1. **Basics of Autoencoders:**

An artificial neural network design called an autoencoder is employed in unsupervised learning. In order to decode the input data back to its original form, they must first encode it into a lower-dimensional representation (encoder). As a result of being trained to reduce reconstruction error, the network is compelled to learn the key characteristics of the data.

1. **Role of Autoencoders in Unsupervised Learning:**

In unsupervised learning, autoencoders are essential because they can learn usable representations of the data without needing labeled examples. They can be used for many different things, including feature extraction, denoising, and data compression. When working with high-dimensional data, autoencoders are especially helpful since they may identify interesting patterns and structures.

1. **Types of Autoencoders:**

There are several types of autoencoders based on their architectures and purposes. Some common types include:

**Simple Autoencoder:** The basic autoencoder we discussed earlier, consisting of an encoder and decoder.

**Denoising Autoencoder:** Trained to remove noise from corrupted input data.

**Sparse Autoencoder:** Introduces sparsity constraints on the encoded representation.

**Variational Autoencoder (VAE):** Learns a probabilistic distribution over the encoded data, enabling the generation of new samples.

**Convolutional Autoencoder:** Utilizes convolutional layers for image data to capture spatial information.

**Stacked Autoencoder:** Consists of multiple layers of encoders and decoders to learn hierarchical representations.

1. **Use Case: Simple Autoencoder - Reconstructing the Existing Image:**

The basic autoencoder you gave is utilized to recreate the first picture by learning a packed portrayal. It catches the main elements of the info picture in the encoding system and afterward remakes it utilizing the unraveling layers.

1. **Use Case: Stacked Autoencoder:**

In order to learn hierarchical data representations, stacked autoencoders are utilized. They are made up of many layers of encoders and decoders, where each layer picks up on what came out of the layer before it to encode higher-level features. Stacked autoencoders have uses in image recognition, natural language processing, and other areas. They are useful for learning deep and complex data representations.

**IN CLASS PROGRAMMING:**

1. **Add One more hidden layer to the autoencoder:**

In this step, we modify the autoencoder architecture to include an additional hidden layer. This new layer will be placed after the original encoded layer and before the decoder part. The number of nodes in this new layer is chosen to be 64 with the ReLU activation function.

# Modified autoencoder architecture

encoded = Dense(encoding\_dim, activation='relu')(encoded)

# New hidden layer

new\_hidden\_layer = Dense(64, activation='relu')(encoded)

# ...

**Explanation:**

By adding this extra hidden layer, the autoencoder now has an additional layer to capture more complex patterns and features in the data.

1. **Do the prediction on the test data and then visualize one of the reconstructed versions of that test data. Also, visualize the same test data before reconstruction using Matplotlib:**

After training the autoencoder, we perform predictions on the test data and visualize one randomly selected reconstructed image along with its original image using Matplotlib.

# After training the autoencoder...

# Predict the test data

reconstructed\_images = autoencoder.predict(x\_test)

# Choose one random image index from the test data for visualization

image\_index = np.random.randint(0, len(x\_test))

# Original image

plt.imshow(x\_test[image\_index].reshape(28, 28), cmap='gray')

plt.title("Original Image")

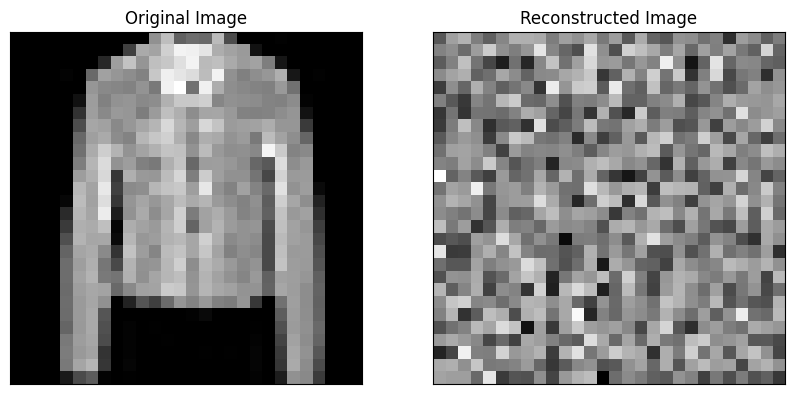
plt.show()

# Reconstructed image

plt.imshow(reconstructed\_images[image\_index].reshape(28, 28), cmap='gray')

plt.title("Reconstructed Image")

plt.show()



**Explanation:**

We use the trained autoencoder to predict the reconstructed images from the test data. Then, we randomly choose one test image, display the original image using plt.imshow(), and show the corresponding reconstructed image using the same function.

