## 4. Autoencoder for Anomaly Detection

**Aim**: To implement an autoencoder for anomaly detection, learning to reconstruct "normal" data and identifying deviations as anomalies.

**Theory**: An **autoencoder** is a type of artificial neural network used for unsupervised learning, particularly for dimensionality reduction and feature learning. It consists of two main parts:

- **Encoder**: This part compresses the input data into a lower-dimensional latent representation (also known as the code or bottleneck).
- **Decoder**: This part takes the latent representation and reconstructs the original input data.

The autoencoder is trained by minimizing the **reconstruction error** between the original input and its reconstructed output. The core idea for **anomaly detection** is to train the autoencoder exclusively on "normal" data. When presented with anomalous data, the autoencoder will struggle to reconstruct it accurately, resulting in a high reconstruction error. This high error can then be used as a threshold to flag anomalies.

**Steps and Explanation**:

- **a. Import Required Libraries**: Load TensorFlow, Keras, NumPy, and Matplotlib.
- **b. Upload/Access the Dataset**: For this demonstration, we'll use a synthetic dataset or a common benchmark like the Fashion MNIST dataset, treating a subset as "normal" and the rest as potentially "anomalous." A dataset with clear patterns for normal data is ideal.
- **c. The Encoder Converts into Latent Representation**: Define the encoder network. This typically involves a series of Dense layers with decreasing numbers of neurons, progressively compressing the input. An activation function like ReLU is commonly used.
- **d. Decoder Networks Convert Back to Original Input**: Define the decoder network. It mirrors the encoder but with Dense layers that increase in neuron count, expanding the latent representation back to the original input dimensions. The final layer's activation should match the input data's range (e.g., sigmoid for normalized data between 0 and 1).
- **e. Compile the Models**:
    - Combine the encoder and decoder into a single Sequential model.
    - Compile the autoencoder using an appropriate **optimizer** (e.g., adam) and a **loss function** that measures the difference between input and output. **Mean Squared Error (MSE)** or **Binary Crossentropy** (if data is binary or normalized with sigmoid) are common choices.

   - **Evaluation Metrics**: While accuracy isn't directly applicable here, MSE itself serves as a key metric.

