Experiment 6

Aim: Training a Sentiment Analysis model on IMDB dataset using RNN with LSTM notes

Importing The Libraries

```
import numpy as np
from keras.models import Sequential
from keras.preprocessing import sequence
from keras.layers import Dropout, Dense, Embedding, LSTM
from keras.datasets import imdb
from keras.callbacks import EarlyStopping
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import re
import nltk

nltk.download('stopwords')
nltk.download('wordnet')
```

Loading Datasets

```
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=10000)
word_index = imdb.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
```

Preprocessing Data

```
def preprocess_text(text):
text = re.sub(r'<[^>]+>', '', text)
text = re.sub(r'\d+', '', text)
text = re.sub(r'[^\w\s]', '', text)
```

```
text = text.lower()
stop_words = set(stopwords.words('english'))
words = text.split()
words = [word for word in words if word.lower() not in stop_words]
lemmatizer = WordNetLemmatizer()
words = [lemmatizer.lemmatize(word) for word in words]
return ' '.join(words)
x_train_text = [' '.join([reverse_word_index.get(i - 3, '?') for i in sequence]) for sequence in x_train]
x_test_text = [' '.join([reverse_word_index.get(i - 3, '?') for i in sequence]) for sequence in x_test]
x_train_text = [preprocess_text(text) for text in x_train_text]
x_test_text = [preprocess_text(text) for text in x_test_text]
maxlen= 200
tokenizer= Tokenizer(num_words=10000)
tokenizer.fit_on_texts(x_train_text)

x_train_seq = tokenizer.texts_to_sequences(x_train_text)
x_test_seq = tokenizer.texts_to_sequences(x_test_text)

x_train = pad_sequences(x_train_seq, maxlen=maxlen)
x_test = pad_sequences(x_test_seq, maxlen=maxlen)

y_train = np.array(y_train)
y_test = np.array(y_test)
```

Model Building and compiling

```
n_unique_words = 10000
model = Sequential()
model.add(Embedding(n_unique_words, 64, input_length=maxlen))
model.add(LSTM(32))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
history = model.fit(x_train, y_train, batch_size=128, epochs=10, validation_data=(x_test, y_test))
```

