explain the code #ASS2

```
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import classification_report
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt
import numpy as np

print("[INFO] accessing MNIST...")

((trainX,trainY), (testX,testY)) = mnist.load_data()

trainX.shape

trainX = trainX.reshape((trainX.shape[0], 28*28* 1))
testX = testX.reshape((testX.shape[0], 28 * 28 * 1))

trainX = trainX.astype("float32") / 255.0
testX= testX.astype("float32") / 255.0

lb=LabelBinarizer()
trainY= lb.fit_transform(trainY)
testY = lb.transform (testY)

model = Sequential()

model.add(Dense (256, input_shape=(784,), activation="relu"))
model.add(Dense (128, activation="relu"))
model.add(Dense (64, activation="relu"))
model.add(Dense(10, activation="softmax"))

print("[INFO] training network...")
Adm = Adam(0.01)
model.compile(loss="categorical_crossentropy", optimizer = Adm,
metrics=["accuracy"])
H = model.fit(trainX, trainY, validation_data=(testX, testY),
epochs=100, batch_size=128)

print("[INFO] evaluating network...")
predictions = model.predict(testX, batch_size=128)
print(classification_report(testY.argmax(axis=1),
predictions.argmax(axis=1), target_names=[str(i) for i in range(10)]))

plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 100), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, 100), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
```

This code trains a neural network to classify handwritten digits from the MNIST dataset. Here's a breakdown of each part:

1. **Import Libraries**:
   - `LabelBinarizer` and `classification_report` from `sklearn` for one-hot encoding labels and evaluating the model.
   - `Sequential`, `Dense`, `Adam` from `tensorflow.keras` for defining and training the neural network model.
   - `mnist` from `tensorflow.keras.datasets` to load the MNIST dataset.
   - `matplotlib.pyplot` and `numpy` for plotting and numerical operations.
2. **Load the MNIST Data**:

   python Copy code

   ```
   print("[INFO] accessing MNIST...") ((trainX, trainY), (testX, testY)) =
   mnist.load_data()
   ```

   - Loads the MNIST dataset, where `trainX` and `testX` are images, and `trainY` and `testY` are labels.
3. **Preprocess the Data**:

   python Copy code

   ```
   trainX = trainX.reshape((trainX.shape[0], 28 * 28 * 1)) testX =
   testX.reshape((testX.shape[0], 28 * 28 * 1)) trainX = trainX.astype("float32") /
   255.0 testX = testX.astype("float32") / 255.0
   ```

   - The images are reshaped from 28x28 pixels to a flat 784-pixel array for each image, and the pixel values are scaled to a [0, 1] range.
4. **One-Hot Encode Labels**:

   python Copy code

   ```
   lb = LabelBinarizer() trainY = lb.fit_transform(trainY) testY = lb.transform(testY)
   ```

   - Converts digit labels to one-hot encoded format for multi-class classification.
5. **Build the Model**:

   python Copy code

   ```
   model = Sequential() model.add(Dense(256, input_shape=(784,), activation="relu"))
   model.add(Dense(128, activation="relu")) model.add(Dense(64, activation="relu"))
   model.add(Dense(10, activation="softmax"))
   ```

   - A sequential neural network model is created with four layers:
     - 1st Layer: 256 neurons with ReLU activation.
     - 2nd Layer: 128 neurons with ReLU activation.
     - 3rd Layer: 64 neurons with ReLU activation.
     - 4th Layer: 10 neurons with softmax activation for classification across 10 digits (0-9).
6. **Compile and Train the Model**:

   python Copy code

```
Adm = Adam(0.01) model.compile(loss="categorical_crossentropy", optimizer=Adm,
metrics=["accuracy"]) H = model.fit(trainX, trainY, validation_data=(testX, testY),
epochs=100, batch_size=128)
```

- The model uses the Adam optimizer with a learning rate of 0.01 and is trained for 100 epochs with a batch size of 128.
- Training history, including loss and accuracy, is stored in `H`.

7. **Evaluate the Model**:

python Copy code

```
predictions = model.predict(testX, batch_size=128)
print(classification_report(testY.argmax(axis=1), predictions.argmax(axis=1),
target_names=[str(i) for i in range(10)]))
```

- The model's predictions on the test set are compared to the true labels, and a classification report shows precision, recall, and F1-score.

8. **Plot Training History**:

python Copy code

```
plt.style.use("ggplot") plt.figure() plt.plot(np.arange(0, 100), H.history["loss"],
label="train_loss") plt.plot(np.arange(0, 100), H.history["val_loss"],
label="val_loss") plt.plot(np.arange(0, 100), H.history["accuracy"],
label="train_acc") plt.plot(np.arange(0, 100), H.history["val_accuracy"],
label="val_acc") plt.title("Training Loss and Accuracy") plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy") plt.legend()
```

- Plots the loss and accuracy over the epochs for both training and validation sets to visually assess the model's performance over time.

Overall, this code builds, trains, and evaluates a simple neural network for digit classification on the MNIST dataset, followed by a plot of its training and validation performance.

