ERODE SENGUNTHAR

ENGINEERING COLLEGE

(APPROVED BY AICTE ,NEW DELHI & PERMANENTLY AFFILIATED TO ANNA UNIVERSITY,CHENNAI

ACCREDITED BY NBA,NEW DELHI,NAAC WITH GRADE "A" & IE(I),KOLKATA)

PERUNDURAI,ERODE-638 057.

An Autonomous Institution

BONAFIDE CERTIFICATE

Register No. ……...

Certified that is the Bonafide Record of Work Done

Name of the Student	:	_____
Branch	:	_____
Name of the Lab	:	_____
		_____
Year/Sem	:	_____

Faculty Incharge Head of the Department

Submitted for the Odd Semester Practical

Held on ……..

Internal Examiner External Examiner

INDEX

S.NO	DATE	NAME OF THE EXPERIMENT	PAGE NO	MARKS	FACULTY SIGN
1.		Solving Xor Problem Using DNN			
2.		Character Recognition Using CNN			
3.		Face Recognition Using CNN			
4.		Text Generation Using RNN			
5.		Sentiment Analysis Using LSTM			
6.		Parts Of Speech Tagging Using Sequence To Sequence Architecture			
7.		Machine Translation Using Encoder-Decoder Model			
8.		Image Augmentation Using GAN			

EX.NO: - 1 **SOLVING XOR PROBLEM USING DNN**

DATE: -

AIM:

The aim of this procedure is to solve the XOR problem using a Deep Neural Network (DNN) in Python.

PROCEDURE:

- Import the necessary libraries
- Prepare the dataset
- Create the DNN model
- Compile the model
- Train the model
- Test the model
- Print the predictions

PROGRAM:

#1. Import the necessary libraries:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
#2. Prepare the dataset:
# Define the input dataset
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
# Define the corresponding output labels
y = np.array([[0], [1], [1], [0]])
#3. Create the DNN model:
# Create a Sequential model
model = Sequential()
# Add layers to the model
model.add(Dense(4, input_dim=2, activation='relu'))
```

```
model.add(Dense(1, activation='sigmoid'))
# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
#4. Train the model:
model.fit(X, y, epochs=1000)
#5. Test the model:
# Predict outputs for the input dataset
predictions = model.predict(X)
rounded_predictions = np.round(predictions)
# Print the predictions
for i in range(len(X)):
    print(f'Input: {X[i]}, Predicted Output: {rounded_predictions[i]}')
```

OUTPUT:

```
Input: [0 0], Predicted Output: [0.]
Input: [0 1], Predicted Output: [1.]
Input: [1 0], Predicted Output: [1.]
Input: [1 1], Predicted Output: [0.]
```

RESULT:

Thus the program to solve the XOR problem using a Deep Neural Network (DNN) in Python written Successfully.

EX.NO: - 2 **CHARACTER RECOGNITION USING CNN**

DATE: -

AIM:

The aim of this code is to train a Convolutional Neural Network (CNN) model on the MNIST dataset and evaluate its performance by making predictions on the test set.

PROCEDURE:

- Import the necessary libraries
- Load and preprocess the dataset
- Create the CNN model
- Compile the model
- Train the model
- Evaluate the model
- Make predictions
- Visualize the results

PROGRAM:

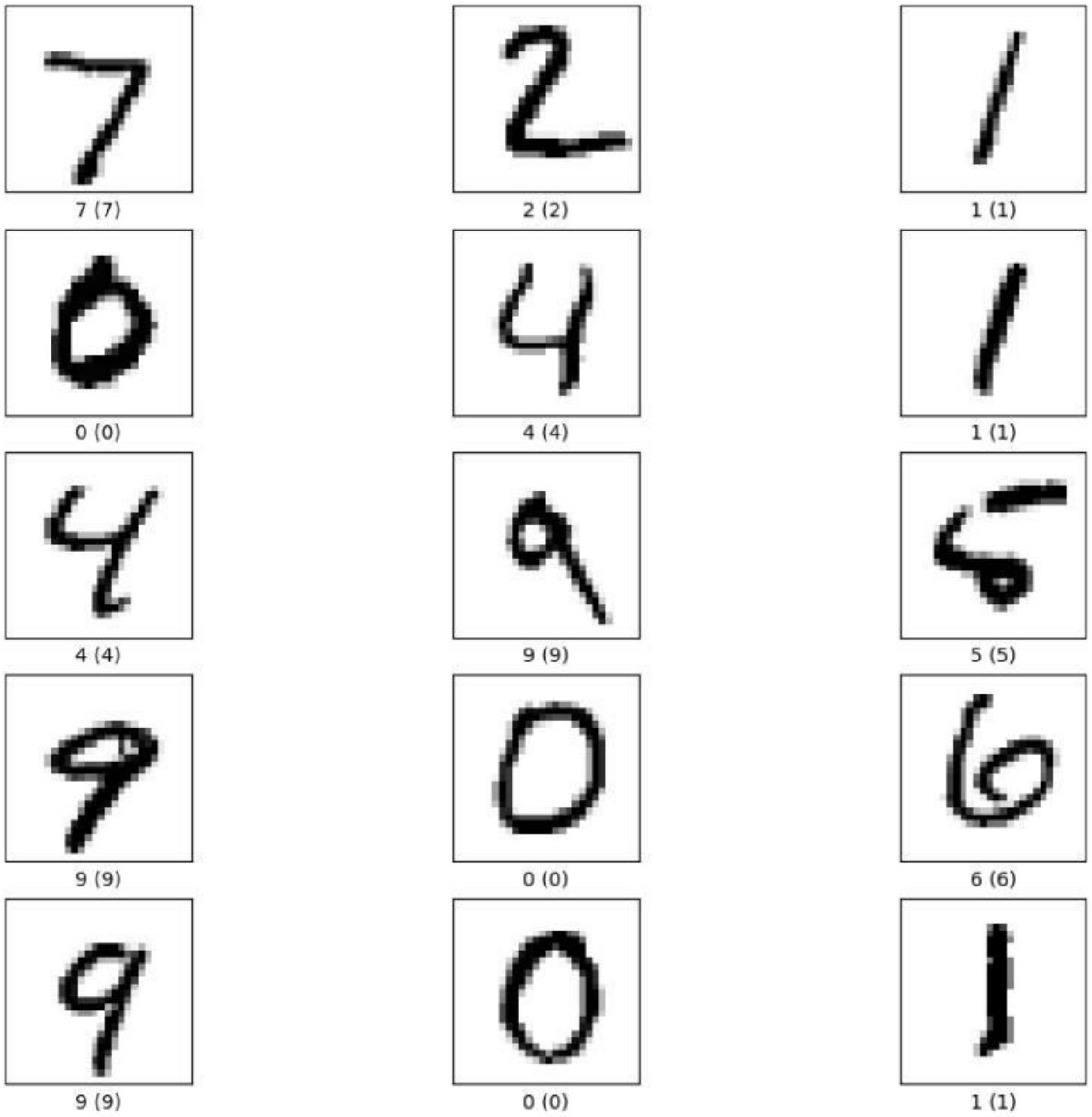
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Normalize the data
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255

# Reshape the data for CNN
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
```

```
# Create the model
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(filters=64, kernel_size=(2,2), strides=(1, 1), padding='same', activation='relu', input_shape=(28,28,1)),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Conv2D(filters=32, kernel_size=(2,2), strides=(1, 1), padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation='softmax')])
# Compile the model
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
 # Train the model
model.fit(x_train, y_train, batch_size=60, epochs=10, verbose=1, validation_split=0.3)
 # Evaluate the model on test set
score = model.evaluate(x_test, y_test, verbose=0)
print('\nTest accuracy:', score[1])
 # Make predictions on the test set
predictions = model.predict(x_test)
 # Display some test images and their predicted labels
num_rows = 5
num_cols = 3
num_images = num_rows * num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
```

```
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(x_test[i].reshape(28, 28), cmap=plt.cm.binary)
    predicted_label = np.argmax(predictions[i])
    plt.xlabel("{} ({})".format(predicted_label, y_test[i]))
plt.show()
```

