ATHARVA PATIL

ASSGN-4(DL)

****

IMPORTANT POINTS

1. **Data Loading and Preprocessing:**
   - dimensional representation with 32 features using a Dense layer with a ReLU activation function.
   - The decoder takes the encoded representation and reconstructs the original image with 784 features using a Dense layer with a sigmoid activation function.
2. **Model Compilation:**
   - The autoencoder model is compiled with the 'adam' optimizer and mean squared error (MSE) loss, which measures the reconstruction error.
3. **Model Training:**
   - The model is trained for 50 epochs with a batch size of 256. The training data is shuffled, and the validation data is used for monitoring the model's performance during training.
4. **Inference and Reconstruction:**
   - The encoder and decoder models are used to encode and decode images. Encoded and decoded images are generated for evaluation and visualization.
5. **Visualization of Original and Reconstructed Images:**
   - The code displays original and reconstructed images side by side for visual comparison.
6. **Anomaly Detection:**
   - Anomaly detection is performed by calculating the mean squared error (MSE) between the original and reconstructed images. An error threshold is set at 2 times the mean MSE, and anomalies are detected based on whether their MSE is greater than this threshold.
7. **Anomaly Detection Results:**
   - The code outputs the number of anomalies detected and visualizes the training and validation loss over the epochs.

****

DETAILED EXPLANATION

```
import numpy as np
from keras.layers import Input, Dense
from keras.models import Model
from keras.datasets import mnist
```

Here's a breakdown of each part:

1. **import numpy as np**: This imports the NumPy library, which is commonly used for numerical computations in Python. It's used to handle numerical operations and manipulate data.
2. **from keras.layers import Input, Dense**: This imports specific layers from the Keras library. The **Input** and **Dense** layers are fundamental building blocks for neural networks. **Input** is used to define the input layer, and **Dense** is used to create fully connected (dense) layers in the network.
3. **from keras.models import Model**: This imports the **Model** class from Keras. It's used to define and create neural network models by specifying the input and output layers.
4. **from keras.datasets import mnist**: This imports the MNIST dataset, a popular dataset for handwritten digit recognition. MNIST contains a large number of 28x28 grayscale images of digits from 0 to 9.

```
(x_train, _), (x_test, _) = mnist.load_data()
```

The code **mnist.load_data()** is used to load the MNIST dataset in Keras. The MNIST dataset consists of a large number of 28x28 grayscale images of handwritten digits from 0 to 9, along with corresponding labels. This code loads the dataset and splits it into training and test sets.

Here's a breakdown of what each part of the code does:

1. **(x_train, _), (x_test, _) = mnist.load_data()**: This line loads the MNIST dataset and splits it into two sets: **x_train** and **x_test**. The _ on the left side of each tuple represents a variable that you don't intend to use. In this case, you are not using the labels (target values) for now, so you assign them to _ to indicate that you're not interested in them.
   - **x_train**: This variable contains the training images, which will be used to train your model.
   - **x_test**: This variable contains the test images, which will be used to evaluate the model's performance.

You now have the MNIST dataset loaded and ready for preprocessing and model training. Typically, you would preprocess the data, define your neural network model (e.g., an autoencoder), and then train the model on the training data. The test data is used to evaluate the model's performance after training.

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
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:])))
```

The provided code is responsible for preprocessing the MNIST dataset before using it for training an autoencoder. Let's break down what each line of code does:

1. **x_train = x_train.astype('float32') / 255.**: This line converts the data type of the elements in the **x_train** array to **float32**. It's important to work with floating-point numbers, as neural networks typically perform better with these data types. Additionally, this line scales the pixel values in the range [0, 255] to the range [0.0, 1.0] by dividing each pixel value by 255.

2. **x_test = x_test.astype('float32') / 255.**: This line performs the same data type conversion and pixel value scaling for the test data in **x_test**.
3. **x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))**: This line reshapes the training data in **x_train**. The original shape of **x_train** is (number of samples, 28, 28), where each sample is a 28x28 pixel image. The **reshape** operation flattens each image into a 1D array by multiplying the dimensions, resulting in a shape of (number of samples, 784). This flattened format is suitable for feeding into the autoencoder.
4. **x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:]))**: Similarly, this line reshapes the test data in **x_test** to match the same flattened format as the training data.

After these preprocessing steps, **x_train** and **x_test** are in a format that can be used as input data for training and evaluating your autoencoder model. Each row in these arrays represents a flattened 28x28 pixel image with pixel values in the range [0.0, 1.0]. This is a common data preparation step when working with image data in neural networks.