Today

```
#ASS3.1
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import datasets,layers,models
import matplotlib.pyplot as plt

cifar10=keras.datasets.cifar10
(train_images, train_labels), (test_images, test_labels) =
datasets.cifar10.load_data()

train_images, test_images = train_images/255.0, test_images/255.0

print("train_images: ", train_images.shape)
print("test_images: ", test_images.shape)

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',
'horse','ship','truck']

plt.figure()
plt.imshow(train_images[5],cmap=plt.cm.binary)
plt.colorbar()
```

```
plt.grid(False)
plt.xlabel(class_names[train_labels[5][0]])
plt.show()

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i],cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()

model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32,
32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.Flatten())
model.add(layers.Dense(64, activation ='relu'))
model.add(layers.Dense(10))

model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=10,
validation_data=(test_images, test_labels))

plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label= 'val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose =
2)
```

This code trains a Convolutional Neural Network (CNN) to classify images from the CIFAR-10 dataset, which contains images from 10 classes (e.g., airplane, automobile, bird, etc.). Here's an explanation of each part:

1. **Import Libraries**:

   python Copy code

   ```
   import tensorflow as tf from tensorflow import keras from tensorflow.keras import
   datasets, layers, models import matplotlib.pyplot as plt
   ```

   - Imports `TensorFlow` for building and training the neural network, and `matplotlib` for visualizing images and results.

2. **Load and Preprocess the CIFAR-10 Dataset**:

   python Copy code

```
cifar10 = keras.datasets.cifar10 (train_images, train_labels), (test_images,
test_labels) = datasets.cifar10.load_data() train_images, test_images =
train_images / 255.0, test_images / 255.0
```

- Loads CIFAR-10 data, splitting it into training and test sets.
- Pixel values are scaled to the range [0, 1] by dividing by 255 for normalization.

3. **Data Shape and Class Labels**:

python Copy code

```
print("train_images: ", train_images.shape) print("test_images: ",
test_images.shape)
```

- Prints the shapes of the training and testing datasets. CIFAR-10 has images of size 32x32 with 3 color channels.

4. **Visualize a Sample Image**:

python Copy code

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',
'horse', 'ship', 'truck'] plt.figure() plt.imshow(train_images[5],
cmap=plt.cm.binary) plt.colorbar() plt.grid(False)
plt.xlabel(class_names[train_labels[5][0]]) plt.show()
```

- Shows a single training image with its label. The label corresponds to an entry in `class_names`.

5. **Display a Grid of Sample Images**:

python Copy code

```
plt.figure(figsize=(10,10)) for i in range(25): plt.subplot(5,5,i+1) plt.xticks([])
plt.yticks([]) plt.grid(False) plt.imshow(train_images[i], cmap=plt.cm.binary)
plt.xlabel(class_names[train_labels[i][0]]) plt.show()
```

- Displays 25 sample images in a 5x5 grid, with labels to provide a visual overview of the data.

6. **Define the CNN Model**:

python Copy code

```
model = models.Sequential() model.add(layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(32, 32, 3))) model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2))) model.add(layers.Conv2D(64, (3, 3),
activation='relu')) model.add(layers.Flatten()) model.add(layers.Dense(64,
activation='relu')) model.add(layers.Dense(10))
```

- A Sequential CNN model is created with layers:
  - 1st Layer: 32 filters, 3x3 kernel, ReLU activation.
  - MaxPooling layer to downsample.
  - 2nd Layer: 64 filters, 3x3 kernel, ReLU activation.
  - MaxPooling layer to downsample.
  - 3rd Layer: 64 filters, 3x3 kernel, ReLU activation.
  - Flatten layer to convert the 2D output to a 1D vector.
  - Dense layer with 64 neurons and ReLU activation.

- Output Dense layer with 10 units (one for each class). No activation here, as it's handled in the loss function.

7. **Compile and Train the Model**:

```python
model.compile(optimizer='adam',
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True), metrics=
['accuracy']) history = model.fit(train_images, train_labels, epochs=10,
validation_data=(test_images, test_labels))
```

   - Compiles the model using the Adam optimizer, sparse categorical cross-entropy loss (as labels are integers), and accuracy as the evaluation metric.
   - Trains for 10 epochs with both training and validation data.

8. **Plot Accuracy over Epochs**:

```python
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label='val accuracy') plt.xlabel('Epoch')
plt.ylabel('Accuracy') plt.ylim([0.5, 1]) plt.legend(loc='lower right')
```

   - Plots training and validation accuracy over the epochs to visualize model performance.

9. **Evaluate the Model on Test Data**:

```python
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
```

   - Evaluates the trained model on the test set, outputting the final test accuracy.

This code builds a basic CNN model, trains it on CIFAR-10, and provides visual and quantitative assessments of model performance.

```
#ASS3.2
from tensorflow import keras
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.metrics import classification_report

fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

print(train_images.shape)
print(train_labels.shape)
print(test_images.shape)
print(test_labels.shape)

plt.figure()
plt.imshow(train_images[5],cmap=plt.cm.binary)
plt.colorbar()
plt.grid(False)
```

```
plt.show()

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()


train_images = train_images / 255.0
test_images = test_images / 255.0

print("train_images: ", train_images.shape)
print("test_images: ", test_images.shape)

model = keras.Sequential([
    keras.layers.Conv2D(filters=64, kernel_size=3, activation='relu',
padding='same', input_shape=(28, 28, 1)),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(filters=128, kernel_size=3, activation='relu',
padding='same'),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dropout(0.25),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dropout(0.25),
    keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

model_history = model.fit(train_images, train_labels, batch_size=50,
epochs=10, steps_per_epoch=840,validation_data=(test_images,
test_labels))

pd.DataFrame(model_history.history).plot()
plt.show()

test_loss, test_acc = model.evaluate(test_images, test_labels,
verbose=2)
print('\nTest accuracy:' , test_acc)

predictions = model.predict(test_images)

def plot_img_label(img, pred_class, pred_percentage, true_class):
    plt.imshow(img,cmap=plt.cm.binary)
    if pred_class == true_class:
        color = 'blue'
    else:
        color='red'
    plt.title(label= f"Predicted: {pred_class} -
{pred_percentage:2.1f}%\nActual: {true_class}",
              fontdict={'color': color})
```

