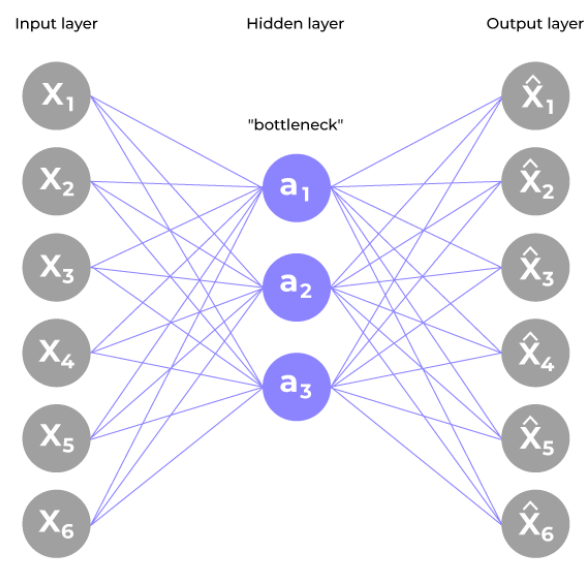
Experiment No. 6	
BE (AI&DS)	ROLL NO : 9742
Date of Implementation: 06/09/2024	
Aim: Implement auto encoder for image compression	
Programming Language Used : Python	
Upon completion of this experiment, students will be able to LO3: Build and train deep learning models for given problem	

Indicator		
Timeline Maintains submission deadline (1)	On time (1)	Otherwise (0)
Completion and Organization (2)	Completed in LAB (2)	Otherwise(1)
Analysis of output and conclusion(2)	Properly done (2)	Otherwise (0)
Viva (10)		

Assessment Marks :

Timeline(1)	
Completion and Organization (2)	
Analysis of output and conclusion(2)	
Viva (10)	
Total (15)	

EXPERIMENT	***6***
Aim	To Implement Auto encoder for Image Compression
Tools	PYTHON
Theory	Traditional feedforward **neural networks** can be great at performing tasks such as classification and regression, but what if we would like to implement solutions such as signal denoising or anomaly detection? One way to do this is by using Autoencoders. Autoencoders are a specialized class of algorithms that can learn efficient representations of input data with no need for labels. It is a class of artificial neural networks designed for unsupervised learning. Learning to compress and effectively represent input data without specific labels is the essential principle of an automatic decoder. This is accomplished using a two-fold structure that consists of an encoder and a decoder. The encoder transforms the input data into a reduced-dimensional representation, which is often referred to as "latent space" or "encoding". From that representation, a decoder rebuilds the initial input. For the network to gain meaningful patterns in data, a process of encoding and decoding facilitates the definition of essential features. **Architecture of Autoencoder in Deep Learning** The general architecture of an autoencoder includes an encoder, decoder, and bottleneck layer.  1. Encoder • Input layer take raw input data • The hidden layers progressively reduce the dimensionality of the input, capturing important features and patterns. These layer compose the encoder.

	• The bottleneck layer (latent space) is the final hidden layer, where the dimensionality is significantly reduced. This layer represents the compressed encoding of the input data. 2. Decoder • The bottleneck layer takes the encoded representation and expands it back to the dimensionality of the original input. • The hidden layers progressively increase the dimensionality and aim to reconstruct the original input. • The output layer produces the reconstructed output, which ideally should be as close as possible to the input data. 3. The loss function used during training is typically a reconstruction loss, measuring the difference between the input and the reconstructed output. Common choices include mean squared error (MSE) for continuous data or binary cross-entropy for binary data. 4. During training, the autoencoder learns to minimize the reconstruction loss, forcing the network to capture the most important features of the input data in the bottleneck layer.
Implementation	1. Import necessary libraries 2. Load mnist digit data set 3. Define a basic autoencoder 4. Compile and fit Autoencoder 5. Visualize the original and reconstructed data
Conclusion	In this experiment, we implemented an Autoencoder for image compression using the MNIST dataset. The model learned to compress images by reducing dimensionality to a latent space and successfully reconstructed the original images. The training process showed decreasing loss values, indicating effective learning. Visualization confirmed the model's ability to reproduce the input data. Overall, this experiment highlighted the potential of Autoencoders as powerful tools for unsupervised learning and applications such as image denoising and anomaly detection.