1. **Repeat the question 2 on the denoising autoencoder:**

For the denoising autoencoder, we follow the same procedure as in question 2. After training the denoising autoencoder, we perform predictions on the noisy test data and visualize one randomly selected denoised image along with its original noisy image using Matplotlib.

# After training the denoising autoencoder...

# Predict the test data after denoising

denoised\_images = autoencoder.predict(x\_test\_noisy)

# Choose one random image index from the test data for visualization

image\_index = np.random.randint(0, len(x\_test\_noisy))

# Original noisy image

plt.imshow(x\_test\_noisy[image\_index].reshape(28, 28), cmap='gray')

plt.title("Noisy Image")

plt.show()

# Denoised image

plt.imshow(denoised\_images[image\_index].reshape(28, 28), cmap='gray')

plt.title("Denoised Image")

plt.show()

A comparison of images of a rectangle

Description automatically generated

**Explanation:**

In this step, we used the trained denoising autoencoder to predict the denoised images from the noisy test data x\_test\_noisy. After predictions, we randomly selected one noisy test image with the index image\_index and displayed the original noisy image using plt.imshow(), enabling us to observe the Fashion MNIST image before denoising. We also displayed the corresponding denoised image using the same function, allowing us to visualize the image after the denoising process using the denoising autoencoder.

1. **Plot loss and accuracy using the history object:**

We monitor the loss during the training process and plot the training and validation losses to visualize how the autoencoder's performance changes over epochs.

import matplotlib.pyplot as plt

# Train the autoencoder

history = autoencoder.fit(x\_train\_noisy, x\_train,

epochs=10,

batch\_size=256,

shuffle=True,

validation\_data=(x\_test\_noisy, x\_test))

# Plot the loss

plt.plot(history.history['loss'], label='train')

plt.plot(history.history['val\_loss'], label='test')

plt.title('Model Loss')

plt.ylabel('Loss')

plt.xlabel('Epoch')

plt.legend()

plt.show()

# Plot the accuracy

plt.plot(history.history['accuracy'], label='train')

plt.plot(history.history['val\_accuracy'], label='test')

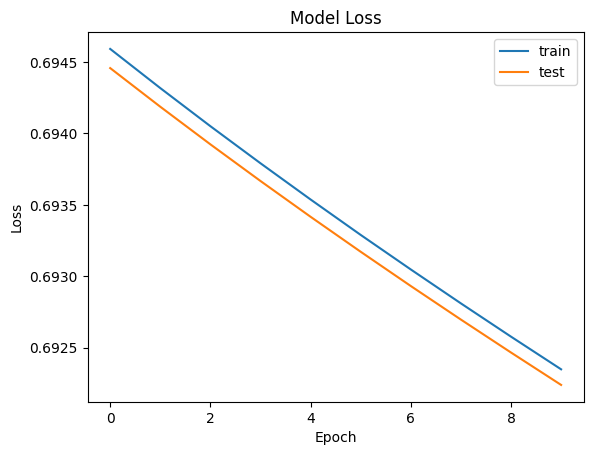
plt.title('Model Accuracy')

plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.legend()

plt.show()



A graph with a line and numbers

Description automatically generated

**Explanation**:

We keep track of training and validation losses in the history object. The loss values for each epoch are then accessed using history['loss'] and history['val\_loss']. Using Matplotlib, we plot these numbers to see how the autoencoder's loss evolves over the training process, which can reveal information about the model's effectiveness and propensity for overfitting.