```
plt.figure(figsize=(14,10))

for i in range(20):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    i=i*5
    img = test_images[i].reshape(28,28)
    pred_class = class_names[np.argmax(predictions[i])]
    pred_percentage = np.max(predictions[i])*100
    true_class = class_names[np.argmax(test_labels[i])]

    plot_img_label(img, pred_class, pred_percentage, true_class)

plt.tight_layout()
plt.show()

predicted_label = np.argmax(predictions, axis=1) if predictions.ndim > 1 else predictions
true_label = np.argmax(test_labels, axis=1) if test_labels.ndim > 1 else test_labels

print(classification_report(true_label, predicted_label, target_names=class_names))
```

This code trains a Convolutional Neural Network (CNN) to classify images from the Fashion MNIST dataset, which includes grayscale images of clothing items across 10 categories (e.g., T-shirt/top, Trouser, etc.). Here's a breakdown:

1. **Import Libraries**:

   python Copy code

   ```
   from tensorflow import keras import matplotlib.pyplot as plt import numpy as np
   import pandas as pd from sklearn.metrics import classification_report
   ```

   - `tensorflow` for building and training the CNN model, `matplotlib` for visualization, `numpy` for numerical operations, `pandas` for data handling, and `sklearn.metrics` for model evaluation.

2. **Load and Display Dataset Information**:

   python Copy code

   ```
   fashion_mnist = keras.datasets.fashion_mnist (train_images, train_labels),
   (test_images, test_labels) = fashion_mnist.load_data()
   ```

   - Loads Fashion MNIST data into training and test sets.

3. **Class Names and Data Shape**:

   python Copy code

   ```
   class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal',
   'Shirt', 'Sneaker', 'Bag', 'Ankle boot'] print(train_images.shape)
   print(train_labels.shape) print(test_images.shape) print(test_labels.shape)
   ```

   - Defines class labels and prints shapes of the training and testing datasets.

4. **Visualize Sample Images**:

python Copy code

```
plt.figure() plt.imshow(train_images[5], cmap=plt.cm.binary) plt.colorbar()
plt.grid(False) plt.show()
```

- Displays a single training image with a grayscale color map.

python Copy code

```
plt.figure(figsize=(10,10)) for i in range(25): plt.subplot(5,5,i+1) plt.xticks([])
plt.yticks([]) plt.grid(False) plt.imshow(train_images[i], cmap=plt.cm.binary)
plt.xlabel(class_names[train_labels[i]]) plt.show()
```

- Shows a 5x5 grid of 25 sample images with labels.

5. **Normalize the Image Data**:

python Copy code

```
train_images = train_images / 255.0 test_images = test_images / 255.0
```

- Normalizes pixel values to the range [0, 1].

6. **Define the CNN Model**:

python Copy code

```
model = keras.Sequential([ keras.layers.Conv2D(64, kernel_size=3,
activation='relu', padding='same', input_shape=(28, 28, 1)),
keras.layers.MaxPool2D(pool_size=2), keras.layers.Conv2D(128, kernel_size=3,
activation='relu', padding='same'), keras.layers.MaxPool2D(pool_size=2),
keras.layers.Flatten(), keras.layers.Dense(128, activation='relu'),
keras.layers.Dropout(0.25), keras.layers.Dense(64, activation='relu'),
keras.layers.Dropout(0.25), keras.layers.Dense(10, activation='softmax') ])
```

- Defines a CNN model with:
  - 1st Convolutional Layer: 64 filters, 3x3 kernel, ReLU activation, with padding to preserve spatial dimensions.
  - MaxPooling to downsample.
  - 2nd Convolutional Layer: 128 filters, 3x3 kernel, ReLU activation, with padding.
  - Another MaxPooling layer.
  - Flatten layer to convert 2D output to 1D.
  - Dense layers with 128 and 64 neurons and ReLU activation, with 25% dropout for regularization.
  - Output layer with 10 neurons (softmax for multiclass classification).

7. **Compile and Train the Model**:

python Copy code

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=
['accuracy']) model_history = model.fit(train_images, train_labels, batch_size=50,
epochs=10, steps_per_epoch=840, validation_data=(test_images, test_labels))
```

- Compiles the model using the Adam optimizer, sparse categorical cross-entropy loss (suitable for integer labels), and accuracy metric.
- Trains the model for 10 epochs with batch size 50 and validation on the test set.

8. **Plot Training History**:

python Copy code

```
pd.DataFrame(model_history.history).plot() plt.show()
```

- Uses Pandas to plot the training and validation accuracy/loss across epochs for visual analysis.

9. **Evaluate the Model on Test Data**:

python Copy code

```
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print('\nTest accuracy:', test_acc)
```

- Evaluates the model on test data and prints the final test accuracy.

