6CS012 - Artificial Intelligence and Machine Learning. Representation Learning and Autoencoder.

Prepared By: Siman Giri {Module Leader - 6CS012}

March 30, 2025

———————— Tutorial - 7. ————————-

1 Instructions

This sheet contains exercises for various operations used in Autoencoder, accompanying the tutorial slides

- Please Complete all the exercise on Jupyter Notebook.

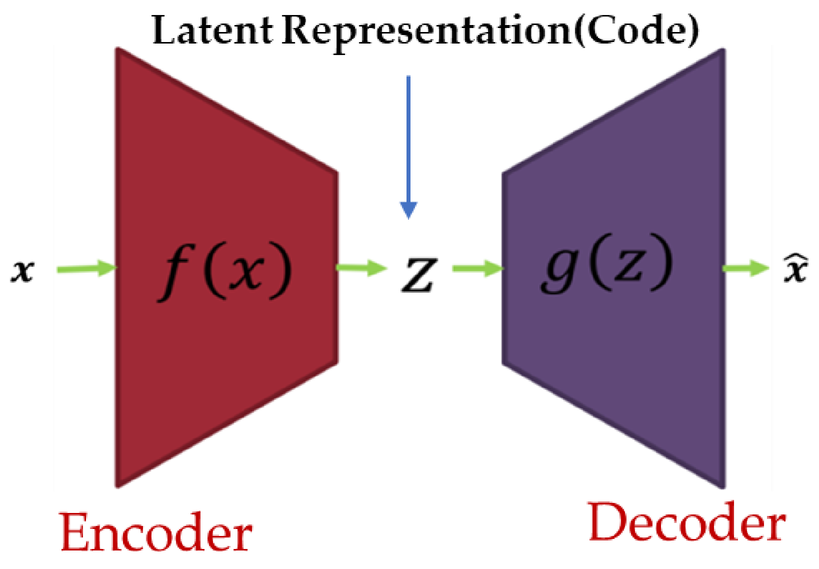


Figure 1: An example of Auto - Encoder.

2 A Recap on Autoencoder.

What is Autoencoder?

An autoencoder is a type of artificial neural network used to learn efficient representations of data, typically for the purpose of dimensionality reduction, feature extraction or Noise removal. Autoencoder are based on Encoder - Decoder style architecture.

Structure of an Autoencoder:

1. **Encoder:**
    - The encoder maps the input data **X** into a smaller representation, typically called the latent space **Z**.
    - The encoder learns to extract the most important features from the input data while ignoring the less important ones.

$$Z = f(X)$$

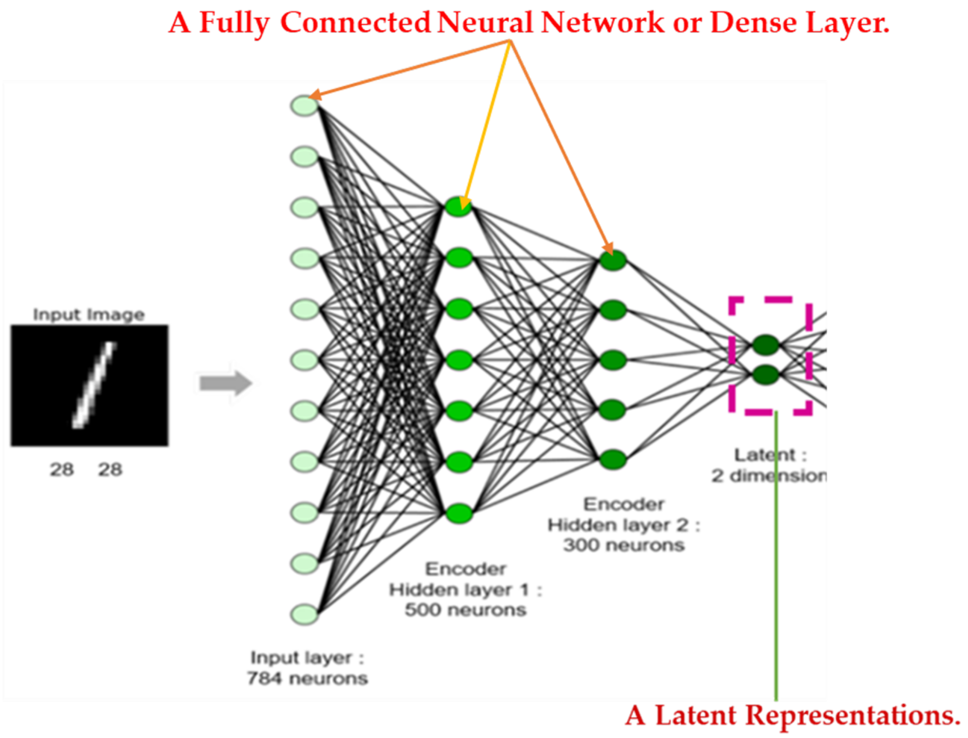

A Fully Connected Neural Network or Dense Layer.
Input Image
28 28
Input layer : 784 neurons
Encoder Hidden layer 1 : 500 neurons
Encoder Hidden layer 2 : 300 neurons
Latent : 2 dimension
A Latent Representations.