OUTPUT:

```
Epoch 1/10
700/700 [==============================] - 44s 62ms/step - loss: 0.4044 - accuracy: 0.8701 - val_loss: 0.1232 - val_accuracy: 0.9623
.
.
.
.
Epoch 10/10
700/700 [==============================] - 36s 51ms/step - loss: 0.0545 - accuracy: 0.9829 - val_loss: 0.0356 - val_accuracy: 0.9889

Test accuracy: 0.9901999831199646
313/313 [==============================] - 2s 6ms/step
```



RESULT:

Thus the python code is to train a Convolutional Neural Network (CNN) model on the MNIST dataset and evaluate its performance by making predictions on the test set executed successfully.

EX.NO: - 3 **FACE RECOGNITION USING CNN**

DATE: -

AIM:

The aim of this program is to perform real-time face detection using the OpenCV library in Python. It captures video from your default camera (usually the webcam), detects faces in the video frames, and highlights the detected faces with green rectangles.

PROCEDURE:

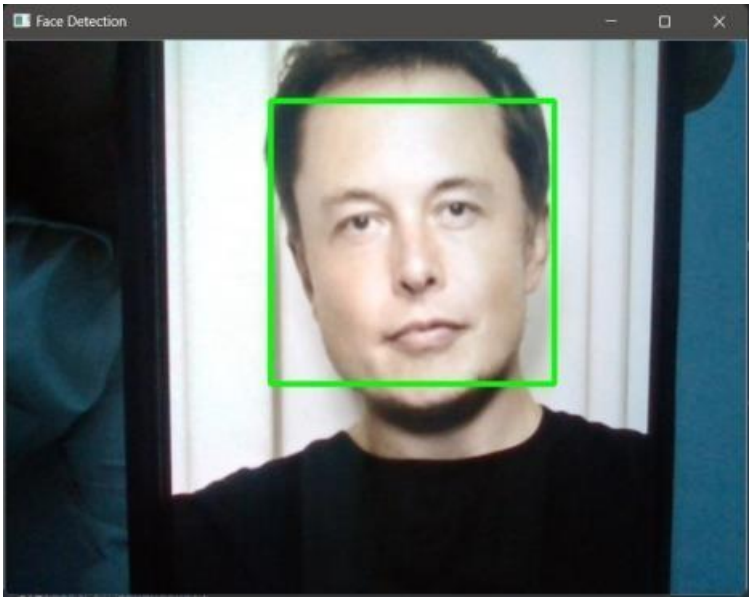
1. Import the OpenCV library.
2. Create a CascadeClassifier object and load the pre-trained face detection model (download -'haarcascade_frontalface_default.xml').
3. Initialize the video capture using the default camera (usually the webcam).
4. Start an infinite loop to continuously capture and process frames from the camera.
5. Read a frame from the camera.
6. Convert the frame to grayscale.
7. Use the CascadeClassifier to detect faces in the grayscale frame.
8. For each detected face, draw a green rectangle around it.
9. Display the frame with detected faces in a window called 'Face Detection.'
10. Check for the 'q' key press; if it's pressed, exit the loop.
11. Release the video capture and close all OpenCV windows when the loop is exited.

PROGRAM:

```
import cv2

# Load the pre-trained face detection model

face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')

# Initialize the video capture

cap = cv2.VideoCapture(0)

while True:

    ret, frame = cap.read()  # Read a frame from the camera
```

```
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)  # Convert the frame to grayscale
    # Detect faces in the grayscale frame
    faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5,
minSize=(30, 30))
    # Draw green rectangles around detected faces
    for (x, y, w, h) in faces:
        cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 3)
    # Display the frame with detected faces
    cv2.imshow('Face Detection', frame)
    # Check for 'q' key press; if pressed, exit the loop
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
# Release the video capture and close OpenCV windows
cap.release()
cv2.destroyAllWindows()
```

OUTPUT:



RESULT:

The program provides real-time face detection in the video feed from your camera, and the output is a window displaying the live video with highlighted faces.

EX.NO: - 4 **TEXT GENERATION USING RNN**

DATE: -

AIM:

The aim of this Python code is to train a text generation model using RNNand generate text based on a given input text.

PROCEDURE:

1. Read and process a text file (in this case, a Shakespearean text).
2. Tokenize the text, create a character-to-index mapping, and convert the text to numerical values.
3. Create a dataset for training the text generation model.
4. Build a deep learning model using TensorFlow's Keras API. The model consists of an embedding layer, two LSTM layers, dropout layers, batch normalization, and a dense output layer.
5. Define a custom loss function for training the model.
6. Generate text using the trained model, starting from a user-provided input text.

