**ASS 1**

**import tensorflow as tf**

TensorFlow library in Python. TensorFlow is an open-source machine learning framework developed by Google that is widely used for various machine learning and deep learning tasks

**What is Keras vs TensorFlow?**

TensorFlow is an open-sourced end-to-end platform, a library for multiple machine learning tasks, while Keras is a high-level neural network library that runs on top of TensorFlow

**Theano**

Theano is a Python library that allows you to define, optimize, and efficiently evaluate mathematical expressions involving multi-dimensional arrays.

**import theano.tensor as T from theano import function**

It looks like you're trying to use Theano, a deep learning library. To create a Theano function, you can define symbolic variables and expressions using Theano's tensor module (**theano.tensor**) and then compile a function with **theano.function**.

**PyTorch**

PyTorch is an open-source deep learning framework that provides a flexible and dynamic computational graph. It allows you to define and train neural networks for various machine learning tasks.

**ASS 2**

**sklearn.preprocessing.LabelBinarizer**: This is from Scikit-learn (sklearn) and is used for label binarization, which converts class labels into binary vectors (one-hot encoding).

**sklearn.metrics.classification\_report**: Also from Scikit-learn, this function generates a text report showing various classification metrics (e.g., precision, recall, F1-score) for a classification model's performance.

**tensorflow.keras.models.Sequential**: This import is for defining a sequential neural network model in TensorFlow's Keras API. The **Sequential** model allows you to build a simple stack of layers.

**tensorflow.keras.layers.Dense**: This is for defining densely connected (fully connected) neural network layers in TensorFlow's Keras API.

**tensorflow.keras.optimizers.SGD**: Importing the Stochastic Gradient Descent (SGD) optimizer, which is commonly used for training neural networks.

**tensorflow.keras.datasets.mnist**: Importing the MNIST dataset from TensorFlow's Keras datasets, which provides the popular handwritten digits dataset for machine learning and deep learning tasks.

**tensorflow.keras.backend as K**: This import allows you to access the backend functions of TensorFlow, often used for custom loss functions and other advanced operations.

**matplotlib.pyplot as plt**: Importing Matplotlib for data visualization and plotting.

**numpy as np**: Importing NumPy for numerical operations and array handling.

**((X\_train, Y\_train), (X\_test, Y\_test)) = mnist.load\_data()**: You load the MNIST dataset using **mnist.load\_data()**. This code loads the training and testing data into **X\_train** and **X\_test** as well as their corresponding labels **Y\_train** and **Y\_test**.

**X\_train = X\_train.reshape((X\_train.shape[0], 28 \* 28 \* 1))**: You reshape the training data to a 2D array where each image is flattened to a vector of size 28281, which is 784. This is a common step when working with image data.

**X\_test = X\_test.reshape((X\_test.shape[0], 28 \* 28 \* 1))**: Similarly, you reshape the testing data to match the same format as the training data.

**X\_train = X\_train.astype("float32") / 255.0**: You convert the pixel values in the training data to floating-point values and then normalize them by dividing by 255. This scales the pixel values to the range [0, 1], which is a common practice in deep learning to ensure that features have similar scales and to improve convergence during training.

**X\_test = X\_test.astype("float32") / 255.0**: You perform the same pixel value conversion and normalization for the testing data.

**model = Sequential()**: You create a sequential model. The **Sequential** model is a linear stack of layers, where you can add layers one after the other. It's a simple way to build a feedforward neural network.

**model.add(Dense(128, input\_shape=(784,), activation="sigmoid"))**: You add the first layer to the model. This is a dense (fully connected) layer with 128 units, and it specifies the input shape as (784,), which corresponds to the flattened MNIST images. The activation function used in this layer is the sigmoid function.

**model.add(Dense(64, activation="sigmoid"))**: You add a second dense layer with 64 units. Since the input shape is not specified in this layer, it's assumed to be the output shape of the previous layer, which is 128 in this case. The activation function is again the sigmoid function.

**model.add(Dense(10, activation="softmax")**: The final layer of the model is another dense layer with 10 units. This is typically used for classification tasks, and the activation function is softmax. The softmax function assigns probabilities to each class, making it suitable for multi-class classification problems like MNIST.

