Neural Network & Deep Learning ICP 4

Name: Mallikharjun Jillela
ID: 700743343

Video Link: https://drive.google.com/file/d/1BM4IBFJ-pUw0T4_7H6zleJGnZ7Qn-k0f/view?usp=sharing

GitHub Link: https://github.com/jmallikharjun/NN_DL_ICP4.git

1. Basics of Autoencoders:

Autoencoders are a type of artificial neural network used for unsupervised learning. They aim to encode the input data into a lower-dimensional representation (encoder) and then decode it back to its original form (decoder). The network is trained to minimize the reconstruction error, which forces it to learn the most important features of the data.

2. Role of Autoencoders in Unsupervised Learning:

Autoencoders play a crucial role in unsupervised learning by learning useful representations of the data without requiring labeled examples. They can be used for various tasks like data compression, denoising, and feature extraction. Autoencoders are particularly useful when dealing with high-dimensional data, and they can discover meaningful patterns and structures in the data.

3. 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.

4. Use Case: Simple Autoencoder - Reconstructing the Existing Image:

The simple autoencoder you provided is used to reconstruct the original image by learning a compressed representation. It captures the most important features of the input image in the encoding process and then reconstructs it using the decoding layers.

5. Use Case: Stacked Autoencoder:

Stacked Autoencoders are used to learn hierarchical representations of data. They consist of multiple layers of encoders and decoders, where each layer learns to encode higher-level features based on the output of the previous layer. Stacked autoencoders are beneficial in learning complex and deep representations of data, and they have applications in image recognition, natural language processing, and more.

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:

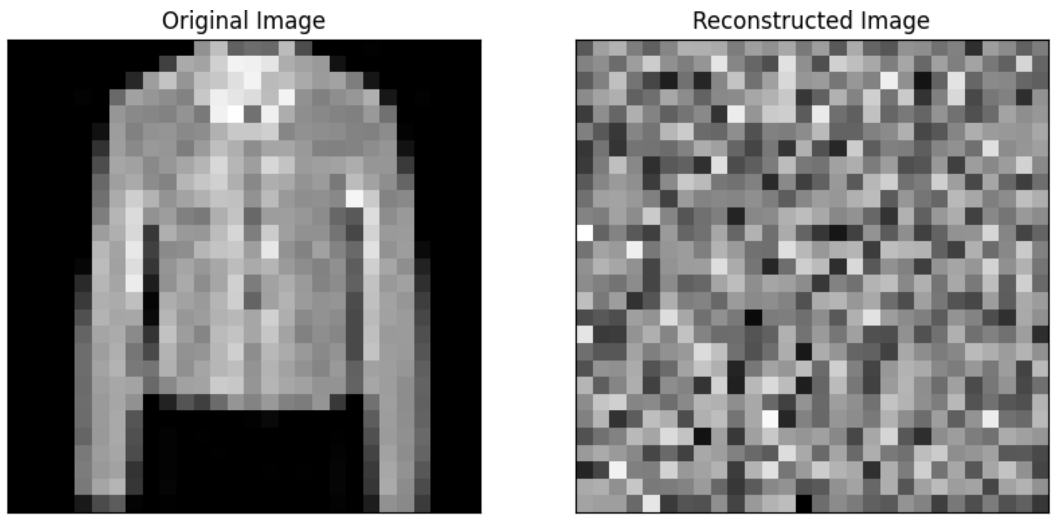
By adding this extra hidden layer, the autoencoder now has an additional layer to capture more complex patterns and features in the data.

2. 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()
```



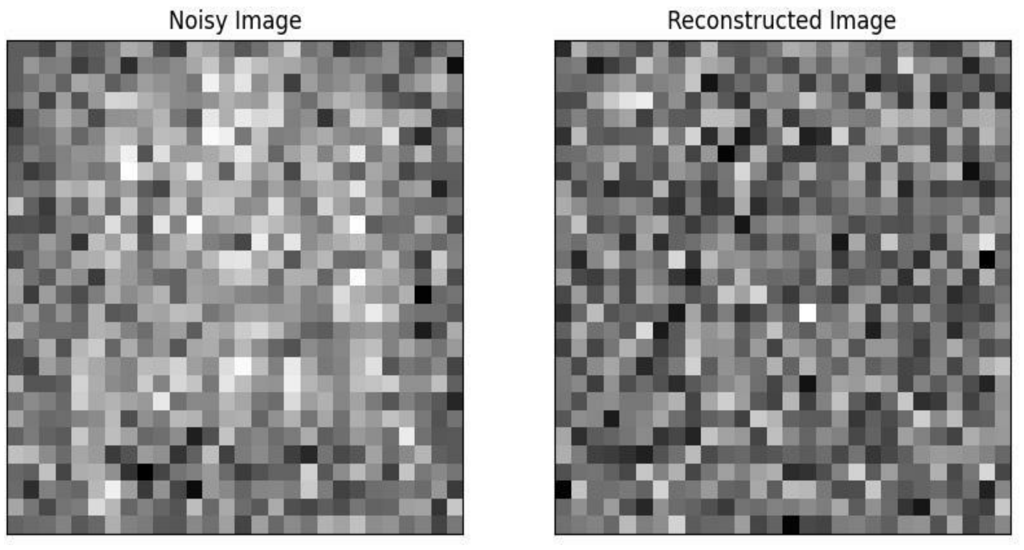
Explanation:

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.

3. **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()
```