Figure 2: An Encoder built with Linear or Dense Layer of Fully Connected Network.

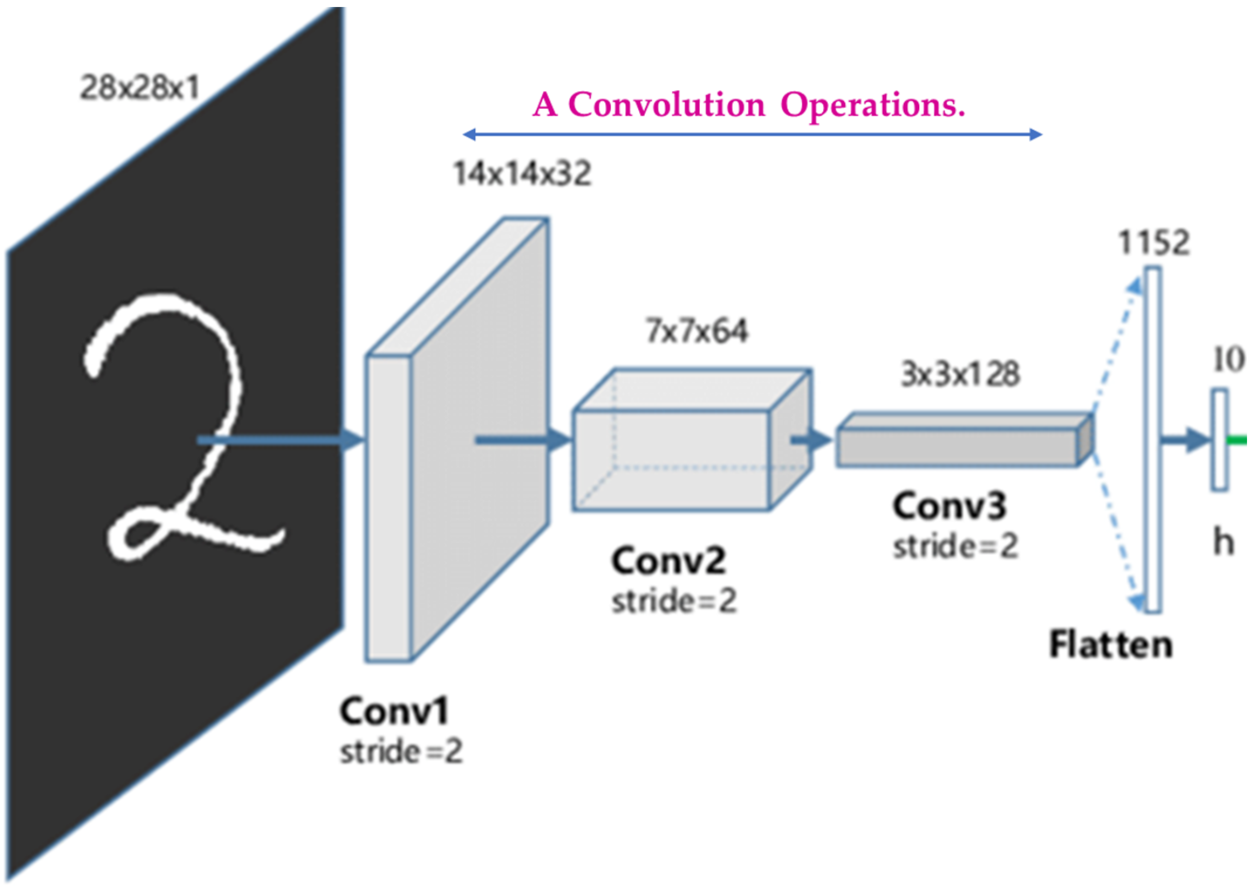


Figure 3: An Encoder built with Convolutional Layer.

2. **Decoder:**

- The decoder reconstructs the original input data from the encoded latent space representation.
- The decoder's output in autoencoder is typically the same shape as the original input data.

$$\mathbf{X}' = \mathbf{g}(\mathbf{Z})$$

$\mathbf{X}' \rightarrow$ is the reconstructed output.