10. **Make Predictions and Define Visualization**:

python Copy code

```
predictions = model.predict(test_images) def plot_img_label(img, pred_class,
pred_percentage, true_class): plt.imshow(img, cmap=plt.cm.binary) color = 'blue' if
pred_class == true_class else 'red' plt.title(f"Predicted: {pred_class} -
{pred_percentage:2.1f}%\nActual: {true_class}", fontdict={'color': color})
```

- Stores model predictions on test images.
- Defines a function to visualize predictions with title colors indicating correct (blue) or incorrect (red) predictions.

11. **Display Prediction Results for a Few Images**:

python Copy code

```
plt.figure(figsize=(14,10)) for i in range(20): plt.subplot(5,5,i+1) plt.xticks([])
plt.yticks([]) plt.grid(False) i = i * 5 img = test_images[i].reshape(28, 28)
pred_class = class_names[np.argmax(predictions[i])] pred_percentage =
np.max(predictions[i]) * 100 true_class = class_names[test_labels[i][0]]
plot_img_label(img, pred_class, pred_percentage, true_class) plt.tight_layout()
plt.show()
```

- Shows 20 images with predictions, where correct predictions are in blue and incorrect in red.

12. **Generate Classification Report**:

python Copy code

```
predicted_label = np.argmax(predictions, axis=1) true_label = test_labels
print(classification_report(true_label, predicted_label, target_names=class_names))
```

- Converts predictions to class labels, then generates a classification report showing precision, recall, and F1-score for each class.

Overall, this code builds, trains, and evaluates a CNN on Fashion MNIST, providing visualizations of predictions and model performance.

#ASS4
# Importing libraries
import numpy as np
import pandas as pd
import tensorflow as tf

```
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras import Model, Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.model_selection import train_test_split

# Define the path to the dataset. You can change this to your local file
path if needed.
path =
'http://storage.googleapis.com/download.tensorflow.org/data/ecg.csv'

# Read the ECG dataset into a Pandas DataFrame
data = pd.read_csv(path, header=None)

data.head()

# Get information about the dataset, such as column data types and
non-null counts
data.info()

# Splitting the dataset into features and target
features = data.drop(140, axis=1)  # Features are all columns except
the last (column 140)
target = data[140]  # Target is the last column (column 140)

# Split the data into training and testing sets (80% training, 20%
testing)
x_train, x_test, y_train, y_test = train_test_split(
    features, target, test_size=0.2
)

# Get the indices of the training data points labeled as "1" (anomalies)
train_index = y_train[y_train == 1].index

# Select the training data points that are anomalies
train_data = x_train.loc[train_index]

# Initialize the Min-Max Scaler to scale the data between 0 and 1
min_max_scaler = MinMaxScaler(feature_range=(0, 1))

# Scale the training data
x_train_scaled = min_max_scaler.fit_transform(train_data.copy())

# Scale the testing data using the same scaler
x_test_scaled = min_max_scaler.transform(x_test.copy())

# Creating an Autoencoder model by extending the Model class from
Keras
class AutoEncoder(Model):
    def __init__(self, output_units, ldim=8):
        super().__init__()
        # Define the encoder part of the Autoencoder
        self.encoder = Sequential([
            Dense(64, activation='relu'),
            Dropout(0.1),
            Dense(32, activation='relu'),
            Dropout(0.1),
            Dense(16, activation='relu'),
            Dropout(0.1),
```

```
            Dense(ldim, activation='relu')
        ])
        # Define the decoder part of the Autoencoder
        self.decoder = Sequential([
            Dense(16, activation='relu'),
            Dropout(0.1),
            Dense(32, activation='relu'),
            Dropout(0.1),
            Dense(64, activation='relu'),
            Dropout(0.1),
            Dense(output_units, activation='sigmoid')
        ])

    def call(self, inputs):
        # Forward pass through the Autoencoder
        encoded = self.encoder(inputs)
        decoded = self.decoder(encoded)
        return decoded

# Create an instance of the AutoEncoder model with the appropriate
output units
model = AutoEncoder(output_units=x_train_scaled.shape[1])

# Compile the model with Mean Squared Logarithmic Error (MSLE)
loss and Mean Squared Error (MSE) metric
model.compile(loss='msle', metrics=['mse'], optimizer='adam')

# Train the model using the scaled training data
history = model.fit(
    x_train_scaled,  # Input data for training
    x_train_scaled,  # Target data for training (autoencoder reconstructs
the input)
    epochs=20,       # Number of training epochs
    batch_size=512,   # Batch size
    validation_data=(x_test_scaled, x_test_scaled),  # Validation data
    shuffle=True     # Shuffle the data during training
)

plt.plot(history.history["loss"], label="Training Loss")
plt.plot(history.history["val_loss"], label="Validation Loss")
plt.legend()

# Function to find the threshold for anomalies based on the training
data
def find_threshold(model, x_train_scaled):
    # Reconstruct the data using the model
    recons = model.predict(x_train_scaled)

    # Calculate the mean squared log error between reconstructed data
and the original data
    recons_error = tf.keras.metrics.msle(recons, x_train_scaled)

    # Set the threshold as the mean error plus one standard deviation
    threshold = np.mean(recons_error.numpy()) +
np.std(recons_error.numpy())

    return threshold

# Function to make predictions for anomalies based on the threshold
def get_predictions(model, x_test_scaled, threshold):
```

```
    # Reconstruct the data using the model
    predictions = model.predict(x_test_scaled)

    # Calculate the mean squared log error between reconstructed data
and the original data
    errors = tf.keras.losses.msle(predictions, x_test_scaled)

    # Create a mask for anomalies based on the threshold
    anomaly_mask = pd.Series(errors) > threshold

    # Map True (anomalies) to 0 and False (normal data) to 1
    preds = anomaly_mask.map(lambda x: 0.0 if x == True else 1.0)

    return preds

# Find the threshold for anomalies
threshold = find_threshold(model, x_train_scaled)
print(f"Threshold: {threshold}")

# Get predictions for anomalies based on the model and threshold
predictions = get_predictions(model, x_test_scaled, threshold)

# Calculate the accuracy score by comparing the predicted anomalies
to the true labels
accuracy = accuracy_score(predictions, y_test)

# Print the accuracy score
print(f"Accuracy Score: {accuracy}")
```