```
input_img = Input(shape=(784,))
encoded = Dense(32, activation='relu')(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)
```

The provided code defines the architecture of an autoencoder model using Keras. Let's break down what each line of code does:

1. **input_img = Input(shape=(784,))**: This line defines the input layer of the autoencoder model. The **Input** function creates a placeholder for the model's input data. In this case, it specifies that the input data should have a shape of (784,), which matches the flattened format of the MNIST images after preprocessing.
2. **encoded = Dense(32, activation='relu')(input_img)**: This line defines the encoding (or compression) part of the autoencoder. It applies a dense (fully connected) layer with 32 units (neurons) and a ReLU (Rectified Linear Unit) activation function to the input data. This layer reduces the dimensionality of the data and captures important features.
3. **decoded = Dense(784, activation='sigmoid')(encoded)**: This line defines the decoding (or reconstruction) part of the autoencoder. It applies another dense layer with 784 units (matching the original input dimension) and a sigmoid activation function to the encoded data. This layer aims to reconstruct the original input data from the compressed representation.

In summary, the code defines a simple autoencoder architecture with an input layer of 784 units (representing the flattened MNIST images), an encoding layer with 32 units, and a decoding layer with 784 units. The activation functions are used to introduce non-linearity in the model.

This architecture forms the foundation of your autoencoder, but you'll still need to compile the model, train it, and evaluate its performance. Autoencoders are commonly used for tasks like image denoising, feature extraction, and dimensionality reduction.

```
encoded_input = Input(shape=(32,))
encoder = Model(input_img, encoded)
```

The code you provided is used to build the encoder part of your autoencoder model and create a separate encoder model. Here's a breakdown of each part:

1. **encoded_input = Input(shape=(32,))**: This line defines an input layer for the encoder model. The **Input** function creates a placeholder for the encoder's input data, and it specifies that the input data should have a shape of (32,), matching the dimensionality of the encoded representation produced by the encoder part of the autoencoder.
2. **encoder = Model(input_img, encoded)**: This line creates an instance of the **Model** class to define the encoder model. The first argument, **input_img**, specifies the input to the model, which is the original input to the autoencoder (the flattened MNIST images with 784 features). The second argument, **encoded**, specifies the output of the encoder part of the autoencoder. This line essentially creates a model that maps the input data to the encoded representation.

The encoder model is a standalone model that can be used to encode data into the compressed representation produced by the autoencoder's encoder part. It takes the same input as the autoencoder and produces the encoded representation as output. This can be useful for various applications where you want to extract features or representations from your data.

The decoder part of the autoencoder can be similarly created as a separate model if needed. In this way, you can use different parts of the autoencoder independently for specific tasks.

```
autoencoder = Model(input_img, decoded)
```

The code **autoencoder = Model(input_img, decoded)** creates the full autoencoder model by specifying both the input and the output layers. Here's a breakdown of what this line of code does:

1. **autoencoder = Model(input_img, decoded)**: This line defines the **autoencoder** model using the **Model** class from Keras. The first argument, **input_img**, specifies the input to the autoencoder model. This input corresponds to the original data that you want to encode and decode. The second argument, **decoded**, specifies the output of the autoencoder model, which represents the reconstructed data.

In other words, this line of code combines the encoder (which compresses the input data) and the decoder (which reconstructs the data) into a single autoencoder model. The autoencoder takes input data, encodes it into a compressed representation, and then decodes it to produce a reconstruction.

The **autoencoder** model can be used for training, evaluation, and other tasks related to autoencoder-based tasks, such as denoising images, feature extraction, or dimensionality reduction. It encompasses both the encoding and decoding steps within a single model.

```
decoder_layer = autoencoder.layers[-1]
decoder = Model(encoded_input,
decoder_layer(encoded_input))
```

The provided code is used to extract the decoder part of the autoencoder model as a separate model. Let's break down what each line does:

1. **decoder_layer = autoencoder.layers[-1]**: This line retrieves the last layer of the **autoencoder** model, which corresponds to the decoding part of the autoencoder. In the previous code, the autoencoder was defined to have an input layer, an encoding layer, and a decoding layer. By selecting the last layer, you're effectively isolating the decoding layer.
2. **decoder = Model(encoded_input, decoder_layer(encoded_input))**: This line creates a new model called **decoder** by specifying both the input and output layers. The **encoded_input** is used as the input to this model, representing the encoded data. The **decoder_layer(encoded_input)** connects the **decoder_layer** (which is the decoding layer from the autoencoder) to the **encoded_input**.

Essentially, the **decoder** model takes the encoded representation produced by the encoder part of the autoencoder and maps it to the reconstructed data using the decoding layer. This allows you to use the **decoder** model to generate reconstructions for encoded representations independently.

The **decoder** model is a useful tool when you want to perform tasks that involve only the decoding step, such as generating reconstructions from encoded data or using the decoding layer for feature visualization.