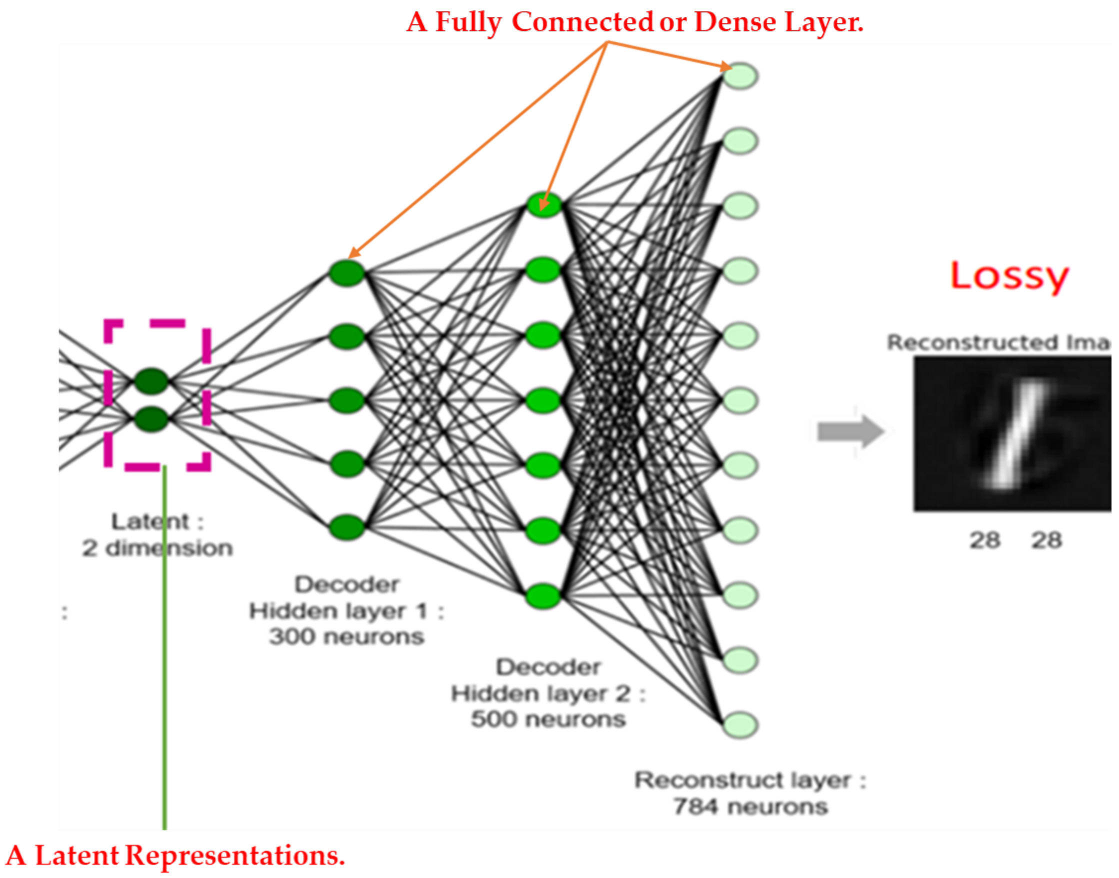

A Fully Connected or Dense Layer.
Lossy
Reconstructed Ima
28 28
Latent : 2 dimension
Decoder Hidden layer 1 : 300 neurons
Decoder Hidden layer 2 : 500 neurons
Reconstruct layer : 784 neurons
A Latent Representations.

Figure 4: A Decoder built with Linear or Dense Layer of Fully Connected Network.