PROGRAM:

```
text(file_path):
    text = open(file_path, 'rb').read().decode(encoding='utf-8')
    vocab = sorted(set(text))
    char2idx = {u: i for i, u in enumerate(vocab)}
    idx2char = np.array(vocab)
    text_as_int = np.array([char2idx[c] for c in text])
    return text_as_int, vocab, char2idx, idx2char

def split_input_target(chunk):
    input_text, target_text = chunk[:-1], chunk[1:]
    return input_text, target_text

def create_dataset(text_as_int, seq_length=100, batch_size=64, buffer_size=10000):
    char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)
    dataset = char_dataset.batch(seq_length + 1, drop_remainder=True).map(split_input_target)
    dataset = dataset.shuffle(buffer_size).batch(batch_size, drop_remainder=True)
    return dataset

def build_model(vocab_size, embedding_dim=256, rnn_units=1024):
    model = tf.keras.Sequential([
        tf.keras.layers.Embedding(vocab_size, embedding_dim),
        tf.keras.layers.LSTM(rnn_units, return_sequences=True, recurrent_initializer='glorot_uniform'),
        tf.keras.layers.Dropout(0.1),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.LSTM(rnn_units, return_sequences=True, recurrent_initializer='glorot_uniform'),
        tf.keras.layers.Dropout(0.1),
        tf.keras.layers.BatchNormalization(),
```

```
        tf.keras.layers.Dense(vocab_size)
    ])
    return model

def loss(labels, logits):
    return tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=True)

def generate_text(model, char2idx, idx2char, start_string, generate_char_num=1000,
temperature=1.0):
    input_eval = [char2idx[s] for s in start_string]
    input_eval = tf.expand_dims(input_eval, 0)
    text_generated = []

    for i in range(generate_char_num):
        predictions = model(input_eval)
        predictions = tf.squeeze(predictions, 0)
        predictions /= temperature
        predicted_id = tf.random.categorical(predictions, num_samples=1)[-1, 0].numpy()
        input_eval = tf.expand_dims([predicted_id], axis=0)
        text_generated.append(idx2char[predicted_id])

    return start_string + ''.join(text_generated)

path_to_file = tf.keras.utils.get_file('shakespeare.txt',
'https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt')

text_as_int, vocab, char2idx, idx2char = process_text(path_to_file)
dataset = create_dataset(text_as_int)
model = build_model(vocab_size=len(vocab))
model.compile(optimizer='adam', loss=loss)
model.summary()
history = model.fit(dataset, epochs=1)
model.save_weights("gen_text_weights.h5", save_format='h5')

model = build_model(vocab_size=len(vocab))
model.load_weights("gen_text_weights.h5")
model.build(tf.TensorShape([1, None]))
model.summary()

user_input = input("Write the beginning of the text, the program will complete it. Your input is: ")
generated_text = generate_text(model, char2idx, idx2char, start_string=user_input,
generate_char_num=2000)
print(generated_text)
```

OUTPUT:

" First Citizen:We are accounted poor citizens, the patricians good.What authority surfeits on would relieve us: if they would yield us but the superfluity, while it were wholesome, we might guess they relieved us humanely;but they think we are too dear: the leanness that afflicts us, the object of our misery, is as an inventory to particularise their abundance; our sufferance is a gain to them Let us revenge this with our pikes, ere we become rakes: for the gods know I speak this in hunger for bread, not in thirst for revenge. "

RESULT:

The code demonstrates how to train a text generation model and use it to create coherent text based on a given starting point.

EX.NO: - 5 **SENTIMENT ANALYSIS USING LSTM**

DATE: -

AIM:

The aim of this code is to perform sentiment analysis on a dataset of comments using deep learning techniques.

PROCEDURE:

1. Import necessary libraries including NumPy, Pandas, Matplotlib, Hazm (a Persian natural language processing library), and scikit-learn.

2. Read a dataset from a CSV file called 'Snappfood - Sentiment Analysis.csv'.

3. Perform data preprocessing, including tokenization, lemmatization, normalization, and removing stopwords and punctuations.

4. Create a new dataset containing the cleaned text and sentiment labels.

5. Tokenize and pad the text data for model training.

6. Build a deep learning model using Keras with an embedding layer, Bidirectional LSTM layers, SpatialDropout1D, BatchNormalization, and a Dense output layer.

7. Compile the model with the Adam optimizer and binary cross-entropy loss.

8. Train the model on the training data.

9. Make predictions on both the training and test data.

10. Evaluate the model using classification reports to measure its performance.

PROGRAM:

```
import numpy as np
import pandas as pd
import string
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.utils import pad_sequences
from keras.models import Sequential
from keras.layers import Embedding, Bidirectional, LSTM, Dense, Dropout, SpatialDropout1D,
BatchNormalization

data = pd.read_csv('/content/synthetic_persian_sentiment.csv', delimiter=',', on_bad_lines='skip')
data = data[['text', 'label']].dropna().reset_index(drop=True)

punctuations = string.punctuation + "،" + "؛"
translator = str.maketrans('', '', punctuations)

def clean_text(text):
text = str(text).lower()
```

```
text = text.translate(translator)
text = " ".join(text.split())
return text

data['Text'] = data['text'].apply(clean_text)
data.info()
data

from tensorflow.keras.utils import to_categorical
le = LabelEncoder()
data['label_encoded'] = le.fit_transform(data['label'])
num_classes = len(le.classes_)
Y = to_categorical(data['label_encoded'], num_classes=num_classes)

X = data['Text']
xtrain, xtest, ytrain, ytest = train_test_split(X, Y, test_size=0.01, random_state=42)

max_words = 5000
max_len = 100
tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(xtrain)

def tokenize_and_pad(texts, maxlen=max_len):
seq = tokenizer.texts_to_sequences(texts)
return pad_sequences(seq, padding='post', maxlen=maxlen)

xtrain_pad = tokenize_and_pad(xtrain)
xtest_pad = tokenize_and_pad(xtest)

model = Sequential()
model.add(Embedding(input_dim=max_words, output_dim=128, input_length=max_len))
model.add(Bidirectional(LSTM(128, return_sequences=True, dropout=0.2)))
model.add(SpatialDropout1D(0.2))
model.add(Bidirectional(LSTM(64, dropout=0.2)))
model.add(BatchNormalization())
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))   # <-- multiclass

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()

history = model.fit(
xtrain_pad, ytrain,
validation_data=(xtest_pad, ytest),
epochs=5,
batch_size=32
)

history = model.fit(
xtrain_pad, ytrain,
validation_data=(xtest_pad, ytest),
epochs=5,
```

```
batch_size=32
)
ypred_test = np.argmax(model.predict(xtest_pad), axis=1)
ytrue_test = np.argmax(ytest, axis=1)
print("Test Report\n", classification_report(ytrue_test, ypred_test, target_names=le.classes_))
```

OUTPUT:

The output includes the model summary, training and evaluation reports, and the model's predictions.