This code is for detecting anomalies in ECG data using an autoencoder neural network built with TensorFlow and Keras. The autoencoder attempts to reconstruct normal ECG patterns and flags deviations as anomalies. Here's a step-by-step explanation:

1. **Import Libraries**

python Copy code

```
import numpy as np import pandas as pd import tensorflow as tf import matplotlib.pyplot
as plt from sklearn.metrics import accuracy_score from sklearn.preprocessing import
MinMaxScaler from tensorflow.keras import Model, Sequential from tensorflow.keras.layers
import Dense, Dropout from sklearn.model_selection import train_test_split
```

- Libraries include TensorFlow for the model, Matplotlib for visualization, Pandas and NumPy for data handling, and Scikit-learn for metrics and data splitting.

2. **Load and Inspect Data**

python Copy code

```
path = 'http://storage.googleapis.com/download.tensorflow.org/data/ecg.csv' data =
pd.read_csv(path, header=None) data.info()
```

- Loads an ECG dataset from a CSV file. `data.info()` displays column types and non-null counts, helpful for data preparation.

3. **Split Data into Features and Target**

python Copy code

```
features = data.drop(140, axis=1) target = data[140]
```

- Splits `features` (all columns except the last) and `target` (last column). Column 140 contains labels, where `1` usually represents anomalies.

4. Train-Test Split and Extract Anomalies

python Copy code

```
x_train, x_test, y_train, y_test = train_test_split(features, target, test_size=0.2)
train_index = y_train[y_train == 1].index train_data = x_train.loc[train_index]
```

- Splits data into training and testing sets with an 80/20 ratio.
- Isolates training instances with label `1` (anomalies) for model training.

5. Data Scaling

python Copy code

```
min_max_scaler = MinMaxScaler(feature_range=(0, 1)) x_train_scaled =
min_max_scaler.fit_transform(train_data.copy()) x_test_scaled =
min_max_scaler.transform(x_test.copy())
```

- Scales data to the range [0,1] using `MinMaxScaler`, which improves model performance by normalizing the input.

6. Define Autoencoder Model

python Copy code

```
class AutoEncoder(Model): def __init__(self, output_units, ldim=8): super().__init__()
self.encoder = Sequential([Dense(64, activation='relu'), Dropout(0.1), Dense(32,
activation='relu'), Dropout(0.1), Dense(16, activation='relu'), Dropout(0.1),
Dense(ldim, activation='relu')]) self.decoder = Sequential([Dense(16,
activation='relu'), Dropout(0.1), Dense(32, activation='relu'), Dropout(0.1), Dense(64,
activation='relu'), Dropout(0.1), Dense(output_units, activation='sigmoid')]) def
call(self, inputs): encoded = self.encoder(inputs) decoded = self.decoder(encoded)
return decoded
```

- Defines an autoencoder with:
  - Encoder layers to compress data and Decoder layers to reconstruct it.
  - Dropout layers for regularization, with `ldim=8` as the latent dimension size.
  - Output units set to the number of features.

7. Compile and Train the Model

python Copy code

```
model = AutoEncoder(output_units=x_train_scaled.shape[1]) model.compile(loss='msle',
metrics=['mse'], optimizer='adam') history = model.fit(x_train_scaled, x_train_scaled,
epochs=20, batch_size=512, validation_data=(x_test_scaled, x_test_scaled), shuffle=True)
plt.plot(history.history["loss"], label="Training Loss")
plt.plot(history.history["val_loss"], label="Validation Loss") plt.legend()
```

- Compiles the autoencoder with Mean Squared Logarithmic Error (MSLE) as the loss function and Mean Squared Error (MSE) as a metric, training it for 20 epochs with batch size 512.

- Plots the training and validation loss to monitor performance over epochs.

8. **Determine Anomaly Threshold**

python Copy code

```
def find_threshold(model, x_train_scaled): recons = model.predict(x_train_scaled)
recons_error = tf.keras.metrics.msle(recons, x_train_scaled) threshold =
np.mean(recons_error.numpy()) + np.std(recons_error.numpy()) return threshold
```

- Calculates reconstruction error on the training data and sets the anomaly threshold as the mean error plus one standard deviation.

9. **Predict Anomalies in Test Data**

python Copy code

```
def get_predictions(model, x_test_scaled, threshold): predictions =
model.predict(x_test_scaled) errors = tf.keras.losses.msle(predictions, x_test_scaled)
anomaly_mask = pd.Series(errors) > threshold preds = anomaly_mask.map(lambda x: 0.0 if x
== True else 1.0) return preds
```

- Reconstructs the test data, calculates the reconstruction error, and classifies points as anomalies if their error exceeds the threshold.
- Labels anomalies as `0` and normal points as `1`.