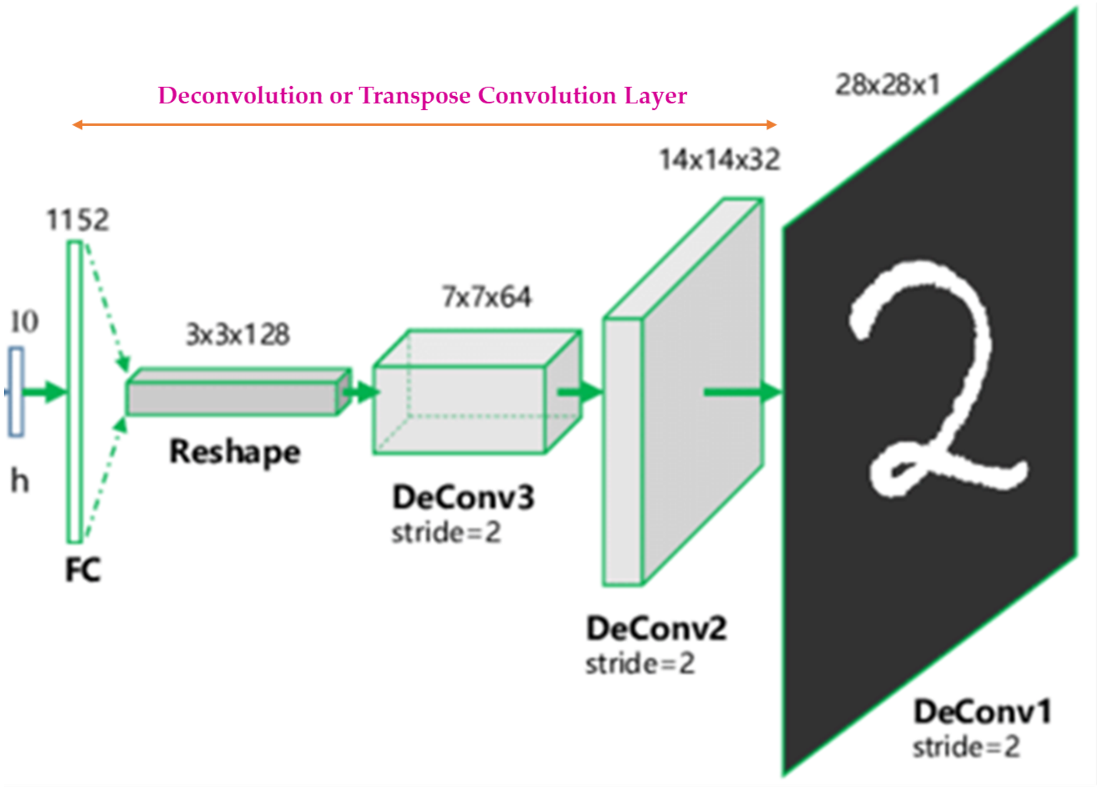


Figure 5: A decoder built with Transposed Convolution or De convolution Layer

How Does an Autoencoder Work?

1. **Training the Autoencoder:**
    - During training, the network learns to compress the input data into compact representation and then reconstruct it.
    - The model is trained to minimize the loss between the original input and the reconstructed output.
    - This means that the autoencoder tries to reproduce the original input as accurately as possible after encoding and decoding.
2. **Loss Function:** The most common loss function used is mean squared error or binary cross entropy.
    - **Mean Squared Error:** Measures the squared difference between the input and the output, i.e. how much the reconstructed image differs from the original.
    - **Binary Cross - Entropy:** Commonly used when working with images where the pixel values are normalized to the range: $[\mathbf{0}-\mathbf{1}]$.

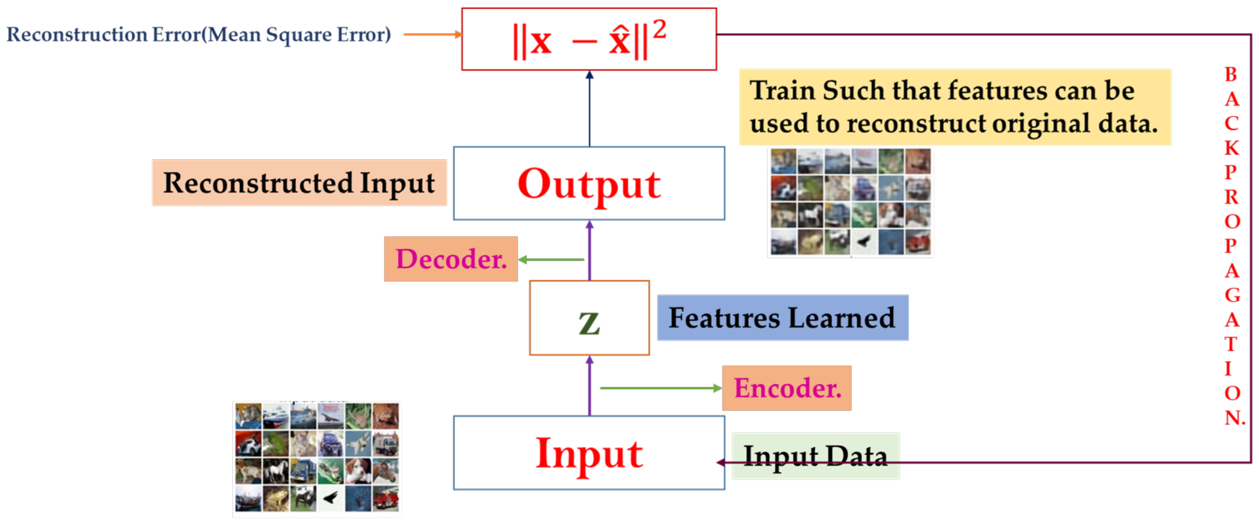

Reconstruction Error(Mean Square Error)
$||\mathbf{x} - \hat{\mathbf{x}}||^2$
Train Such that features can be used to reconstruct original data.
BACKPROPAGATION.
Reconstructed Input
Output
Decoder.
z
Features Learned
Encoder.
Input
Input Data

Figure 6: Training an Autoencoder.

3 Application of Autoencoder: Denoising.

In this section we will build and denoising autoencoders proposed by Y. Bengio et. Al. (Learning Deep Architectures 2009).As suggested by the original work we will work with image data, and a gaussian noise to it, and then train an convolutional autoencoder model to remove the noise. By the end of this section you should be able to:

- Build and train a denoising autoencoder.
- Understand the impact of noise on image data and how autoencoders help in denoising.
- Evaluate the model performance visually.

For the demonstration we will be using MNIST Handwritten Dataset.