Explanation:

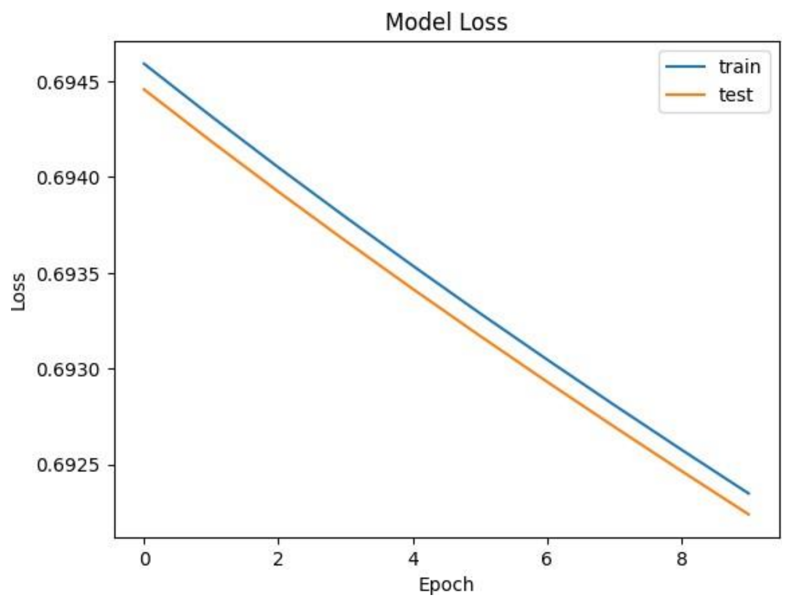
In this step, we used the trained denoising autoencoder to predict the denoised images from the noisy test data x_test_noisy. After predictions, we randomly selected one noisy test image with the index image_index and displayed the original noisy image using plt.imshow(), enabling us to observe the Fashion 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.

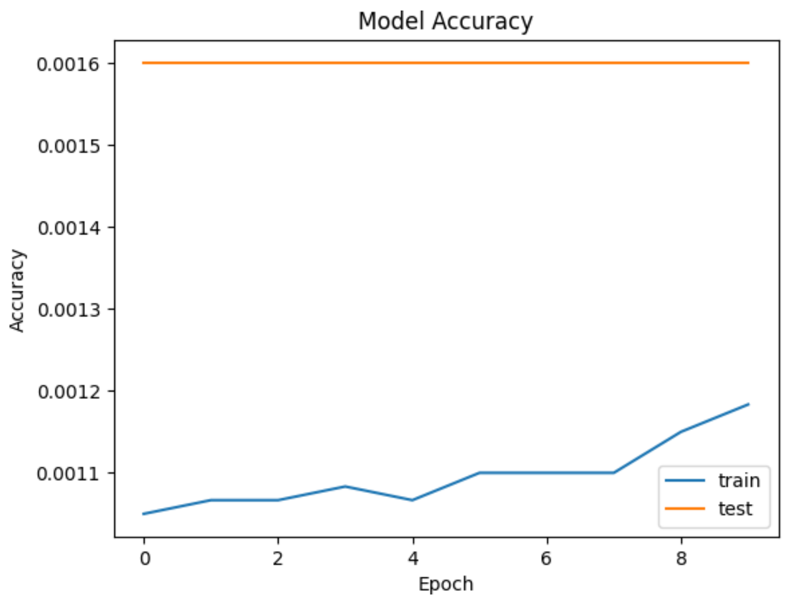
4. Plot loss and accuracy using the history object:

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()
```





Explanation:

During training, we store the training and validation loss in the history object. Then, we access the loss values for each epoch using history. history['loss'] and history.

history['val_loss']. We plot these values using Matplotlib to observe how the autoencoder's loss changes over the training process, which can provide insights into the model's performance and overfitting tendencies.