Laboratory Record of

NN & DL

Sheet No. 18

Experiment No. 06

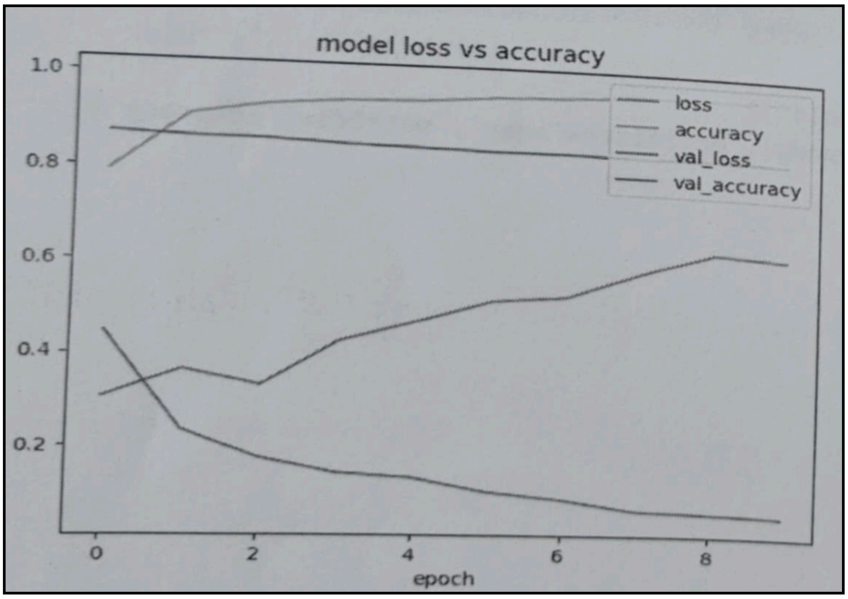
Date

```
Epoch 1/10
196/196 [==============================] - 41s 199ms/step - loss: 0.4446 - accuracy: 0.7917 - val_loss: 0.3063 - val_accuracy: 0.8720
Epoch 2/10
196/196 [==============================] - 42s 217ms/step - loss: 0.2366 - accuracy: 0.9128 - val_loss: 0.3656 - val_accuracy: 0.8656
Epoch 3/10
196/196 [==============================] - 41s 211ms/step - loss: 0.1772 - accuracy: 0.9376 - val_loss: 0.3333 - val_accuracy: 0.8641
Epoch 4/10
196/196 [==============================] - 42s 213ms/step - loss: 0.1454 - accuracy: 0.9500 - val_loss: 0.4308 - val_accuracy: 0.8571
Epoch 5/10
196/196 [==============================] - 41s 212ms/step - loss: 0.1360 - accuracy: 0.9538 - val_loss: 0.4748 - val_accuracy: 0.8534
Epoch 6/10
196/196 [==============================] - 46s 237ms/step - loss: 0.1061 - accuracy: 0.9650 - val_loss: 0.5224 - val_accuracy: 0.8493
Epoch 7/10
196/196 [==============================] - 42s 214ms/step - loss: 0.0913 - accuracy: 0.9701 - val_loss: 0.5358 - val_accuracy: 0.8493
Epoch 8/10
196/196 [==============================] - 42s 214ms/step - loss: 0.0672 - accuracy: 0.9791 - val_loss: 0.5928 - val_accuracy: 0.8453
Epoch 9/10
196/196 [==============================] - 41s 212ms/step - loss: 0.0632 - accuracy: 0.9800 - val_loss: 0.6418 - val_accuracy: 0.8490
Epoch 10/10
196/196 [==============================] - 41s 210ms/step - loss: 0.0586 - accuracy: 0.9820 - val_loss: 0.6359 - val_accuracy: 0.8462
```

```
from matplotlib import pyplot as plt

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Loss vs Accuracy')
plt.xlabel('Epoch')
plt.legend(['Loss', 'Accuracy', 'Val_Loss', 'Val_Accuracy'], loc='upper right')
plt.show()
sample_text = "This is a great movie with fantastic performances!"
sample_text = preprocess_text(sample_text)
tokenized_sample = tokenizer.texts_to_sequences([sample_text])
padded_sample = pad_sequences(tokenized_sample, maxlen=maxlen)
prediction = model.predict(padded_sample)
threshold = 0.5

if prediction[0][0] > threshold:
    print(f"The sample text is predicted as positive with confidence: {prediction[0][0]}")
else:
    print(f"The sample text is predicted as negative with confidence: {1 - prediction[0]
[0]}")
```



The sample text is predicted as positive with confidence: 0.86

Output: Trained a sentiment analysis model on IMDB dataset using RNN layers and LSTM notes and made predictions on sample text.

Experiment 7

Aim: Applying the Auto encoder algorithms for encoding the real-world data.

Importing The Libraries

```
from keras.layers import Dense, Conv2D, MaxPooling2D, UpSampling2D
from keras import Input, Model
from keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
```

Model Architecture

```
encoding_dim = 15
input_img = Input(shape=(784,))
encoded = Dense(encoding_dim, activation='relu')(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)
autoencoder = Model(input_img, decoded)
```

Encoder and Decoder Models

```
encoder = Model(input_img, encoded)
encoded_input = Input(shape=(encoding_dim,))
decoder_layer = autoencoder.layers[-1]
decoder = Model(encoded_input, decoder_layer(encoded_input))
```

Model Compilation

```
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

Data Preparation

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)
```

Output: (60000,784)
(10000,784)