**Code Example (using Fashion MNIST, treating some classes as normal):**

Python

```
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt

# --- a. Import Required Libraries (Done above) ---

# --- b. Upload/Access the Dataset ---

# Load Fashion MNIST dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

# Normalize pixel values to be between 0 and 1
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Flatten the images
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:])))

print(f"Original training data shape: {x_train.shape}") # (60000, 784)
```

```
# Define "normal" classes (e.g., T-shirts/tops, Trousers, Pullovers, Dress, Coat)
# Note: Adjust these based on desired anomaly definition.
normal_classes = [0, 1, 2, 3, 4] # T-shirt/top, Trouser, Pullover, Dress, Coat
# normal_classes = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] # Use all classes for a basic autoencoder demo

# Filter training data to include only normal classes
idx_train_normal = np.isin(y_train, normal_classes)
x_train_normal = x_train[idx_train_normal]
y_train_normal = y_train[idx_train_normal] # Not used for training, but for reference

# Filter test data for normal and anomalous samples
idx_test_normal = np.isin(y_test, normal_classes)
x_test_normal = x_test[idx_test_normal]
y_test_normal = y_test[idx_test_normal]

idx_test_anomaly = ~idx_test_normal # "not normal"
x_test_anomaly = x_test[idx_test_anomaly]
y_test_anomaly = y_test[idx_test_anomaly]

print(f"Training data shape (normal only): {x_train_normal.shape}")
print(f"Test data shape (normal): {x_test_normal.shape}")
print(f"Test data shape (anomaly): {x_test_anomaly.shape}")

# --- c. The encoder converts it into a latent representation ---
input_dim = x_train_normal.shape[1] # 784
latent_dim = 32 # Size of the latent space

# Encoder
encoder_input = Input(shape=(input_dim,))
```

```
encoder_layer_1 = Dense(128, activation='relu')(encoder_input)

latent_representation = Dense(latent_dim, activation='relu')(encoder_layer_1)

encoder = Model(inputs=encoder_input, outputs=latent_representation, name="encoder")


# --- d. Decoder networks convert it back to the original input ---

# Decoder

decoder_input = Input(shape=(latent_dim,))

decoder_layer_1 = Dense(128, activation='relu')(decoder_input)

reconstructed_output = Dense(input_dim, activation='sigmoid')(decoder_layer_1) # Sigmoid
for [0,1] range

decoder = Model(inputs=decoder_input, outputs=reconstructed_output, name="decoder")


# Autoencoder = Encoder + Decoder

autoencoder_input = Input(shape=(input_dim,))

encoded_img = encoder(autoencoder_input)

decoded_img = decoder(encoded_img)

autoencoder = Model(inputs=autoencoder_input, outputs=decoded_img,
name="autoencoder")


autoencoder.summary()


# --- e. Compile the models with Optimizer, Loss, and Evaluation Metrics ---

autoencoder.compile(optimizer=Adam(learning_rate=0.001),

                    loss='mse') # Mean Squared Error for reconstruction loss


# Train the autoencoder ONLY on normal data

print("\nTraining the autoencoder on normal data...")

history = autoencoder.fit(x_train_normal, x_train_normal, # Input and output are the same

                          epochs=10, # Reduced epochs for demonstration

                          batch_size=256,
                     shuffle=True,
                     validation_data=(x_test_normal, x_test_normal)) # Validate on normal test
data

# Predict reconstructions for normal and anomalous test data
x_test_normal_pred = autoencoder.predict(x_test_normal)
x_test_anomaly_pred = autoencoder.predict(x_test_anomaly)

# Calculate reconstruction errors (MSE)
mse_normal = np.mean(np.power(x_test_normal - x_test_normal_pred, 2), axis=1)
mse_anomaly = np.mean(np.power(x_test_anomaly - x_test_anomaly_pred, 2), axis=1)

# Plotting reconstruction errors
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.hist(mse_normal, bins=50, alpha=0.7, label='Normal Data Reconstruction Error')
plt.title('Reconstruction Error on Normal Test Data')
plt.xlabel('Mean Squared Error')
plt.ylabel('Frequency')
plt.legend()

plt.subplot(1, 2, 2)
plt.hist(mse_anomaly, bins=50, alpha=0.7, label='Anomaly Data Reconstruction Error',
color='red')
plt.title('Reconstruction Error on Anomaly Test Data')
plt.xlabel('Mean Squared Error')
plt.ylabel('Frequency')
plt.legend()
plt.tight_layout()
plt.show()

# Determine a threshold (e.g., based on the 95th percentile of normal errors)
threshold = np.percentile(mse_normal, 95)
print(f"\nReconstruction threshold (95th percentile of normal errors): {threshold:.6f}")

# Classify test data based on the threshold
anomalies_detected_normal = mse_normal > threshold
anomalies_detected_anomaly = mse_anomaly > threshold


print(f"\nNumber of normal samples misclassified as anomalies:
{np.sum(anomalies_detected_normal)}")
print(f"Number of anomaly samples correctly detected:
{np.sum(anomalies_detected_anomaly)}")
print(f"Total anomaly samples: {len(mse_anomaly)}")

# Visualize some reconstructions
n = 5 # Number of examples to visualize
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original normal image
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test_normal[i].reshape(28, 28))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax.set_title(f"Original Normal ({y_test_normal[i]})")

    # Display reconstructed normal image
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(x_test_normal_pred[i].reshape(28, 28))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax.set_title(f"Reconstructed (Err: {mse_normal[i]:.3f})")
plt.suptitle("Reconstructions of Normal Data")
plt.show()

plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original anomaly image
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test_anomaly[i].reshape(28, 28))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax.set_title(f"Original Anomaly ({y_test_anomaly[i]})")

    # Display reconstructed anomaly image
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(x_test_anomaly_pred[i].reshape(28, 28))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax.set_title(f"Reconstructed (Err: {mse_anomaly[i]:.3f})")
plt.suptitle("Reconstructions of Anomaly Data")
plt.show()
```

**Conclusion**: This practical illustrated how an autoencoder can be effectively used for anomaly detection. By training on normal data, the autoencoder learns to reconstruct it with low error. Anomalous data, deviating from the learned patterns, results in a significantly higher reconstruction error, allowing us to identify these outliers. The choice of normal classes and the threshold are critical for performance.