Step 1: Load, pre - process and add a Noise to the Dataset:

1.1 Tasks to Do:

1. Load the MNIST dataset using keras.
2. Normalize the images to be in the range [0 - 1] {rescaling}.
3. Reshape the images to be $\mathbf{28} \times \mathbf{28} \times \mathbf{1}$.
4. Add a Gaussian noise to the images with noise factor of 0.5.
5. Clip the noisy images to ensure all pixel values are between 0 and 1.
6. Visualize the Input image and Noise added image.

Add Noise and Visualize Data.

```python
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
# Load and preprocess MNIST
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))
#Generate noisy versions
noise_factor_low = 0.1
noise_factor_high = 0.5
x_train_noisy_low = x_train + noise_factor_low * np.random.normal(loc=0.0, scale=1.0, size=x_train.
    shape)
x_train_noisy_high = x_train + noise_factor_high * np.random.normal(loc=0.0, scale=1.0, size=x_train
    .shape)
x_test_noisy_low = x_test + noise_factor_low * np.random.normal(loc=0.0, scale=1.0, size=x_test.
    shape)
x_test_noisy_high = x_test + noise_factor_high * np.random.normal(loc=0.0, scale=1.0, size=x_test.
    shape)
# Clip pixel values to [0, 1]
x_train_noisy_low = np.clip(x_train_noisy_low, 0., 1.)
```

```
x_test_noisy_low = np.clip(x_test_noisy_low, 0., 1.)
x_train_noisy_high = np.clip(x_train_noisy_high, 0., 1.)
x_test_noisy_high = np.clip(x_test_noisy_high, 0., 1.)
#Visualize in a 3 x 3 grid
n = 3 # number of images to show
plt.figure(figsize=(9, 9))
for i in range(n):
    # Original image
    ax = plt.subplot(n, 3, i * 3 + 1)
    plt.imshow(x_train[i].reshape(28, 28), cmap='gray')
    plt.title("Original")
    plt.axis('off')
    # Low noise image
    ax = plt.subplot(n, 3, i * 3 + 2)
    plt.imshow(x_train_noisy_low[i].reshape(28, 28), cmap='gray')
    plt.title("Noise 0.1")
    plt.axis('off')
    # High noise image
    ax = plt.subplot(n, 3, i * 3 + 3)
    plt.imshow(x_train_noisy_high[i].reshape(28, 28), cmap='gray')
    plt.title("Noise 0.5")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Noise Factor - Controls amount of noise we want to add.

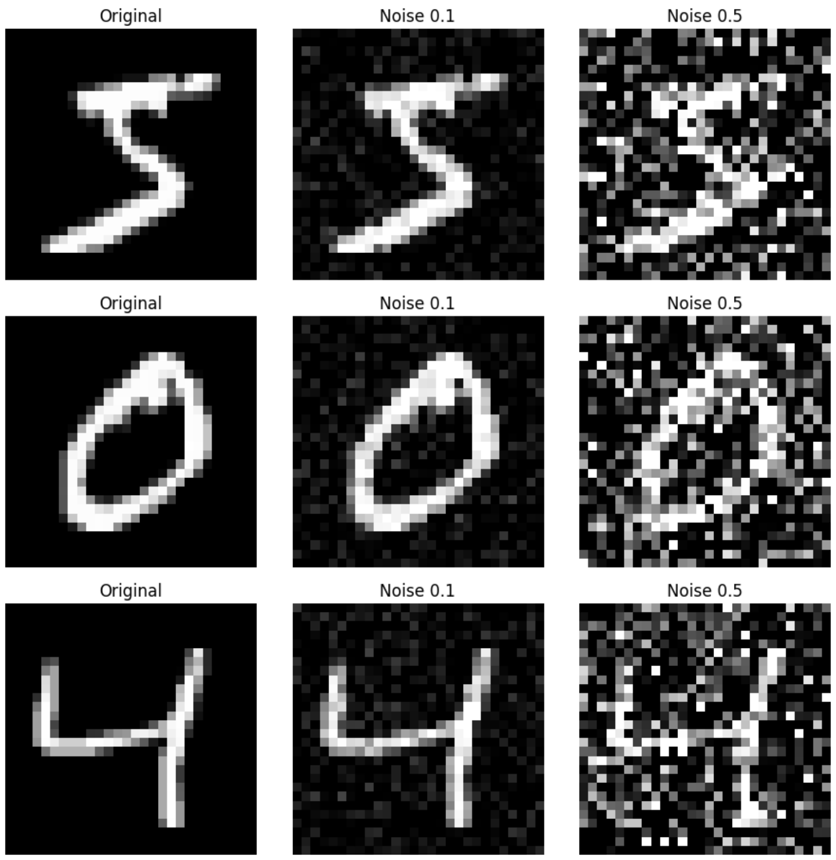


Figure 7: Loaded and Added Noise.

Step 2: Build the Convolutional Denoising Autoencoder Model:

2.1 Tasks to Do:

- **Encoder:**
  - Convolutional Layers + ReLU activation.
  - Maxpooling to down-sample the features.
- **Decoder:**
  - Convolutional layers + ReLU activation.
  - Upsampling to upsample the feature maps.
  - Final Layer with sigmoid activation to keep output pixels in range $[0, 1]$.
- **Transpose Convolutional is achieved in Keras using convolutional and Upsampling operations.**
- Compile the Model with `binary_crossentropy` loss and ADAM optimizer.

Sample Code Implementation with Keras:

We will be using functional approach, but free to use any architecture.

- **Encoder:**

Building Encoder.