	Unnamed: 0	comment	label	label_id
0	NaN	واقعا حیف وقت که بنویسم سرویس دهیتون شده افتضاح	SAD	1.0
1	NaN	...قرار بود ا ساعته برسه ولی نیم ساعت زودتر از مو	HAPPY	0.0
2	NaN	...قیمت این مدل اصلا با کیفیتش سازگاری نداره، فقط	SAD	1.0
3	NaN	...عاللللی بود همه چه درست و به اندازه و کیفیت خوب	HAPPY	0.0
4	NaN	.شیرینی وانیلی فقط یک مدل بود	HAPPY	0.0
...	...	...	...	...
69995	NaN	...سلام من به فاکتور غذاهایی که سفارش میدم احتیاج	SAD	1.0
69996	NaN	...سایز پیتزا نسبت به سفارشاتی که قبلا گذشتم کم ش	SAD	1.0
69997	NaN	... من قارچ اضافه رو اضافه کرده بودم بودم اما اگر	HAPPY	0.0
69998	NaN	...همرو بعد ۲ساعت تاخیر اشتباه آوردن پولشم رفت رو	SAD	1.0
69999	NaN	.فلفلش خیییلی تند بود	HAPPY	0.0

70000 rows × 4 columns

```
array([[  77,   18,  554, ...,    0,    0,    0],
       [ 128,   41,  334, ...,    0,    0,    0],
       [ 300, 1577,   93, ...,    0,    0,    0],
       ...,
       [   9,   11,    1, ...,    0,    0,    0],
       [  35,  222,  289, ...,    0,    0,    0],
       [   5,    2,  172, ...,    0,    0,    0]], dtype=int32)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 231, 80)	144800
bidirectional (Bidirectional)	(None, 231, 512)	690176
spatial_dropout1d (SpatialDropout1D)	(None, 231, 512)	0
bidirectional_1 (Bidirectional)	(None, 256)	656384
batch_normalization (BatchNormalization)	(None, 256)	1024
dense (Dense)	(None, 1)	257

Total params: 1,492,641

Trainable params: 1,492,129
Non-trainable params: 512

Epoch 1/20
459/459 [==============================] - 99s 194ms/step - loss: 0.4216 - accura
cy: 0.8075 - val_loss: 0.4008 - val_accuracy: 0.8245
Epoch 2/20
459/459 [==============================] - 67s 147ms/step - loss: 0.3860 - accura
cy: 0.8295 - val_loss: 0.3657 - val_accuracy: 0.8331
.

.

.

.

Epoch 20/20
459/459 [==============================] - 64s 138ms/step - loss: 0.2415 - accura
cy: 0.8986 - val_loss: 0.4892 - val_accuracy: 0.8187

22/22 [==============================] - 1s 25ms/step

	precision	recall	f1-score	support
0.0	0.82	0.81	0.81	340
1.0	0.82	0.83	0.82	355
accuracy			0.82	695
macro avg	0.82	0.82	0.82	695
weighted avg	0.82	0.82	0.82	695

RESULT:

The code provides a sentiment analysis model capable of classifying text comments into sentiment categories. The result includes various metrics like accuracy, precision, recall, and F1-score to assess the model's performance on both the training and test datasets.

EX.NO:- 6 PARTS OF SPEECH TAGGING USING SEQUENCE TO

DATE:- SEQUENCE ARCHITECTURE

AIM: -

The aim of the provided Python code is to perform Part-of-Speech (POS) tagging on a given text using a Recurrent Neural Network (RNN) model.

PROCEDURE: -

1. Import the necessary libraries and modules, including NLTK, numpy, Keras, and related functions.

2. Define a paragraph of text that you want to perform POS tagging on.

3. Tokenize the paragraph into words using NLTK's `word_tokenize` function.

4. Perform POS tagging on the tokenized words using NLTK's `pos_tag` function.

5. Create a vocabulary and encode words into integers using Keras's `Tokenizer` and `texts_to_sequences` functions.

6. Create a vocabulary for POS tags and encode them into integers.

7. Prepare the data for training by converting the word sequences and POS tag sequences into numpy arrays.

8. Define an RNN model using Keras with embedding, SimpleRNN, and TimeDistributed layers.

9. Compile the model with the appropriate loss and optimizer.

10. Train the model on the data.

11. Use the trained model to predict POS tags for the input text.

12. Extract the predicted POS tags and display them alongside the original words.

PROGRAM: -

```
import nltk

import numpy as np

from nltk.tokenize import word_tokenize

from nltk.corpus import brown

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad_sequences

from keras.models import Sequential
```

```
from keras.layers import Embedding, SimpleRNN, Dense, TimeDistributed

from keras.utils import to_categorical

nltk.download('punkt')

nltk. download('averaged_perceptron_tagger')


paragraph = """

Natural language processing (NLP) is a field of computer science, artificial intelligence,

and computational linguistics concerned with the interactions between computers and human (natural) languages.

NLP focuses on the interaction between humans and computers using natural language.

The ultimate goal of NLP is to read, decipher, understand, and make sense of the human language in a manner

that is both valuable and meaningful. Most NLP techniques rely on machine learning to derive meaning from human languages.