```
autoencoder.compile(optimizer='adam', loss='mse')
```

The provided code compiles the autoencoder model by specifying the optimizer and loss function. Here's a breakdown of what this line does:

1. **autoencoder.compile(optimizer='adam', loss='mse')**: This line compiles the **autoencoder** model using the following settings:
   - **optimizer='adam'**: This specifies the optimizer to be used during training. In this case, the 'adam' optimizer is chosen. Adam is a popular optimization algorithm that adapts the learning rate during training to optimize model parameters efficiently.
   - **loss='mse'**: This specifies the loss function used for training the autoencoder. Here, 'mse' stands for Mean Squared Error. Mean Squared Error is commonly used as a loss function in autoencoders for reconstruction tasks. It measures the squared difference between the original input and the reconstructed output, encouraging the autoencoder to minimize the reconstruction error.

After compiling the autoencoder, it's ready for training using the specified optimizer and loss function. Training the autoencoder involves feeding it input data, computing the loss, and updating the model's weights to minimize the loss, thereby improving the quality of the reconstruction.

```
history = autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

The provided code trains the autoencoder model using the MNIST dataset. Here's a breakdown of the parameters and what each line does:

1. **history = autoencoder.fit(x_train, x_train, epochs=50, batch_size=256, shuffle=True, validation_data=(x_test, x_test))**:
   - **x_train, x_train**: This specifies the training data for both input and target values. In the context of autoencoders, you train the model to reconstruct the same data it receives as input. Therefore, both the input and target data are set to **x_train**.
   - **epochs=50**: This parameter specifies the number of training epochs, which is the number of times the entire training dataset is passed forward and backward through the model. In this case, training will proceed for 50 epochs.
   - **batch_size=256**: This parameter determines the number of training examples used in each forward and backward pass. Training is done in batches, and each batch contains 256 samples. This can help speed up training and make better use of hardware resources.
   - **shuffle=True**: Setting **shuffle** to **True** means that the training data will be shuffled randomly before each epoch. Shuffling the data helps prevent the model from learning patterns related to the order of the data.
   - **validation_data=(x_test, x_test)**: This parameter specifies the validation dataset. During training, the model's performance on the validation data is evaluated at the end of each epoch. In this case, the validation data is set to **x_test**, which consists of images not used for training.

The **fit** method trains the autoencoder model by minimizing the mean squared error (MSE) loss between the input and the reconstructed output. The training process involves forward and backward passes, weight updates, and optimization using the Adam optimizer specified earlier. The training history, including loss and other metrics, is stored in the **history** object.

After training, you can use the trained autoencoder for various tasks, such as generating reconstructions or extracting features from data. The **history** object can also be used to visualize the training progress and performance metrics.

```
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
```

The code you provided is used for inference with the encoder and decoder models after training the autoencoder. Let's break down what each line does:

1. **encoded_imgs = encoder.predict(x_test)**: This line uses the **encoder** model to predict encoded representations for the data in **x_test**. In other words, it takes the test data and encodes it into a lower-dimensional representation. The result is stored in the **encoded_imgs** variable.
2. **decoded_imgs = decoder.predict(encoded_imgs)**: After encoding the test data, this line uses the **decoder** model to decode the encoded representations back into reconstructed data. The **encoded_imgs** from the previous step is used as input to the **decoder**, and the result is stored in the **decoded_imgs** variable.

Essentially, you are using the trained encoder to transform the test data into a lower-dimensional representation, and then using the decoder to reconstruct the data from that lower-dimensional representation. This can be helpful for various tasks, such as visualizing encoded features, generating reconstructions, or applying dimensionality reduction to the data.

The **encoded_imgs** and **decoded_imgs** variables now contain the results of the encoding and decoding steps, which you can use for further analysis or visualization.

```
import matplotlib.pyplot as plt
n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    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)
    # Display reconstruction
    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()
```

The provided code is used to display a comparison of the original MNIST digits and their corresponding reconstructions generated by the trained autoencoder. Here's what each part of the code does:

