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

DATE:

AIM:

To solve XOR problem using DNN

PROCEDURE:

a. Define the XOR input and outputdata
b. Define the DNNarchitecture
c. Compile the model
d. Train the DNN
e. 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: 0.0039
Epoch 2/10
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
```

CHECK OUTPUT AS IMAGES IN MENTIONED FOLDER

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
modelseed_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
    lengthif 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

This is a sample tentrfornlanguags modnging nsing Rgn.rginsrngangrngangnoggrng
nsingrngingndgg nsinorng ngrngadgsinorng
```

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
sequencefrom 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,
truncatelonger 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

.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:
0.8412Epoch 2/5

391/391 [==============================] - 296s 757ms/step - loss: 0.3381 -
accuracy:

0.8587 - val_loss: 0.3609 - val_accuracy:
0.8532Epoch 3/5

391/391 [==============================] - 313s 801ms/step - loss: 0.2642 -
accuracy:

0.8945 - val_loss: 0.3168 - val_accuracy:
0.8678Epoch 4/5

391/391 [==============================] - 433s 1s/step - loss: 0.2263 -
accuracy: 0.9

142 - val_loss: 0.3119 - val_accuracy:
0.8738Epoch 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 architecture**

Date

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
datasetinput_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
statedecoder_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
tagsdef 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_indexstates_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 ininput_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+00Epoch 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 sequencesinput_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.spl
        it()):
        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.s
    plit()):
        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 translationsdef
translate(input_sequence):
    states_value =
    encoder_model.predict(input_s
    equence)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_inde
        x]
        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 ininput_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**

Date

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 mc

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, Flattenfrom
tensorflow.keras.layers import
BatchNormalization, Dropout from
tensorflow.keras.layers import Conv2D,
Conv2DTranspose
from tensorflow.keras.models
import Sequential, Modelfrom
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)
/ 127.5x_train =
np.expand_dims(x_train, axis=-1)
```

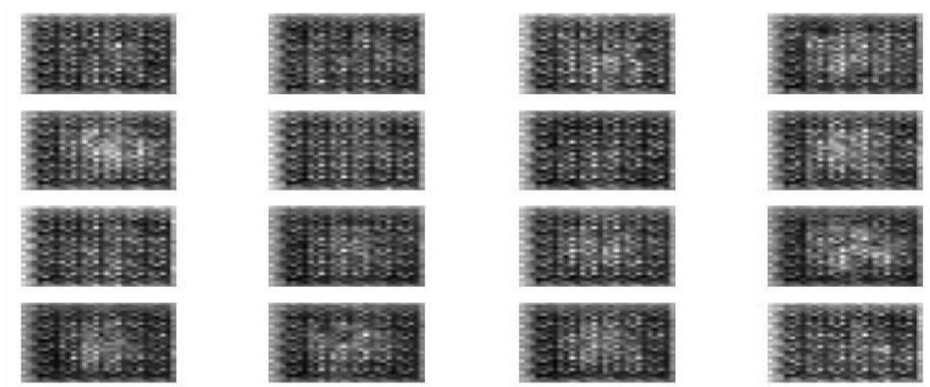
```
#
Define
the
generat
or
network
generat
or =
Sequent
ial()
generator.add(Dense(7 * 7 * 256, input_dim=100))
generator.add(Reshape((7, 7, 256)))
generator.add(BatchNormalization())
generator.add(Conv2DTranspose(128, kernel_size=5, strides=1,
padding='same',activation='relu'))
generator.add(BatchNormalization())
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
discriminat
or network
discriminat
or =
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 modelgan_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))
#
Tra
ini
ng
hyp
erp
ara
met
ers
epo
chs
=
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 samplesif epoch %
    sample_interval == 0:
        print(f'Epoch: {epoch}   Discriminator Loss:
{d_loss[0]}                      GeneratorLoss: {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 += 1plt.show()
```

OUTPUT

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



RESULT

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