"""

pip install nltk

nltk.download('punkt_tab')

nltk.download('averaged_perceptron_tagger_eng')

nltk.download('averaged_perceptron_tagger')

words = word_tokenize(paragraph)

pos_tags = nltk.pos_tag(words)

tokenizer = Tokenizer()

tokenizer.fit_on_texts(words)

word_sequences = tokenizer.texts_to_sequences(words)

word_sequences = pad_sequences(word_sequences, padding='post')

pos_tags_set = set(tag for word, tag in pos_tags)

num_pos_tags = len(pos_tags_set)

pos_tag_to_idx = {tag: i for i, tag in enumerate(pos_tags_set)}

pos_tag_sequences = [pos_tag_to_idx[tag] for word, tag in pos_tags]

X = np.array(word_sequences)

Y = np.array(pos_tag_sequences)

model = Sequential()
```

```
model.add(Embedding(input_dim=len(tokenizer.word_index) + 1, output_dim=32,
input_length=X.shape[1]))

model.add(SimpleRNN(64, return_sequences=True))

model.add(TimeDistributed(Dense(num_pos_tags, activation='softmax')))

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

predictions = model.predict(X)

predicted_pos_tags = []

for i in range(len(X)):

    predicted_pos_tags.append([list(pos_tag_to_idx.keys())[list(pos_tag_to_idx.values()).index(tag)] for
tag in predictions[i].argmax(axis=-1)])

for word, pos_tag_list in zip(words, predicted_pos_tags): print(f"{word}: {pos_tag_list}")
```

OUTPUT: -

The output of the code will display the original text with their predicted POS tags.

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.
True
Epoch 1/10
3/3 - 2s - loss: 2.9998 - accuracy: 0.0460 - 2s/epoch - 501ms/step
Epoch 2/10
3/3 - 0s - loss: 2.9815 - accuracy: 0.0812 - 14ms/epoch - 5ms/step
Epoch 3/10
3/3 - 0s - loss: 2.9644 - accuracy: 0.1119 - 12ms/epoch - 4ms/step
Epoch 4/10
3/3 - 0s - loss: 2.9466 - accuracy: 0.1167 - 12ms/epoch - 4ms/step
Epoch 5/10
3/3 - 0s - loss: 2.9289 - accuracy: 0.1403 - 13ms/epoch - 4ms/step
Epoch 6/10
3/3 - 0s - loss: 2.9101 - accuracy: 0.1444 - 13ms/epoch - 4ms/step
```

Epoch 7/10

3/3 - 0s - loss: 2.8902 - accuracy: 0.1526 - 12ms/epoch - 4ms/step

Epoch 8/10

3/3 - 0s - loss: 2.8690 - accuracy: 0.1485 - 16ms/epoch - 5ms/step

Epoch 9/10

3/3 - 0s - loss: 2.8466 - accuracy: 0.1470 - 12ms/epoch - 4ms/step

Epoch 10/10

3/3 - 0s - loss: 2.8226 - accuracy: 0.1579 - 19ms/epoch - 6ms/step

Natural: ['JJ']

language: ['NN']

processing: ['NN']

(: ['(']

NLP: ['NN']

): [')']

is: ['VBZ']

a: ['DT']

field: ['NN']

of: ['IN']

computer: ['NN']

science: ['NN']

,: [',']

artificial: ['JJ']

intelligence: ['NN']

,: [',']

and: ['NN']

computational: ['JJ']

linguistics: ['NN']

concerned: ['NN']

with: ['IN']

the: ['DT']

interactions: ['NN']

between: ['NN']

computers: ['NN']

and: ['NN']

RESULT: -

The RNN model is trained to predict POS tags for words in the input text. The result consists of the predicted POS tags associated with each word in the given text.

EX.NO: -7 MACHINE TRANSLATION USING ENCODER-DECODER MODEL

DATE: -

AIM:

The aim of this code is to create a machine translation model from English to French using a sequence-to-sequence architecture with LSTM layers.

PROCEDURE:

1. Import necessary libraries, including NumPy, Pandas, and TensorFlow's Keras module.
2. Define hyperparameters such as batch size, epochs, latent dimension, and the number of samples.
3. Load the data from the 'fra.txt' file and preprocess it to create input and target sequences.
4. Build token dictionaries for both input and target characters.
5. Create one-hot encoded data for encoder and decoder inputs as well as decoder targets.
6. Define the encoder and decoder models.
7. Compile the model for training using the RMSprop optimizer and categorical cross-entropy loss.
8. Train the model on the training data.
9. Save the trained model to a file named 'eng2french.h5'.
10. Define sampling models for inference.
11. Implement a decoding function to translate English input sentences to French.
12. Iterate over a set of input sentences and generate translations.

PROGRAM:

```
import numpy as np
import pandas as pd
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, LSTM, Dense
import os


# Hyperparameters
batch_size = 64
epochs = 40
latent_dim = 256  # here latent dim represent hidden state or cell state
num_samples = 10000
```

```
# Create sample dataset for English to French translation
sample_data = [
    "Hello\tBonjour",
    "Good morning\tBonjour",
    "How are you?\tComment ça va ?",
    "Thank you\tMerci",
    "Goodbye\tAu revoir",
    "Yes\tOui",
    "No\tNon",
    "Please\tS'il vous plaît",
    "Sorry\tDésolé",
    "I love you\tJe t'aime",
    "My name is\tJe m'appelle",
    "What is your name?\tComment vous appelez-vous ?",
    "Where is the bathroom?\tOù sont les toilettes ?",
    "How much does it cost?\tCombien ça coûte ?",
    "I don't understand\tJe ne comprends pas",
    "Can you help me?\tPouvez-vous m'aider ?",
    "I'm hungry\tJ'ai faim",
    "Water\tEau",
    "Food\tNourriture",
    "Coffee\tCafé",
    "Tea\tThé",
    "Milk\tLait",
    "Bread\tPain",
    "Cheese\tFromage",
    "Wine\tVin",
    "Beer\tBière",
    "One\tUn",
    "Two\tDeux",
    "Three\tTrois",
    "Four\tQuatre",
```