1. **n = 10**: This variable specifies how many digits will be displayed in the comparison. In this case, you want to display 10 digits.
2. **plt.figure(figsize=(20, 4))**: This line creates a Matplotlib figure with a specified size for displaying the original and reconstructed images. The **figsize** argument determines the width and height of the figure in inches.
3. The following loop runs 10 times, as specified by **n**, and for each iteration:

a. **ax = plt.subplot(2, n, i + 1)**: This line creates a subplot within the figure. The **2** in the first argument indicates that you want to arrange the subplots in two rows. The **n** in the second argument specifies the total number of subplots, and **i + 1** determines the current subplot position.
b. **plt.imshow(x_test[i].reshape(28, 28))**: This line displays the original MNIST digit from the test data. The **x_test[i]** represents the i-th test image, and **.reshape(28, 28)** reshapes it from the flattened 1D format to the original 28x28 format.
c. **plt.gray()**: This line sets the color map to grayscale, ensuring that the displayed images are in grayscale.
d. **ax.get_xaxis().set_visible(False)** and **ax.get_yaxis().set_visible(False)**: These lines hide the x-axis and y-axis labels for the subplot, making the images cleaner.
e. The next block of code, starting with **ax = plt.subplot(2, n, i + 1 + n)**, is similar to the first block but displays the reconstructed images from **decoded_imgs**.

4. **plt.show()**: Finally, this line displays the entire figure with the original and reconstructed images.

The code effectively visualizes the autoencoder's performance by showing the original images and their corresponding reconstructed versions, allowing you to assess how well the autoencoder has learned to represent and reconstruct the data.

```
mse = np.mean(np.power(x_test - decoded_imgs, 2), axis=1)
error_threshold = 2 * mse.mean()
```

The provided code calculates a reconstruction error (Mean Squared Error) for each example in the test set and then sets a threshold for identifying anomalies. Let's break down what each part of the code does:

1. **mse = np.mean(np.power(x_test - decoded_imgs, 2), axis=1)**: This line calculates the Mean Squared Error (MSE) for each example in the test set. Here's the breakdown:
   - **np.power(x_test - decoded_imgs, 2)**: This calculates the squared differences between the original test data (**x_test**) and the reconstructed data (**decoded_imgs**).
   - **np.mean(...)**: This computes the mean (average) of the squared differences for each example along **axis=1**. As a result, **mse** is a 1D NumPy array where each element corresponds to the MSE for a single example.
2. **error_threshold = 2 * mse.mean()**: This line sets a threshold for identifying anomalies. The threshold is defined as two times the mean of the MSE values in the **mse** array. Anomalies will be those examples with reconstruction errors (MSE) greater than this threshold.

After calculating the **mse** values and setting the **error_threshold**, you can use this threshold to detect anomalies or outliers in your data. Examples with reconstruction errors significantly higher than the threshold are more likely to be considered anomalies, as they deviate from the typical reconstruction error.

This approach can be useful for identifying data points that the autoencoder has difficulty reconstructing accurately, which may be indicative of anomalies or data instances that differ significantly from the learned data distribution.

```
# detect anomalies
anomalies = mse > error_threshold
print("Found %d anomalies!" % np.sum(anomalies))
*****
plt.figure(figsize=(12, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

The provided code detects anomalies based on the previously calculated mean squared error (MSE) values and then visualizes the training loss over epochs. Let's break down what each part of the code does:

1. **anomalies = mse > error_threshold**: This line creates a Boolean mask where **True** values indicate anomalies. Anomalies are identified as the examples in the test set for which the MSE is greater than the defined **error_threshold**. The result is a Boolean array where each element corresponds to whether the corresponding test example is considered an anomaly.

2. **print("Found %d anomalies!" % np.sum(anomalies))**: This line prints the number of anomalies found based on the **anomalies** Boolean mask. It counts the **True** values in the mask using **np.sum** to determine how many anomalies were detected.
3. **plt.figure(figsize=(12, 6))**: This line creates a Matplotlib figure for plotting the training loss.
4. **plt.plot(history.history['loss'], label='Training Loss')**: This line plots the training loss over epochs. The training loss is typically the reconstruction error (MSE) for each epoch during the training process.
5. **plt.plot(history.history['val_loss'], label='Validation Loss')**: This line plots the validation loss over epochs. The validation loss is also typically the reconstruction error (MSE) but calculated on a separate validation dataset.
6. **plt.xlabel('Epochs')**: This sets the label for the x-axis of the plot to indicate the number of training epochs.
7. **plt.ylabel('Loss')**: This sets the label for the y-axis of the plot to indicate the loss value.
8. **plt.legend()**: This displays a legend on the plot, differentiating between the training loss and validation loss.
9. **plt.show()**: This displays the loss plot, showing how the training and validation losses change over the course of training.

The code allows you to visually inspect the training loss and identify anomalies in the test data based on the reconstruction error threshold. The number of anomalies detected is printed, and you can assess the training process by examining the loss plot.