Implementation:

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import mnist
```

[2]:
```
# Load MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()

# Normalize the data (Pixel values between 0 and 1)
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Reshape to fit into the model (flattened into 784-dimensional vectors)
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:])))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
11490434/11490434              0s
0us/step
```

[3]:
```
# Define the size of the encoded representation (latent space size)
encoding_dim = 32

# Input placeholder
input_img = Input(shape=(784,))

# Encoder: Encodes input into a smaller representation
encoded = Dense(encoding_dim, activation='relu')(input_img)

# Decoder: Decodes the smaller representation back to the original input size
decoded = Dense(784, activation='sigmoid')(encoded)

# Model that maps input to its reconstruction
autoencoder = Model(input_img, decoded)

# Separate Encoder Model
```

```python
encoder = Model(input_img, encoded)

# Create a decoder model (decoder layer)
encoded_input = Input(shape=(encoding_dim,))
decoder_layer = autoencoder.layers[-1]  # Last layer of the autoencoder
decoder = Model(encoded_input, decoder_layer(encoded_input))
```

[6]:
```python
# Compile the autoencoder
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Train the autoencoder and store the training history
history = autoencoder.fit(x_train, x_train,
                          epochs=50,
                          batch_size=256,
                          shuffle=True,
                          validation_data=(x_test, x_test))
```

```
Epoch 1/50
235/235              5s 17ms/step -
loss: 0.0926 - val_loss: 0.0915
Epoch 2/50
235/235              3s 11ms/step -
loss: 0.0925 - val_loss: 0.0915
Epoch 3/50
235/235              5s 13ms/step -
loss: 0.0925 - val_loss: 0.0915
Epoch 4/50
235/235              3s 13ms/step -
loss: 0.0925 - val_loss: 0.0915
Epoch 5/50
235/235              4s 10ms/step -
loss: 0.0925 - val_loss: 0.0914
Epoch 6/50
235/235              2s 9ms/step -
loss: 0.0925 - val_loss: 0.0915
Epoch 7/50
235/235              3s 9ms/step -
loss: 0.0925 - val_loss: 0.0914
Epoch 8/50
235/235              3s 11ms/step -
loss: 0.0923 - val_loss: 0.0915
Epoch 9/50
235/235              5s 9ms/step -
loss: 0.0925 - val_loss: 0.0914
Epoch 10/50
235/235              3s 9ms/step -
loss: 0.0925 - val_loss: 0.0914
```

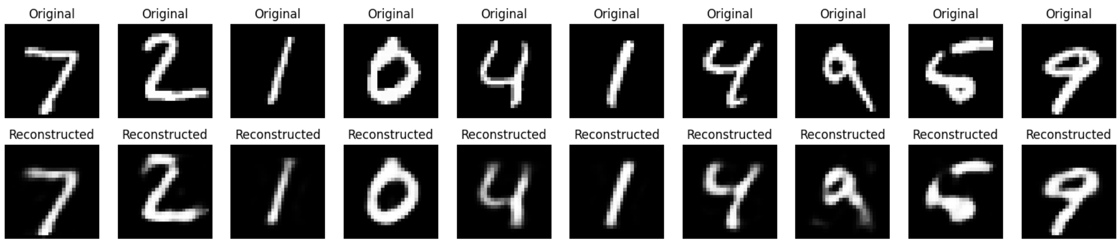
```
Epoch 11/50
235/235            2s 9ms/step -
loss: 0.0926 - val_loss: 0.0914
Epoch 12/50
235/235            3s 11ms/step -
loss: 0.0923 - val_loss: 0.0914
Epoch 13/50
235/235            5s 9ms/step -
loss: 0.0925 - val_loss: 0.0914
Epoch 14/50
235/235            2s 9ms/step -
loss: 0.0923 - val_loss: 0.0914
Epoch 15/50
235/235            3s 13ms/step -
loss: 0.0925 - val_loss: 0.0914
Epoch 16/50
235/235            4s 16ms/step -
loss: 0.0925 - val_loss: 0.0914
Epoch 17/50
235/235            4s 15ms/step -
loss: 0.0923 - val_loss: 0.0914
Epoch 18/50
235/235            2s 9ms/step -
loss: 0.0925 - val_loss: 0.0914
Epoch 19/50
235/235            2s 9ms/step -
loss: 0.0922 - val_loss: 0.0914
Epoch 20/50
235/235            3s 9ms/step -
loss: 0.0925 - val_loss: 0.0914
Epoch 21/50
235/235            4s 14ms/step -
loss: 0.0924 - val_loss: 0.0913
Epoch 22/50
235/235            4s 9ms/step -
loss: 0.0922 - val_loss: 0.0914
Epoch 23/50
235/235            2s 9ms/step -
loss: 0.0925 - val_loss: 0.0915
Epoch 24/50
235/235            2s 9ms/step -
loss: 0.0925 - val_loss: 0.0914
Epoch 25/50
235/235            3s 11ms/step -
loss: 0.0922 - val_loss: 0.0914
Epoch 26/50
235/235            5s 10ms/step -
loss: 0.0924 - val_loss: 0.0913
```