"Five\tCinq",

"Ten\tDix",

"Twenty\tVingt",

"Hundred\tCent",

"Today\tAujourd'hui",

"Tomorrow\tDemain",

"Yesterday\tHier",

"Now\tMaintenant",

"Later\tPlus tard",

"Never\tJamais",

"Always\tToujours",

"Maybe\tPeut-être",

"Certainly\tCertainement",

"Beautiful\tBeau",

"Ugly\tLaid",

"Big\tGrand",

"Small\tPetit",

"Hot\tChaud",

"Cold\tFroid",

"Good\tBon",

"Bad\tMauvais",

"Easy\tFacile",

"Difficult\tDifficile",

"Fast\tRapide",

"Slow\tLent",

"Young\tJeune",

"Old\tVieux",

"New\tNouveau",

"Old (thing)\tVieille",

"Expensive\tCher",

"Cheap\tBon marché",

"Rich\tRiche",

"Poor\tPauvre",

"Happy\tHeureux",

"Sad\tTriste",

"Angry\tEn colère",

"Tired\tFatigué",

"Excited\tExcited",

"Bored\tEnnuyé",

"Surprised\tSurpris",

"Scared\tEffrayé",

"Brave\tCourageux",

"Cowardly\tLâche",

"Strong\tFort",

"Weak\tFaible",

"Healthy\tEn bonne santé",

"Sick\tMalade",

"Clean\tPropre",

"Dirty\tSale",

"Wet\tMouillé",

"Dry\tSec",

"Full\tPlein",

"Empty\tVide",

"Heavy\tLourd",

"Light\tLéger",

"Dark\tSombre",

"Bright\tClair",

"Noisy\tBruyant",

"Quiet\tCalme",

"Crowded\tBondé",

"Empty\tVide",

"Dangerous\tDangereux",

"Safe\tSûr",

"Open\tOuvert",

"Closed\tFermé",

"Free\tGratuit",

```
    "Busy\tOccupé",
    "Early\tTôt",
    "Late\tTard",
    "First\tPremier",
    "Last\tDernier",
    "Next\tProchain",
    "Previous\tPrécédent"
]

# Write sample data to file
with open("fra.txt", "w", encoding="utf-8") as f:
    for line in sample_data:
        f.write(line + "\tCC-BY\n")

data_path = "fra.txt"

# 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)]:
    if line.strip() and '\t' in line:  # Check if line is not empty and contains tab
        parts = line.split('\t')
        if len(parts) >= 2:  # Ensure there are at least 2 parts
            input_text, target_text = parts[0], parts[1]
            # 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)

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)):
    # Process encoder input
    for t, char in enumerate(input_text):
        encoder_input_data[i, t, input_token_index[char]] = 1.0
    # Pad the rest with spaces
    for t in range(len(input_text), max_encoder_seq_length):
        encoder_input_data[i, t, input_token_index[' ']] = 1.0

    # Process decoder input and target
    for t, char in enumerate(target_text):
        decoder_input_data[i, t, target_token_index[char]] = 1.0
        if t > 0:
            # decoder_target_data is ahead of decoder_input_data by one timestep
            decoder_target_data[i, t - 1, target_token_index[char]] = 1.0

    # Pad the rest with spaces
    for t in range(len(target_text), max_decoder_seq_length):
        decoder_input_data[i, t, target_token_index[' ']] = 1.0
        if t > 0:
            decoder_target_data[i, t - 1, target_token_index[' ']] = 1.0

# Define an input sequence and process it.
encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
```

```
# We discard encoder_outputs and only keep the states.
encoder_states = [state_h, state_c]

# Set up the decoder, using encoder_states as initial state.
decoder_inputs = Input(shape=(None, num_decoder_tokens))
# We set up our decoder to return full output sequences,
# and to return internal states as well. We don't use the
# return states in the training model, but we will use them in inference.
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)

# Define the model that will turn
# encoder_input_data & decoder_input_data into decoder_target_data
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

# Run training
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
)

model.save('eng2french.h5')

# Define sampling models
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-lookup token index to decode sequences back to
# something readable.
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())

def decode_sequence(input_seq):
    # Encode the input as state vectors.
    states_value = encoder_model.predict(input_seq, verbose=0)

    # Generate empty target sequence of length 1.
    target_seq = np.zeros((1, 1, num_decoder_tokens))
    # Populate the first character of target sequence with the start character.
    target_seq[0, 0, target_token_index['\t']] = 1.0

    # Sampling loop for a batch of sequences
    # (to simplify, here we assume a batch of size 1).
    stop_condition = False
    decoded_sentence = ''

    while not stop_condition:
        output_tokens, h, c = decoder_model.predict(
```

```
            [target_seq] + states_value, verbose=0
        )

        # Sample a token
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_char = reverse_target_char_index[sampled_token_index]
        decoded_sentence += sampled_char

        # Exit condition: either hit max length
        # or find stop character.
        if (sampled_char == '\n' or
            len(decoded_sentence) > max_decoder_seq_length):
            stop_condition = True

        # Update the target sequence (of length 1).
        target_seq = np.zeros((1, 1, num_decoder_tokens))
        target_seq[0, 0, sampled_token_index] = 1.0

        # Update states
        states_value = [h, c]

    return decoded_sentence

# Test the model
for seq_index in range(min(20, len(input_texts))):  # Test on first 20 samples
    input_seq = encoder_input_data[seq_index: seq_index + 1]
    decoded_sentence = decode_sequence(input_seq)
    print('-')
    print('Input sentence:', input_texts[seq_index])
    print('Decoded sentence:', decoded_sentence)
```

OUTPUT:

```
Number of samples: 10000
Number of unique input tokens: 71
```

Number of unique output tokens: 93

Max sequence length for inputs: 15

Max sequence length for outputs: 59

Epoch 1/40

125/125 [==============================] - 11s 25ms/step - loss: 1.2294 - accuracy: 0.7319 - val_loss: 1.2328 - val_accuracy: 0.7112

Epoch 2/40

125/125 [==============================] - 2s 13ms/step - loss: 0.9437 - accuracy: 0.7476 - val_loss: 1.0409 - val_accuracy: 0.7087

.

.

Epoch 40/40

125/125 [==============================] - 1s 11ms/step - loss: 0.3223 - accuracy: 0.9032 - val_loss: 0.4583 - val_accuracy: 0.8676

Input sentence: Go.

Decoded sentence: Pars !

1/1 [==============================] - 0s 16ms/step

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

1/1 [==============================] - 0s 21ms/step

1/1 [==============================] - 0s 18ms/step

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

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

1/1 [==============================] - 0s 18ms/step

1/1 [==============================] - 0s 21ms/step

RESULT:

The code trains a machine translation model that can translate English sentences into French. The result includes the translations of input sentences provided in the 'fra.txt' dataset.