Model Fitting

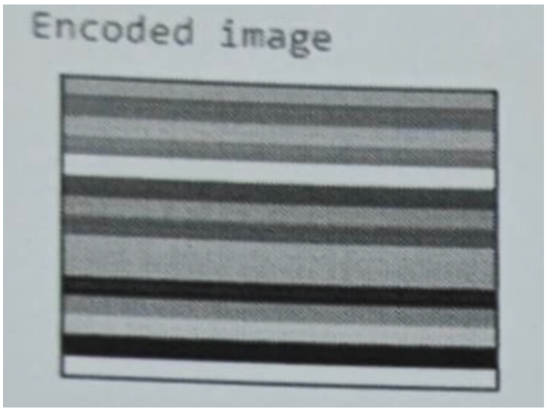
```
autoencoder.fit(
        x_train, x_train,
        epochs=15,
        batch_size=256,
        validation_data=(x_test, x_test)
)
```

```
Epoch 1/15
235/235 [==============================] - 5s 17ms/step - loss: 0.3093 - val_loss: 0.2156
Epoch 2/15
235/235 [==============================] - 4s 17ms/step - loss: 0.1965 - val_loss: 0.1812
Epoch 3/15
235/235 [==============================] - 4s 18ms/step - loss: 0.1736 - val_loss: 0.1641
Epoch 4/15
235/235 [==============================] - 3s 15ms/step - loss: 0.1595 - val_loss: 0.1534
Epoch 5/15
235/235 [==============================] - 4s 16ms/step - loss: 0.1514 - val_loss: 0.1477
Epoch 6/15
235/235 [==============================] - 4s 16ms/step - loss: 0.1472 - val_loss: 0.1445
Epoch 7/15
235/235 [==============================] - 3s 13ms/step - loss: 0.1446 - val_loss: 0.1423
Epoch 8/15
235/235 [==============================] - 2s 9ms/step - loss: 0.1427 - val_loss: 0.1407
Epoch 9/15
235/235 [==============================] - 2s 11ms/step - loss: 0.1410 - val_loss: 0.1390
Epoch 10/15
235/235 [==============================] - 2s 10ms/step - loss: 0.1395 - val_loss: 0.1374
Epoch 11/15
235/235 [==============================] - 4s 17ms/step - loss: 0.1381 - val_loss: 0.1363
Epoch 12/15
235/235 [==============================] - 4s 19ms/step - loss: 0.1369 - val_loss: 0.1350
Epoch 13/15
235/235 [==============================] - 4s 15ms/step - loss: 0.1358 - val_loss: 0.1340
Epoch 14/15
235/235 [==============================] - 4s 16ms/step - loss: 0.1350 - val_loss: 0.1332
Epoch 15/15
235/235 [==============================] - 5s 19ms/step - loss: 0.1344 - val_loss: 0.1326
<keras.src.callbacks.History at 0x7a3b6a093e80>
```

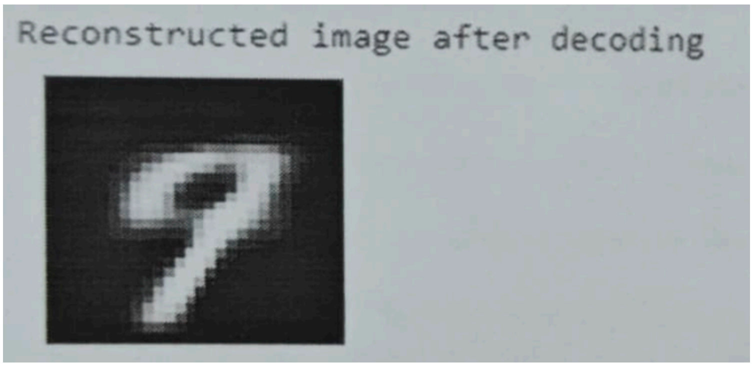
Evaluation and Visualization

```
plt.figure(figsize=(20, 6))
encoded_img = encoder.predict(x_test)
decoded_img = decoder.predict(encoded_img)
import random
i = random.randint(0, 10)
print("Original image")
ax = plt.subplot(3, 1, 1)
plt.imshow(x_test[i].reshape(28, 28))
plt.gray()
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
plt.show()
```


```
print("Encoded image")
encoded_image = encoded_img[i].reshape(encoding_dim, 1)
ax = plt.subplot(3, 1, 2)
plt.imshow(encoded_image, aspect=0.05)
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
plt.show()
```




```
print("Reconstructed image after decoding")
ax = plt.subplot(3, 1, 3)
plt.imshow(decoded_img[i].reshape(28, 28))
plt.gray()
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
plt.show()
```



Output: **Applied Auto encoder algorithm on MNIST dataset and displayed the original, encoded and decoded images**

Experiment 8

Aim: : Applying Generative Adversial Networks for image generation and Unsupervised Tasks

Importing The Libraries

```
import tensorflow as tf
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import numpy as np
import os
import time
from IPython import display
```

Loading Datasets

```
(train_images, train_labels), _ = tf.keras.datasets.mnist.load_data()
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5
```

Generator Model

```
BUFFER_SIZE = 10000
BATCH_SIZE = 128
train_dataset=tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7 * 7 * 256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Reshape((7, 7, 256)))
    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
```

```
        model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
use_bias=False))
        model.add(layers.BatchNormalization())
        model.add(layers.LeakyReLU())
        model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
use_bias=False, activation='tanh'))
        return model

    generator = make_generator_model()
```