```
Epoch 27/50
235/235            2s 10ms/step -
loss: 0.0924 - val_loss: 0.0914
Epoch 28/50
235/235            3s 10ms/step -
loss: 0.0923 - val_loss: 0.0913
Epoch 29/50
235/235            3s 11ms/step -
loss: 0.0923 - val_loss: 0.0913
Epoch 30/50
235/235            5s 10ms/step -
loss: 0.0925 - val_loss: 0.0913
Epoch 31/50
235/235            2s 9ms/step -
loss: 0.0923 - val_loss: 0.0913
Epoch 32/50
235/235            2s 9ms/step -
loss: 0.0922 - val_loss: 0.0913
Epoch 33/50
235/235            3s 11ms/step -
loss: 0.0923 - val_loss: 0.0913
Epoch 34/50
235/235            3s 13ms/step -
loss: 0.0924 - val_loss: 0.0913
Epoch 35/50
235/235            4s 10ms/step -
loss: 0.0925 - val_loss: 0.0913
Epoch 36/50
235/235            2s 9ms/step -
loss: 0.0922 - val_loss: 0.0913
Epoch 37/50
235/235            3s 10ms/step -
loss: 0.0924 - val_loss: 0.0913
Epoch 38/50
235/235            5s 21ms/step -
loss: 0.0920 - val_loss: 0.0913
Epoch 39/50
235/235            2s 9ms/step -
loss: 0.0923 - val_loss: 0.0912
Epoch 40/50
235/235            2s 9ms/step -
loss: 0.0922 - val_loss: 0.0913
Epoch 41/50
235/235            3s 10ms/step -
loss: 0.0923 - val_loss: 0.0913
Epoch 42/50
235/235            2s 9ms/step -
loss: 0.0924 - val_loss: 0.0912
```

```
Epoch 43/50
235/235              4s 15ms/step -
loss: 0.0922 - val_loss: 0.0914
Epoch 44/50
235/235              4s 10ms/step -
loss: 0.0923 - val_loss: 0.0913
Epoch 45/50
235/235              2s 9ms/step -
loss: 0.0922 - val_loss: 0.0913
Epoch 46/50
235/235              3s 9ms/step -
loss: 0.0922 - val_loss: 0.0913
Epoch 47/50
235/235              4s 13ms/step -
loss: 0.0924 - val_loss: 0.0913
Epoch 48/50
235/235              3s 12ms/step -
loss: 0.0924 - val_loss: 0.0913
Epoch 49/50
235/235              5s 9ms/step -
loss: 0.0925 - val_loss: 0.0912
Epoch 50/50
235/235              2s 9ms/step -
loss: 0.0923 - val_loss: 0.0912
```

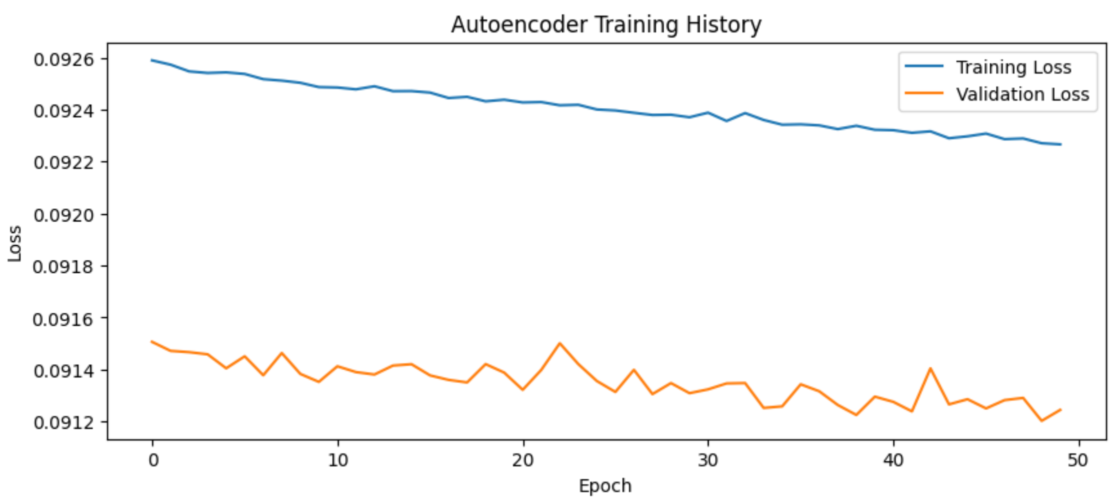
[7]:
```python
# Encode and decode some digits from the test set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)

# Visualize the original and reconstructed images
n = 10  # Number of digits to display
plt.figure(figsize=(20, 4))

for i in range(n):
    # Display original images
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    plt.title("Original")
    plt.axis('off')

    # Display reconstructed images
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
    plt.title("Reconstructed")
    plt.axis('off')

plt.show()
```

```
313/313                 1s 2ms/step
313/313                 0s 1ms/step
```



[8]:

```python
# Plot training history
plt.figure(figsize=(10, 4))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Autoencoder Training History')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



[10]:

```python
# Calculate and print compression ratio
original_size = x_test.nbytes
encoded_imgs = encoder.predict(x_test)  # Get the encoded images
compressed_size = encoded_imgs.nbytes
compression_ratio = original_size / compressed_size
print(f"Compression Ratio: {compression_ratio:.2f}")
```

313/313 0s 1ms/step

Compression Ratio: 24.50