```
def build_encoder(input_shape=(28, 28, 1)):
    """
    Builds the encoder part of the convolutional autoencoder.
    Parameters:
        input_shape (tuple): Shape of the input image. Default is (28, 28, 1) for MNIST.
    Returns:
        input_img (Keras Input): Input layer of the model.
        encoded (Keras Tensor): Encoded representation after convolution and pooling.
    """
    input_img = Input(shape=input_shape, name="input")
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
    x = MaxPooling2D((2, 2), padding='same')(x)
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    encoded = MaxPooling2D((2, 2), padding='same', name="encoded")(x)
    return input_img, encoded
```

- **Decoder:**

Building Decoder.

```
    def build_decoder(encoded_input):
        """
        Builds the decoder part of the convolutional autoencoder.

        Parameters:
            encoded_input (Keras Tensor): The output from the encoder.
        Returns:
            decoded (Keras Tensor): The reconstructed output after upsampling and
                convolution.
        """
        x = Conv2D(64, (3, 3), activation='relu', padding='same')(encoded_input)
        x = UpSampling2D((2, 2))(x)
        x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
        x = UpSampling2D((2, 2))(x)
        decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same', name="decoded")(x)
        return decoded
```

- **Complete Autoencoder:**

Building Autoencdoder.

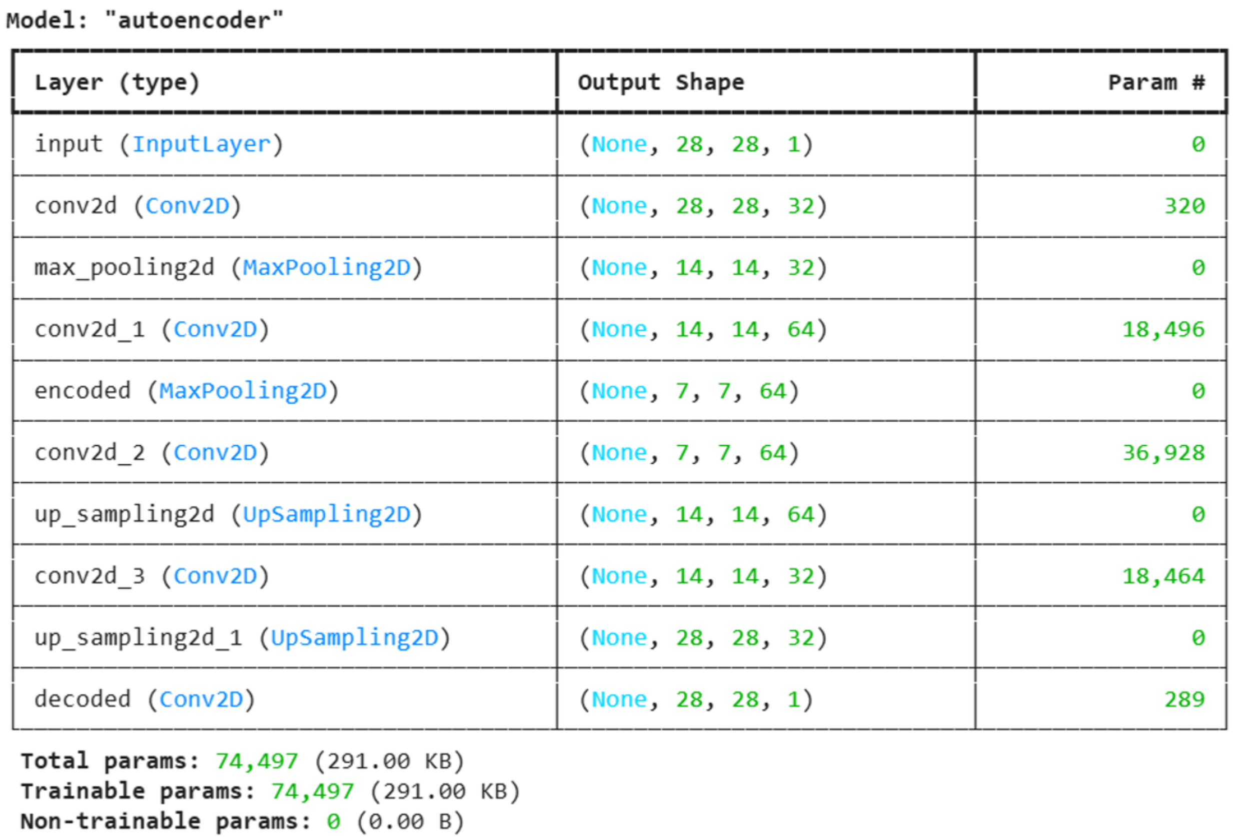
```
    def build_autoencoder():
    """
    Constructs and compiles the full convolutional autoencoder by connecting encoder and
        decoder.

