epoch 47/50

1/1 [==============================] - 0s 12ms/step - loss: 0.1257

Epoch 48/50

1/1 [==============================] - 0s 933us/step - loss: 0.1170

Epoch 49/50

1/1 [==============================] - 0s 17ms/step - loss: 0.1090

Epoch 50/50

1/1 [==============================] - 0s 0s/step - loss: 0.1017

**EX.NO:5 Sentiment analysis using LSTM**

**DATE:**

**AIM:**

To implement Sentiment analysis using LSTM

**PROCEDURE**

|  |  |
| --- | --- |
| **1** | Load the IMDB dataset, which consists of movie reviews labeled with positive or negative sentiment. |
| **2** | Preprocess the data by padding sequences to a fixed length (**max\_review\_length**) and limiting the vocabulary size to the most frequent words (**num\_words**). |
| **3** | Build an LSTM-based model. The Embedding layer is used to map word indices to dense vectors, the LSTM layer captures sequence dependencies, and the Dense layer produces a binary sentiment prediction. |
| **4** | The model is compiled with binary cross-entropy loss and the Adam optimizer. |
| **5** | Train the model using the training data. Finally, we evaluate the model on the test data and print the test accuracy.  **PROGRAM**  import numpy as np import tensorflow as tf  from tensorflow.keras.datasets import imdb  from tensorflow.keras.preprocessing import sequence from tensorflow.keras.models import Sequential  from tensorflow.keras.layers import Embedding, LSTM, Dense  # Load the IMDb movie review dataset  max\_features = 5000 # Number of words to consider as features  max\_len = 500 # Maximum length of each review (pad shorter reviews, truncate longer reviews)  (x\_train, y\_train), (x\_test, y\_test) = imdb.load\_data(num\_words=max\_features) x\_train = sequence.pad\_sequences(x\_train, maxlen=max\_len)  x\_test = sequence.pad\_sequences(x\_test, maxlen=max\_len)  # Define the LSTM model  embedding\_size = 32 # Dimensionality of the word embeddings  model = Sequential()  model.add(Embedding(max\_features, embedding\_size, input\_length=max\_len)) model.add(LSTM(100)) # LSTM layer with 100 units  model.add(Dense(1, activation='sigmoid'))  # Compile the model  model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])  # Train the model batch\_size = 64  epochs = 5  model.fit(x\_train, y\_train, batch\_size=batch\_size, epochs=epochs, validation\_data=(x\_test, y\_test))  # Evaluate the model  loss, accuracy = model.evaluate(x\_test, y\_test) print("Loss:", loss)  print("Accuracy:", accuracy)  **OUTPUT**  Downloading data from [https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb](https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz)  [.npz](https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz)  17464789/17464789 [==============================] - 7s 0us/step  Epoch 1/5  391/391 [==============================] - 286s 727ms/step - loss: 0.4991 - accuracy:  0.7626 - val\_loss: 0.3712 - val\_accuracy: 0.8412 Epoch 2/5  391/391 [==============================] - 296s 757ms/step - loss: 0.3381 - accuracy:  0.8587 - val\_loss: 0.3609 - val\_accuracy: 0.8532 Epoch 3/5  391/391 [==============================] - 313s 801ms/step - loss: 0.2642 - accuracy:  0.8945 - val\_loss: 0.3168 - val\_accuracy: 0.8678 Epoch 4/5  391/391 [==============================] - 433s 1s/step - loss: 0.2263 - accuracy: 0.9  142 - val\_loss: 0.3119 - val\_accuracy: 0.8738 Epoch 5/5  391/391 [==============================] - 302s 774ms/step - loss: 0.1982 - accuracy:  0.9247 - val\_loss: 0.3114 - val\_accuracy: 0.8745  782/782 [==============================] - 74s 95ms/step - loss: 0.3114 - accuracy: 0.  8745  Loss: 0.3113741874694824  Accuracy: 0.8745200037956238  **RESULT**  Thus Sentiment analysis using LSTM is implemented. Ex.No:6 Parts of speech tagging using Sequence to Sequence architectureDate |

**AIM:**

To implement Parts of speech tagging using Sequence to Sequence architecture

## Procedure:

## 

|  |  |
| --- | --- |
| **1** | Define the input and output sequences. |
| **2** | Create a set of all unique words and POS tags in the dataset |
| **3** | Add <sos> and <eos> tokens to target\_words. |
| **4** | Create dictionaries to map words and POS tags to integers. |
| **5** | Define the maximum sequence lengths, Prepare the encoder input data And Prepare the decoder input and target data |
| **6** | Define the encoder input and LSTM layers Define the decoder input and LSTM layers |
| **7** | Define, Compile and train the model |
| **8** | Define the encoder model to get the encoder states and Define the decoder model with encoder states as initial state |
| **9** | Define a function to perform inference and generate POS tags, Test the model. |

**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]

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)

**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

## Date

**AIM:**

To implement Machine Translation using Encoder-Decoder model

## PROCEDURE

|  |  |
| --- | --- |
| **1** | Define the input and output sequences. |
| **2** | Create a set of all unique words in the input and target sequences. |
| **3** | Add <sos> and <eos> tokens to target\_words. |
| **4** | Define the maximum sequence lengths Create dictionaries to map words to integers. Define the maximum sequence lengths |
| **5** | Prepare the encoder input data  Prepare the decoder input and target data |
| **6** | Define the encoder input and LSTM layers Define the decoder input and LSTM layers |
| **7** | Define, Compile and train the model |
| **8** | Define the encoder model to get the encoder states  Define the decoder model with encoder states as initial state  **PROGRAM**  import numpy as np    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 GANsDate **AIM:**  To implement Image augmentation using GANs PROCEDURE:  |  |  | | --- | --- | | **1** | Load the MNIST dataset  Normalize and reshape the images Define the generator network | | **2** | Define the discriminator network Compile the discriminator  Combine the generator and discriminator into a single GAN model |   **3** Train the hyperparameters and the training loop has the following steps:   1. Generate a batch of fake images 2. Train the discriminator 3. Train the generator 4. Print the progress and save samples.   **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 the **Image augmentation using GANs** is implemented.