EX.NO: -8 IMAGE AUGMENTATION USING GAN

DATE:-

AIM:

The aim of this code is to train a Generative Adversarial Network (GAN) to generate synthetic images that resemble handwritten digits from the MNIST dataset.

PROCEDURE:

1. Import necessary libraries and set up constants for image dimensions, channels, and noise vector size.

2. Define a generator model and a discriminator model using Keras.

3. Compile the discriminator model with binary cross-entropy loss and the Adam optimizer.

4. Create the generator model, which takes a noise vector as input and generates images.

5. Create a combined GAN model where the generator is trained to fool the discriminator.

6. Compile the combined model with binary cross-entropy loss and the Adam optimizer.

7. Load the MNIST dataset and preprocess it.

8. Define labels for real and fake images.

9. Training loop:

a. Train the discriminator:

- Select a random batch of real images from the dataset.

- Generate a batch of fake images using the generator.

- Calculate the discriminator loss for both real and fake images and update the discriminator's weights.

b. Train the generator:

- Generate a new batch of fake images.

- Calculate the generator loss by trying to make the discriminator classify the fake images as real.

c. Print and store losses and accuracies at regular intervals.

d. Generate and save sample images at regular intervals.

PROGRAM:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Input, Dense, Reshape, Flatten, Conv2D, Conv2DTranspose,
LSTM, Embedding
from tensorflow.keras.layers import LeakyReLU, BatchNormalization, Dropout,
GlobalAveragePooling1D
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import img_to_array, load_img
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
from PIL import Image
import warnings
warnings.filterwarnings('ignore')

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
tf.get_logger().setLevel('ERROR')

class UniversalGAN:
    def __init__(self, data_type='image', input_shape=None, z_dim=100, data_path=None):
        self.data_type = data_type
        self.input_shape = input_shape
        self.z_dim = z_dim
        self.data_path = data_path
        self.data = None
```

```
        if input_shape is None:
            self.input_shape = self._auto_detect_shape()

        self._build_models()

    def _auto_detect_shape(self):
        if self.data_type == 'image':
            return (28, 28, 1)
        elif self.data_type == 'text':
            return (100,)
        elif self.data_type == 'audio':
            return (128,)
        else:
            return (100,)

    def _build_models(self):
        if self.data_type == 'image':
            self._build_image_models()
        elif self.data_type == 'text':
            self._build_text_models()
        elif self.data_type == 'audio':
            self._build_audio_models()
        else:
            self._build_generic_models()

    def _build_image_models(self):
        self.generator = Sequential([
            Dense(128, input_dim=self.z_dim),
            LeakyReLU(alpha=0.01),
            Dense(256),
            LeakyReLU(alpha=0.01),
            Dense(np.prod(self.input_shape), activation='tanh'),
            Reshape(self.input_shape)
```

```
        ])

        self.discriminator = Sequential([
            Flatten(input_shape=self.input_shape),
            Dense(512),
            LeakyReLU(alpha=0.01),
            Dropout(0.3),
            Dense(256),
            LeakyReLU(alpha=0.01),
            Dropout(0.3),
            Dense(1, activation='sigmoid')
        ])

        self._compile_models()

    def _build_text_models(self):
        self.generator = Sequential([
            Dense(256, input_dim=self.z_dim),
            LeakyReLU(alpha=0.01),
            Dense(512),
            LeakyReLU(alpha=0.01),
            Dense(self.input_shape[0], activation='tanh')
        ])

        self.discriminator = Sequential([
            Dense(512, input_shape=self.input_shape),
            LeakyReLU(alpha=0.01),
            Dropout(0.3),
            Dense(256),
            LeakyReLU(alpha=0.01),
            Dropout(0.3),
            Dense(1, activation='sigmoid')
        ])
```

```
        self._compile_models()

    def _build_audio_models(self):
        self._build_generic_models()

    def _build_generic_models(self):
        self.generator = Sequential([
            Dense(256, input_dim=self.z_dim),
            LeakyReLU(alpha=0.01),
            BatchNormalization(),
            Dense(512),
            LeakyReLU(alpha=0.01),
            BatchNormalization(),
            Dense(1024),
            LeakyReLU(alpha=0.01),
            BatchNormalization(),
            Dense(np.prod(self.input_shape), activation='tanh'),
            Reshape(self.input_shape) if len(self.input_shape) > 1 else None
        ])

        self.generator = Sequential([layer for layer in self.generator.layers if layer is not None])

        self.discriminator = Sequential([
            Flatten(input_shape=self.input_shape) if len(self.input_shape) > 1 else
            Dense(512, input_shape=self.input_shape),
            LeakyReLU(alpha=0.01),
            Dropout(0.3),
            Dense(256),
            LeakyReLU(alpha=0.01),
            Dropout(0.3),
            Dense(1, activation='sigmoid')
        ])
```

```
        self._compile_models()

    def _compile_models(self):
        self.discriminator.compile(
            loss='binary_crossentropy',
            optimizer=Adam(learning_rate=0.0002, beta_1=0.5),
            metrics=['accuracy']
        )

        z = Input(shape=(self.z_dim,))
        generated_data = self.generator(z)
        self.discriminator.trainable = False
        validity = self.discriminator(generated_data)

        self.combined = Model(z, validity)
        self.combined.compile(
            loss='binary_crossentropy',
            optimizer=Adam(learning_rate=0.0002, beta_1=0.5)
        )

    def load_data(self, data=None):
        if data is not None:
            self.data = data
        elif self.data_path:
            self.data = self._load_from_path()
        else:
            (X_train, _), (_, _) = keras.datasets.mnist.load_data()
            X_train = X_train[:3000]
            X_train = X_train / 127.5 - 1.0
            X_train = np.expand_dims(X_train, axis=-1)
            self.data = X_train
```