    Returns:
        autoencoder (Keras Model): Compiled autoencoder model.
    """
    input_img, encoded_output = build_encoder()
    decoded_output = build_decoder(encoded_output)
    autoencoder = Model(inputs=input_img, outputs=decoded_output, name="autoencoder")
    autoencoder.compile(optimizer=Adam(), loss='binary_crossentropy')
    return autoencoder
```

- **Compile the Model:**

Compile the Model.

```
autoencoder = build_autoencoder()
autoencoder.summary()
```



```
Model: "autoencoder"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ input (InputLayer)              │ (None, 28, 28, 1)      │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d (Conv2D)                 │ (None, 28, 28, 32)     │           320 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d (MaxPooling2D)    │ (None, 14, 14, 32)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_1 (Conv2D)               │ (None, 14, 14, 64)     │        18,496 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ encoded (MaxPooling2D)          │ (None, 7, 7, 64)       │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_2 (Conv2D)               │ (None, 7, 7, 64)       │        36,928 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ up_sampling2d (UpSampling2D)    │ (None, 14, 14, 64)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_3 (Conv2D)               │ (None, 14, 14, 32)     │        18,464 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ up_sampling2d_1 (UpSampling2D)  │ (None, 28, 28, 32)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ decoded (Conv2D)                │ (None, 28, 28, 1)      │           289 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 74,497 (291.00 KB)
 Trainable params: 74,497 (291.00 KB)
 Non-trainable params: 0 (0.00 B)
```

Figure 8: Expected Output - The Model Summary.

Step 3: Train the Autoencoder:

3.1 Tasks to Do:

- Train the autoencoder using the noisy images as input and the clean images as the target.
- Monitor the training and validation loss during training.

Training Autoencoder

```
# Build the autoencoder from previous steps
autoencoder = build_autoencoder()
# Train with noisy input and clean target
history = autoencoder.fit(
    x_train_noisy_high, x_train,
    epochs=10,
    batch_size=128,
    shuffle=True,
    validation_data=(x_test_noisy_high, x_test)
)
```

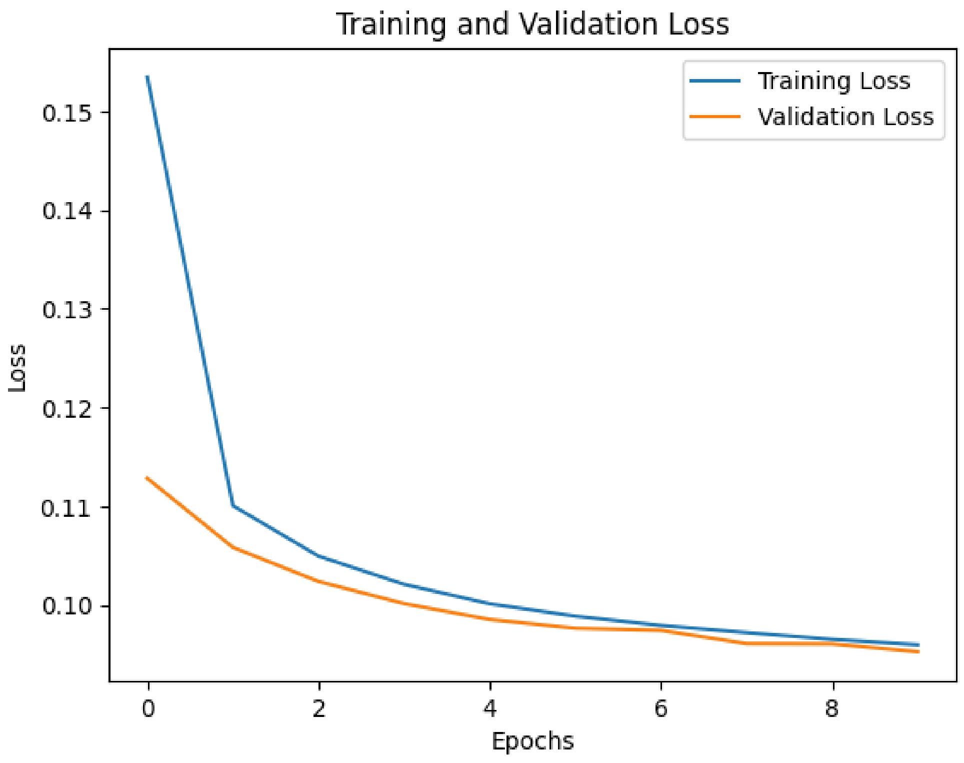


Figure 9: Model Behavior During Training.

Step 4: Evaluate the Autoencoder:

4.1 Tasks to Do:

- Use the trained autoencoder to predict the denoised images from the noisy test images.
- Display the original noisy images, the denoised images, and the clean images.

Evaluate Autoencoder

