Experiment No. 5
Design the architecture and implement the autoencoder model for Image Compression
Date of Performance:
Date of Submission:

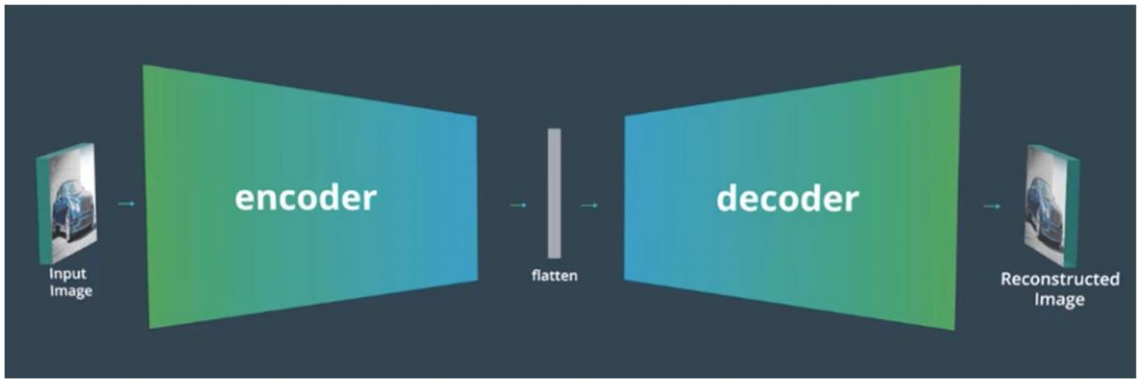
Aim: Design the architecture and implement the autoencoder model for Image Compression.

Objective: Ability to design autoencoder model to solve the given problem.

Theory:

Autoencoders are self-supervised machine learning models which are used to reduce the size of input data by recreating it. These models are trained as supervised machine learning models and during inference, they work as unsupervised models that's why they are called selfsupervised models. Autoencoder is made up of two components:

1. Encoder: It works as a compression unit that compresses the input data.
2. Decoder: It decompresses the compressed input by reconstructing it.



n an Autoencoder both Encoder and Decoder are made up of a combination of NN (Neural Networks) layers, which helps to reduce the size of the input image by recreating it. In the case of CNN Autoencoder, these layers are CNN layers (Convolutional, Max Pool, Flattening, etc.) while in the case of RNN/LSTM their respective layers are used.

Applications of Autoencoders

Three are several applications of Autoencoders some of the important ones are:

File Compression: Primary use of Autoencoders is that they can reduce the dimensionality of input data which we in common refer to as file compression. Autoencoders works with all

kinds of data like Images, Videos, and Audio, this helps in sharing and viewing data faster than we could do with its original file size.

Image De-noising: Autoencoders are also used as noise removal techniques (Image Denoising), what makes it the best choice for De-noising is that it does not require any human interaction, once trained on any kind of data it can reproduce that data with less noise than the original image.

Image Transformation: Autoencoders are also used for image transformations, which is typically classified under GAN(Generative Adversarial Networks) models. Using these we can transform B/W images to colored one and vice versa, we can up-sample and down-sample the input data, etc.

CODE:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = x_train.reshape((x_train.shape[0], 28 * 28))
x_test = x_test.reshape((x_test.shape[0], 28 * 28))
input_img = Input(shape=(28 * 28,))
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(32, activation='relu')(encoded)
```

```
decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(28 * 28, activation='sigmoid')(decoded)
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train,
                epochs=20,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test),
                verbose=1)
decoded_imgs = autoencoder.predict(x_test)
n = 10  # Number of digits to display
plt.figure(figsize=(20, 4))
for i in range(n):
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

OUTPUT:

Conclusion:

The autoencoder architecture, featuring dense layers with ReLU activations, effectively compresses input images into a lower-dimensional bottleneck layer before reconstructing them. The use of a sigmoid activation function in the final layer ensures that reconstructed images match the normalized input range. Training for more epochs typically enhances image reconstruction quality by allowing the model to learn more complex patterns and details. With too few epochs, the model may underfit, resulting in blurry or incomplete images, while excessive training can lead to overfitting. Optimal training strikes a balance, minimizing loss and preserving essential details, resulting in clear and accurate reconstructions.