10. **Calculate and Print Accuracy**

python Copy code

```
threshold = find_threshold(model, x_train_scaled) print(f"Threshold: {threshold}")
predictions = get_predictions(model, x_test_scaled, threshold) accuracy =
accuracy_score(predictions, y_test) print(f"Accuracy Score: {accuracy}")
```

- Computes the accuracy score by comparing the model's predictions to the actual labels, displaying the threshold and accuracy score.

This approach uses the autoencoder to detect anomalies by reconstructing normal ECG patterns and flags any high-reconstruction-error instances as anomalies.

```
#ASS5
import warnings
warnings.filterwarnings(action = 'ignore')

import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib as mpl
import matplotlib.pylab as pylab
import numpy as np

%matplotlib inline
from nltk.tokenize import sent_tokenize, word_tokenize

pip install gensim

pip install --upgrade gensim
```

```
import gensim
from gensim.models import Word2Vec

import re
import bs4 as bs
import urllib.request

scrapped_data=urllib.request.urlopen("https://en.wikipedia.org/wiki/M
achine_learning")
article=scrapped_data.read()

paresed_article=bs.BeautifulSoup(article, 'lxml')

paragraphs = paresed_article.find_all('p')

article_text=""
for p in paragraphs:
    article_text+=p.text
    sentences=article_text
print(article_text)

sentences="""Alice 23 opened the door and found that it led into a
small 90
passage, not much larger than a rat-hole: she knelt down and
looked along the passage into the loveliest garden you ever saw.
How she longed to get out of that dark hall, and wander about
among those beds of bright flowers and those cool fountains, but
she could not even get her head through the doorway; and even if
my head would go through, (thought) Spoor Alice, it would be of
very little use without my shoulders. Oh, how I wish
I could shut up like a telescope! I think I could, if I only
know how to begin. For, you see, so many out-of-the-way things
had happened lately, that Alice had begun to think that very few
things indeed were really impossible.
"""
sentences = re.sub('[^A-Za-z0-9]+',' ', sentences)

sentences = re.sub(r'(?:^| )\w(?:$| )', ' ', sentences).strip()

print(sentences)

import nltk
from nltk.corpus import stopwords

#nltk.download('stopwords')
#nltk.download('punkt')

sentences = re.sub('[^A-Za-z]+', ' ', sentences)

sentences = re.sub(r'(?:^| )\w(?:$| )', ' ', sentences).strip()

sentences = sentences.lower()

all_sent = nltk.sent_tokenize(sentences)

all_words=[nltk.word_tokenize(sent) for sent in all_sent]


for i in range(len(all_words)):
    all_words[i]=[w for w in all_words[i] if w not in
```

```
stopwords.words('english')]

data = all_words

data1 = data[0]

print(data)

model1= gensim.models.Word2Vec(data, min_count = 1, vector_size =
52, window= 5)

vocabulary = model1.wv.key_to_index["door"]
print(vocabulary)
vocabulary = model1.wv.key_to_index
print(vocabulary)

wrd='door'
v1=model1.wv[wrd]
similar_words=model1.wv.most_similar(wrd)

for x in similar_words:
    print(x)

dat= []
print(len(data1))

for i in range(2, len(data1) - 2):
    print(i)
    context = [data1[i - 2], data1[i - 1], data1[i+1], data1[i + 2]]
    target = data1[i]
    print(context, target)
    dat.append((context, target))
    print(dat)
print(dat[:5])

[]
i=3
print(dat[i][0],dat[i][1])

print(model1.predict_output_word(dat[i][0]))

from sklearn.decomposition import PCA
from matplotlib import pyplot as plt

X = model1.wv[model1.wv.key_to_index]
pca = PCA(n_components=2)
result = pca.fit_transform(X)

plt.figure(figsize=(10, 10))
plt.scatter(result[:, 0], result[:, 1])

words = list(model1.wv.key_to_index)
for i, word in enumerate(words[:50]):
    plt.annotate(word, xy=(result[i, 0], result[i, 1]), fontsize=9)


plt.title('PCA of Word Embeddings')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
```

```
plt.grid()
plt.show()
```

This code performs natural language processing on text scraped from a Wikipedia article about machine learning. It uses the Word2Vec model from Gensim to analyze word relationships and then visualizes word embeddings using PCA. Here's an explanation of each section:

1. Import Libraries and Configure Environment

python Copy code

```
import warnings warnings.filterwarnings(action='ignore') import matplotlib.pyplot as
plt import seaborn as sns import numpy as np %matplotlib inline
```

- Imports necessary libraries for text processing and visualization.
- Ignores warnings for a cleaner output.

2. Install and Import Gensim and Other Libraries

python Copy code

```
pip install gensim pip install --upgrade gensim import gensim from gensim.models import
Word2Vec import re import bs4 as bs import urllib.request
```

- Installs and imports Gensim for word embeddings.
- Imports `re` for regex operations, `bs4` for web scraping, and `urllib` to access the webpage.

3. Scrape and Parse Wikipedia Article

python Copy code

```
scrapped_data =
urllib.request.urlopen("https://en.wikipedia.org/wiki/Machine_learning") article =
scrapped_data.read() parsed_article = bs.BeautifulSoup(article, 'lxml') paragraphs =
parsed_article.find_all('p') article_text = "" for p in paragraphs: article_text +=
p.text print(article_text)
```

- Retrieves the machine learning Wikipedia page and extracts the article text.
- Combines all `<p>` tags to form the article's text and prints it.