```
# Predict the denoised images from noisy test images
denoised_images = autoencoder.predict(x_test_noisy)
# Function to display images
def plot_images(noisy_images, denoised_images, clean_images, n=10):
    plt.figure(figsize=(20, 6))
    for i in range(n):
        # Plot noisy images
        plt.subplot(3, n, i + 1)
        plt.imshow(noisy_images[i].reshape(28, 28), cmap='gray')
        plt.title("Noisy Image")
        plt.axis('off')
        # Plot denoised images
        plt.subplot(3, n, i + 1 + n)
        plt.imshow(denoised_images[i].reshape(28, 28), cmap='gray')
        plt.title("Denoised Image")
        plt.axis('off')
        # Plot clean images
        plt.subplot(3, n, i + 1 + 2 * n)
        plt.imshow(clean_images[i].reshape(28, 28), cmap='gray')
        plt.title("Clean Image")
        plt.axis('off')
    plt.show()
# Display the first 10 images
plot_images(x_test_noisy, denoised_images, x_test, n=10)
```

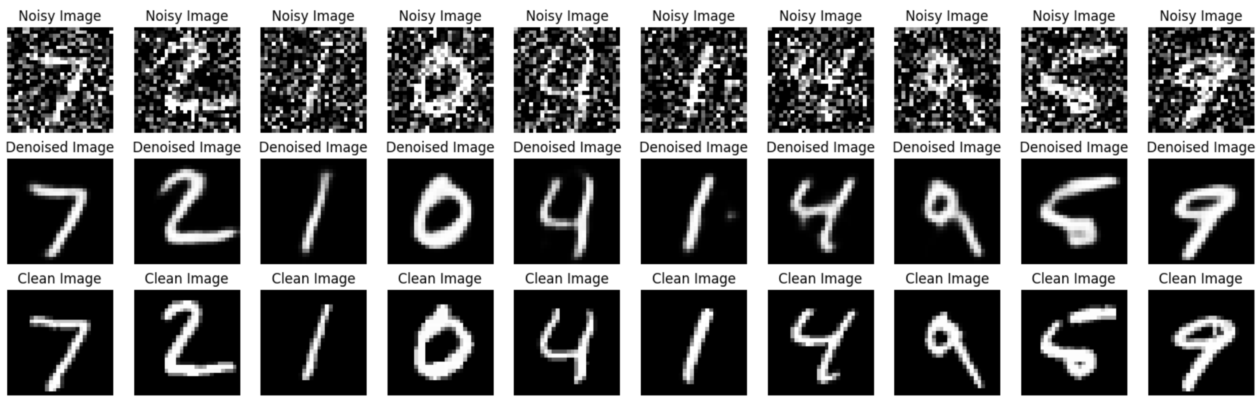


Figure 10: The - Final Output.

4 Exercise: Build a Denoising Convolutional Autoencoder

Dataset: Devnagari Handwritten Digit Dataset:

Objective:

The objective of this task is to build a **Denoising Convolutional Autoencoder** with at least **4 layers deep** to remove noise from images in the **Devnagari Handwritten dataset**. The task involves preprocessing the dataset, adding noise to the images, and training a deep convolutional autoencoder to clean them. This exercise will help you explore both convolutional neural networks and unsupervised learning techniques.
For the Devnagari Handwritten datasets from earlier week, Build a Denoising Convolutional Autoencoder at least 4 layer deep.

Also check `https://blog.keras.io/building-autoencoders-in-keras.html` for more help.

4.1 Instructions:

1. Dataset Preparation and Preprocessing:

- Load the Devnagari dataset using PIL.
- Normalize the images to the range [0, 1].
- Reshape the images to include a channel dimension for Keras Compatibility.
- Split the dataset into training and validation sets.
- Add noise to the images for denoising purposes (Gaussian noise or salt-and-pepper noise).

2. Build the Denoising Convolutional Autoencoder:

Build a convolutional autoencoder with at **least 4 layers**. The model should have an encoder to compress the input and a decoder to reconstruct the original image.

- Use convolutional layers for the encoder and decoder.
- Use ReLU activations for hidden layers and sigmoid for the output layer.

Expected Deliverables:

- A Complete rendered IPYTHON Notebook, with appropriate visualization and Results.

3. Train the Denoising Autoencoder:

Train the model with the noisy images as inputs and the original images as targets. Monitor the training process by plotting the loss curves.

4. Evaluate and Visualize the Results:

Evaluate the performance of the denoising autoencoder by visualizing a few noisy images, the denoised images generated by the model, and the original clean images.

5. Experiment and Fine - Tune the Model:

Try varying the model's architecture, noise levels, or training parameters. Experiment with different numbers of layers, filters, and noise factors. Record your observations about how these changes affect the model's performance.

Good Luck.