```
    return self.data

def _load_from_path(self):
    if self.data_path.endswith('.npy'):
        return np.load(self.data_path)
    elif self.data_path.endswith('.csv'):
        return pd.read_csv(self.data_path).values
    else:
        images = []
        for img_file in os.listdir(self.data_path):
            if img_file.endswith(('.png', '.jpg', '.jpeg')):
                img = load_img(os.path.join(self.data_path, img_file),
                               target_size=self.input_shape[:2],
                               color_mode='grayscale' if self.input_shape[2] == 1 else 'rgb')
                img = img_to_array(img)
                img = img / 127.5 - 1.0
                images.append(img)
        return np.array(images)

def sample_outputs(self, iteration, num_samples=16):
    if self.data_type == 'image':
        self._sample_images(iteration, num_samples)
    elif self.data_type == 'text':
        self._sample_text(iteration, num_samples)
    else:
        self._sample_generic(iteration, num_samples)

def _sample_images(self, iteration, num_samples):
    noise = np.random.normal(0, 1, (num_samples, self.z_dim))
    gen_imgs = self.generator.predict(noise, verbose=0)
    gen_imgs = 0.5 * gen_imgs + 0.5
```

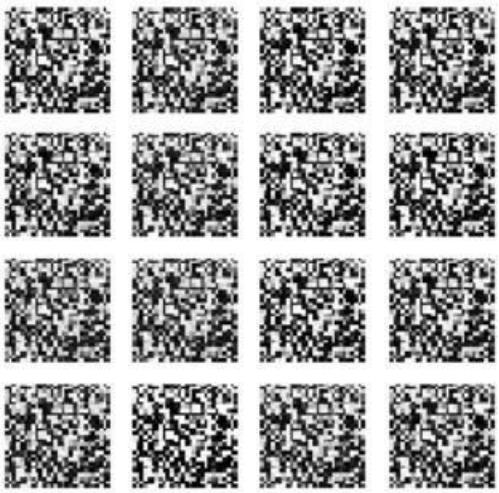
```
        rows = int(np.sqrt(num_samples))
        cols = int(np.sqrt(num_samples))

        fig, axs = plt.subplots(rows, cols, figsize=(4, 4))
        cnt = 0
        for i in range(rows):
            for j in range(cols):
                axs[i, j].imshow(gen_imgs[cnt, :, :, 0], cmap='gray')
                axs[i, j].axis('off')
                cnt += 1

        os.makedirs('outputs', exist_ok=True)
        plt.savefig(f'outputs/samples_{iteration}.png')
        plt.close()

        from IPython.display import Image, display
        display(Image(filename=f'outputs/samples_{iteration}.png'))

    def _sample_text(self, iteration, num_samples):
        print(f"Generated text samples at iteration {iteration}")
        print("Text generation would require specific vocabulary and decoding")

    def _sample_generic(self, iteration, num_samples):
        print(f"Generated {self.data_type} samples at iteration {iteration}")

    def train(self, iterations=20000, batch_size=128, sample_interval=1000):
        data = self.load_data()

        real = np.ones((batch_size, 1))
        fake = np.zeros((batch_size, 1))

        print("OUTPUT:")
        print()
```

```
        print(f"Generated samples at iteration 0:")
        self.sample_outputs(0)
        print()

        for iteration in range(iterations):
            idx = np.random.randint(0, data.shape[0], batch_size)
            real_data = data[idx]

            noise = np.random.normal(0, 1, (batch_size, self.z_dim))
            fake_data = self.generator.predict(noise, verbose=0)

            d_loss_real = self.discriminator.train_on_batch(real_data, real)
            d_loss_fake = self.discriminator.train_on_batch(fake_data, fake)
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

            noise = np.random.normal(0, 1, (batch_size, self.z_dim))
            g_loss = self.combined.train_on_batch(noise, real)

            if iteration % sample_interval == 0:
                print(f"{iteration} [D loss: {d_loss[0]:.6f}, acc.: {100*d_loss[1]:.2f}%] [G loss:
{g_loss:.6f}]")

        print()
        print(f"Generated samples at iteration {iterations-1}:")
        self.sample_outputs(iterations-1)

def run_image_gan():
    gan = UniversalGAN(data_type='image', input_shape=(28, 28, 1))
    gan.train(iterations=20000, batch_size=128, sample_interval=1000)

def run_text_gan():
    gan = UniversalGAN(data_type='text', input_shape=(100,))
```

```
    gan.train(iterations=10000, batch_size=64, sample_interval=1000)

def run_custom_gan(data_path, data_type='image', input_shape=None):
    gan = UniversalGAN(data_type=data_type, input_shape=input_shape, data_path=data_path)
    gan.train()

if __name__ == "__main__":
    run_image_gan()
```

OUTPUT:

```
0 [D loss: 0.002651, acc.: 100.00%] [G loss: 6.328685]
1000 [D loss: 0.020505, acc.: 100.00%] [G loss: 4.181127]
2000 [D loss: 0.077820, acc.: 96.88%] [G loss: 5.661728]
3000 [D loss: 0.197608, acc.: 92.97%] [G loss: 6.843472]
4000 [D loss: 0.136775, acc.: 94.92%] [G loss: 5.205476]
5000 [D loss: 0.461468, acc.: 78.52%] [G loss: 2.651729]
6000 [D loss: 0.119333, acc.: 95.31%] [G loss: 4.650574]
7000 [D loss: 0.349625, acc.: 84.77%] [G loss: 3.938936]
8000 [D loss: 0.478075, acc.: 80.47%] [G loss: 3.259602]
9000 [D loss: 0.324833, acc.: 86.33%] [G loss: 4.168482]
10000 [D loss: 0.217681, acc.: 91.80%] [G loss: 3.417312]
11000 [D loss: 0.411692, acc.: 83.20%] [G loss: 3.185366]
12000 [D loss: 0.282608, acc.: 87.11%] [G loss: 3.108163]
13000 [D loss: 0.283514, acc.: 86.72%] [G loss: 3.717927]
14000 [D loss: 0.341416, acc.: 84.77%] [G loss: 3.769761]
15000 [D loss: 0.382134, acc.: 81.64%] [G loss: 3.191629]
16000 [D loss: 0.329370, acc.: 85.94%] [G loss: 3.672557]
17000 [D loss: 0.251924, acc.: 88.67%] [G loss: 3.597138]
18000 [D loss: 0.247567, acc.: 90.62%] [G loss: 3.474238]
19000 [D loss: 0.336493, acc.: 84.38%] [G loss: 3.227752]
```



RESULT:

The result of running this code is the training of a GAN model to generate synthetic handwritten digit images similar to those in the MNIST dataset. The quality of generated images improves as training progresses, and you can visualize the generated images in the sample image grid displayed during training. The aim is to train the generator to produce images that are indistinguishable from real MNIST digits.