4. Text Preprocessing

python Copy code

```
sentences = """<sample text>""" # Text sample for demonstration sentences =
re.sub('[^A-Za-z0-9]+', ' ', sentences) sentences = re.sub(r'(?:^| )\w(?:$| )', ' ',
sentences).strip() print(sentences)
```

- The code defines a sample text for further processing.
- Removes non-alphanumeric characters and single letters to clean the text.

5. Tokenize Text and Remove Stopwords

python Copy code

```
import nltk from nltk.corpus import stopwords sentences = re.sub('[^A-Za-z]+', ' ',
sentences) sentences = sentences.lower() all_sent = nltk.sent_tokenize(sentences)
```

```
all_words = [nltk.word_tokenize(sent) for sent in all_sent] for i in
range(len(all_words)): all_words[i] = [w for w in all_words[i] if w not in
stopwords.words('english')] data = all_words print(data)
```

- Converts sentences to lowercase, tokenizes them, and removes English stopwords using NLTK.

6. Train Word2Vec Model

python Copy code

```
model1 = gensim.models.Word2Vec(data, min_count=1, vector_size=52, window=5) vocabulary
= model1.wv.key_to_index print(vocabulary)
```

- Trains a Word2Vec model on the tokenized words with a vector size of 52 and a context window of 5.
- Prints the vocabulary to verify that words are processed.

7. Retrieve and Print Similar Words

python Copy code

```
word = 'door' similar_words = model1.wv.most_similar(word) for x in similar_words:
print(x)
```

- Finds and prints words similar to 'door' based on the trained Word2Vec model.

8. Generate Context-Target Pairs for Training

python Copy code

```
dat = [] for i in range(2, len(data1) - 2): context = [data1[i - 2], data1[i - 1],
data1[i + 1], data1[i + 2]] target = data1[i] dat.append((context, target))
print(dat[:5])
```

- Creates context-target pairs from the sample data to better understand the context of each word.

9. Predict Word Using Word2Vec

python Copy code

```
i = 3 print(dat[i][0], dat[i][1]) print(model1.predict_output_word(dat[i][0]))
```

- Tests the model by predicting the target word based on a sample context from the generated pairs.

10. Visualize Word Embeddings Using PCA

python Copy code

```
from sklearn.decomposition import PCA X = model1.wv[model1.wv.key_to_index] pca =
PCA(n_components=2) result = pca.fit_transform(X) plt.figure(figsize=(10, 10))
plt.scatter(result[:, 0], result[:, 1]) words = list(model1.wv.key_to_index) for i, word
in enumerate(words[:50]): plt.annotate(word, xy=(result[i, 0], result[i, 1]),
fontsize=9) plt.title('PCA of Word Embeddings') plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2') plt.grid() plt.show()
```

- Reduces word vectors to 2D with PCA and plots them.

- Labels each point with its corresponding word and visualizes the spatial relationships in the vocabulary.

This code illustrates how Word2Vec embeddings can capture semantic similarity, and the PCA visualization shows relative word positions based on learned embeddings.


```
#ASS6

#Assignment 6: Implementation of object detection using transfer
learning of CNN architectures

from tensorflow.keras.preprocessing.image import
ImageDataGenerator

train_dir='Desktop/mnist/mnistpng/train/'
test_dir ='mnist/mnistpng/test/'

img_gen=ImageDataGenerator(rescale=1.0/255)
data_gen=img_gen.flow_from_directory(
    train_dir,
    target_size=(32,32),
    batch_size=5000,
    shuffle=True,
    class_mode='categorical'
)

x_train,y_train=data_gen[0]
x_test,y_test=data_gen[2]


from tensorflow.keras.applications import VGG16
path =
'Desktop/mnist/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5'
vgg_model = VGG16 (weights=path, include_top=False, input_shape=
(32,32,3))

for layer in vgg_model.layers:
    layer.trainabler=False

from tensorflow import keras
from tensorflow.keras.layers import Dense, Flatten, Dropout

custom_classifier = keras.Sequential([

    Flatten(),
    Dense(100, activation= 'relu'),
    Dropout (0.2),
    Dense(100, activation= 'relu'),
    Dropout (0.2),
    Dense (10, activation='softmax')
])
model=keras.Sequential([
    vgg_model,
    custom_classifier
])
```

```
model.compile(optimizer='adam', loss = 'categorical_crossentropy',
metrics=['accuracy'])

model.fit(x_train, y_train, batch_size=100, epochs=1, validation_data=
(x_test,y_test))

for layer in vgg_model.layers[:-4]:
    layer.trainable=True

model.compile(optimizer='adam',loss='categorical_crossentropy',metri
cs=['accuracy'])
model.fit(x_train, y_train, batch_size=100, epochs=1, validation_data=
(x_test,y_test))

loss,acc=model.evaluate(x_test,y_test)
print(loss, " ",acc)

pred=model.predict(x_test)

labels=list(data_gen.class_indices.keys())

import matplotlib.pyplot as plt
import numpy as np
plt.imshow(x_test[5])
print(str(labels[np.argmax(y_test[5])]))

y_test[5]
```

This code performs object detection on the MNIST dataset using transfer learning with the pre-trained VGG16 model. It fine-tunes the model to classify images by retraining the top layers with a custom classifier. Below is an explanation of the key steps in the code:

1. Import Required Libraries and Define Data Paths

python Copy code

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator train_dir =
'Desktop/mnist/mnistpng/train/' test_dir = 'Desktop/mnist/mnistpng/test/'
```

- Imports `ImageDataGenerator` for data augmentation.
- Sets paths to the training and testing directories containing the MNIST images.

2. Create an Image Data Generator

python Copy code