Discriminator Model

```
    def make_discriminator_model():
        model = tf.keras.Sequential()
        model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[28, 28, 1]))
        model.add(layers.LeakyReLU())
        model.add(layers.Dropout(0.3))
        model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
        model.add(layers.LeakyReLU())
        model.add(layers.Dropout(0.3))
        model.add(layers.Flatten())
        model.add(layers.Dense(1))
        return model

    discriminator = make_discriminator_model()

    cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
    def discriminator_loss(real_output, fake_output):
        real_loss = cross_entropy(tf.ones_like(real_output), real_output)
        fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
        return real_loss + fake_loss
```

Loss Function

```
    def generator_loss(fake_output):
        return cross_entropy(tf.ones_like(fake_output), fake_output)
```

Optimizers

```
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                discriminator_optimizer=discriminator_optimizer,
                generator=generator,
                discriminator=discriminator)
```

Training the Model

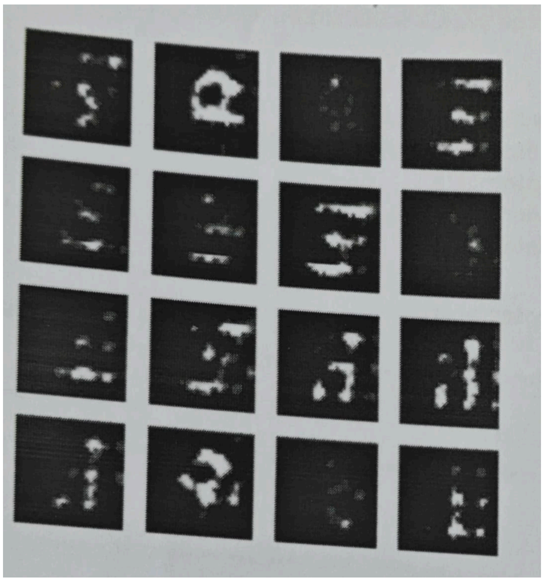
```
EPOCHS = 100
noise_dim = 100
num_examples_to_generate = 16
seed = tf.random.normal([num_examples_to_generate, noise_dim])
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)
        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)
        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)
    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))
```

```
def train(dataset, epochs):
    for epoch in range(epochs):
    start = time.time()
    for image_batch in dataset:
        train_step(image_batch)
    display.clear_output(wait=True)
    generate_and_save_images(generator, epoch + 1, seed)
    if (epoch + 1) % 15 == 0:
        checkpoint.save(file_prefix=checkpoint_prefix)
    print('Time for epoch {} is {} sec'.format(epoch + 1, time.time() - start))
display.clear_output(wait=True)
generate_and_save_images(generator, epochs, seed)
```

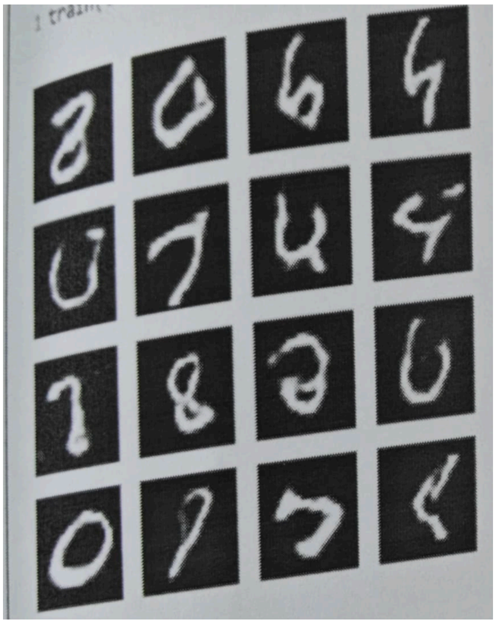
Generating and Saving Images

```
def generate_and_save_images(model, epoch, test_input):
    predictions = model(test_input, training=False)
    fig = plt.figure(figsize=(4, 4))
    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i + 1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
        plt.axis('off')
    plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
    plt.show()
    train(train_dataset, EPOCHS)
```

Output after 1st Epoch



Output after 100th Epoch



Output: Trained Generative Adversial Network on MNIST dataset for image generation.