```
img_gen = ImageDataGenerator(rescale=1.0/255) data_gen = img_gen.flow_from_directory(
train_dir, target_size=(32,32), batch_size=5000, shuffle=True, class_mode='categorical'
) x_train, y_train = data_gen[0] x_test, y_test = data_gen[2]
```

- Rescales images by dividing pixel values by 255.
- Loads the images from `train_dir` and `test_dir`, setting `target_size=(32,32)` to resize images to fit VGG16's input requirements.
- `class_mode='categorical'` prepares labels for multi-class classification.
- Stores training and test images and labels in `x_train`, `y_train`, `x_test`, and `y_test`.

3. Load the Pre-trained VGG16 Model

python Copy code

```
from tensorflow.keras.applications import VGG16 path =
'Desktop/mnist/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5' vgg_model =
VGG16(weights=path, include_top=False, input_shape=(32,32,3)) for layer in
vgg_model.layers: layer.trainable = False
```

- Loads the VGG16 model without the top classification layer, using `include_top=False`.
- Freezes the layers initially to retain pre-trained weights.

4. Define and Add a Custom Classifier

python Copy code

```
from tensorflow.keras.layers import Dense, Flatten, Dropout custom_classifier =
keras.Sequential([ Flatten(), Dense(100, activation='relu'), Dropout(0.2), Dense(100,
activation='relu'), Dropout(0.2), Dense(10, activation='softmax') ]) model =
keras.Sequential([ vgg_model, custom_classifier ])
```

- Adds a fully connected classifier on top of the VGG16 feature extractor.
- `Dense(10, activation='softmax')` is used for the final output layer to match the 10 classes of MNIST.

5. Compile and Train the Model

python Copy code

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(x_train, y_train, batch_size=100, epochs=1, validation_data=(x_test, y_test))
```

- Compiles the model using `categorical_crossentropy` for multi-class classification.
- Initially trains the classifier for one epoch.

6. Unfreeze Some Layers of VGG16 and Re-train

python Copy code

```
for layer in vgg_model.layers[:-4]: layer.trainable = True
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(x_train, y_train, batch_size=100, epochs=1, validation_data=(x_test, y_test))
```

- Unfreezes the last four layers of the VGG16 model for fine-tuning.
- Re-trains the model to adjust weights in the last few layers of VGG16.

7. Evaluate and Make Predictions

python Copy code

```
loss, acc = model.evaluate(x_test, y_test) print(loss, acc) pred =
model.predict(x_test)
```

- Evaluates the model on the test set, printing the loss and accuracy.
- Predicts labels for test images.

8. Visualize Predictions

python Copy code

```
import matplotlib.pyplot as plt import numpy as np labels =
list(data_gen.class_indices.keys()) plt.imshow(x_test[5])
print(str(labels[np.argmax(y_test[5])])) y_test[5]
```

- Displays an image from the test set and prints its true label.

Important Notes

- This script is written for the MNIST dataset but uses `VGG16`, which expects RGB images with 3 channels. If your MNIST dataset images are grayscale (1 channel), ensure to preprocess them accordingly (e.g., convert grayscale images to RGB).
- The model only runs for one epoch; for better accuracy, consider increasing epochs and adjusting the learning rate during fine-tuning.

#ass1
import numpy as np
import tensorflow as tf

print(tf.__version__)

from keras import datasets
(train_images,train_labels), (test_images, test_labels)=
datasets.mnist.load_data()

train_images.shape

!pip3 install torch torchvision torchaudio --extra-index-url
https://download.pytorch.org/whl/cu11

import torch
import torch.nn as nn
print(torch.__version__)
torch.cuda.is_available()

This script initializes an environment to work with TensorFlow, Keras, and PyTorch on the MNIST dataset, which is widely used for training and testing machine learning models. Below is a breakdown of each section:

1. **Importing Libraries**

python Copy code

```
import numpy as np import tensorflow as tf print(tf.__version__)
```

- Imports essential libraries: `numpy` for numerical computations and `tensorflow` for building and training neural networks.
- Prints the TensorFlow version to confirm the installation.

2. **Loading the MNIST Dataset**

python Copy code

```
from keras import datasets (train_images, train_labels), (test_images, test_labels) =
datasets.mnist.load_data() train_images.shape
```

- Uses Keras's `datasets` module to load the MNIST dataset.
- MNIST is split into training and testing sets. Each image is 28x28 pixels, grayscale, and labeled with one of 10 classes (digits 0-9).
- `train_images.shape` shows the dimensions of the training data.

3. **Installing and Importing PyTorch**

python Copy code

```
!pip3 install torch torchvision torchaudio --extra-index-url
https://download.pytorch.org/whl/cu11
```

- Installs PyTorch, along with `torchvision` (for image handling) and `torchaudio` (for audio processing), with CUDA 11 support for GPU acceleration.

python Copy code

```
import torch import torch.nn as nn print(torch.__version__) torch.cuda.is_available()
```

- Imports PyTorch and `nn` for building neural networks.
- Prints the PyTorch version and checks if CUDA is available for GPU-based computations.

Key Points

- **Compatibility**: TensorFlow and PyTorch are both installed here; using them in the same environment is fine, though each has its own syntax for building and training models.
- **Next Steps**: You can now proceed with defining and training neural networks in either TensorFlow or PyTorch, depending on preference.