**sgd = SGD(0.01)**: You create a Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.01. The optimizer is used to update the model's weights during training to minimize the specified loss function.

**epochs = 10**: You set the number of training epochs to 10. An epoch is one complete pass through the training dataset.

**model.compile(loss="categorical\_crossentropy", optimizer=sgd, metrics=["accuracy"])**: This line compiles the model. You specify the loss function as "categorical\_crossentropy," which is commonly used for multi-class classification problems. You use the SGD optimizer you defined earlier.

Additionally, you specify that you want to track the training accuracy as a metric.

**H = model.fit(X\_train, Y\_train, validation\_data=(X\_test, Y\_test), epochs=epochs, batch\_size=128)**: This line starts the training process. It fits the model to the training data (**X\_train** and **Y\_train**). You also provide the validation data (**X\_test** and **Y\_test**) to monitor the model's performance on data it hasn't seen during training. The **epochs** parameter specifies the number of training epochs, and the **batch\_size** parameter determines the size of the mini-batches used during training. The training history is stored in **H**.

**plt.style.use("ggplot")**: This line sets the style of the plot to "ggplot," which is one of the available plot styles in Matplotlib

.

**plt.figure()**: You create a new figure for the plot.

**plt.plot(np.arange(0, epochs), H.history["loss"], label="train\_loss")**: You plot the training loss values stored in **H.history["loss"]** as a function of the number of epochs. The **label** parameter is set to "train\_loss" for the legend.

**plt.plot(np.arange(0, epochs), H.history["val\_loss"], label="val\_loss")**: Similarly, you plot the validation loss values stored in **H.history["val\_loss"]** as a function of the number of epochs with the label "val\_loss" for the legend.

**plt.plot(np.arange(0, epochs), H.history["accuracy"], label="train\_acc")**: You plot the training accuracy values stored in **H.history["accuracy"]** as a function of the number of epochs, with the label "train\_acc" for the legend.

**plt.plot(np.arange(0, epochs), H.history["val\_accuracy"], label="val\_acc")**: You also plot the validation accuracy values stored in **H.history["val\_accuracy"]** as a function of the number of epochs, with the label "val\_acc" for the legend.

**Ass 3**

**plt.figure(figsize=(10,10))**: You create a new figure for the plot with a specified figure size of 10x10 inches.

**for i in range(10)**: This loop iterates through the first 10 images in the training dataset.

**plt.subplot(5,5,i+1)**: Within each iteration, you set up a subplot in a 5x5 grid. **i+1** is used to ensure that each subplot is assigned a unique index.

**plt.xticks([])**: You remove the x-axis ticks, making the plot cleaner.

**plt.yticks([])**: You remove the y-axis ticks.

**plt.grid(False)**: You turn off the grid lines for the subplot.

**plt.imshow(train\_images[i])**: You display the image from the training dataset at index **i**.

**plt.xlabel(class\_names[train\_labels[i][0]])**: Below each image, you set the x-axis label (xlabel) to the class name corresponding to the **train\_labels[i][0]**. **class\_names** is assumed to be a list or array that maps class indices to their respective names.

**plt.show()**: You display the entire figure, which contains the grid of images and labels.

**model = models.Sequential()**: You create a sequential model. This is a linear stack of layers for building a deep learning model.

**model.add(layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(32, 32, 3))**: You add a 2D convolutional layer with 32 filters, each of size (3, 3), and ReLU activation function. This is the input layer, and you specify the input shape as (32, 32, 3), which is typically used for color images with 32x32 pixels and 3 color channels (RGB).

**model.add(layers.MaxPooling2D((2, 2))**: You add a max-pooling layer with a pool size of (2, 2). Max-pooling is used to downsample the spatial dimensions of the feature maps.

**model.add(layers.Conv2D(64, (3, 3), activation='relu')**: You add a second convolutional layer with 64 filters and ReLU activation. Since you didn't specify the input shape, it assumes the output shape of the previous layer.

**model.add(layers.MaxPooling2D((2, 2))**: You add another max-pooling layer to downsample the feature maps.

**model.add(layers.Conv2D(64, (3, 3), activation='relu')**: You add a third convolutional layer with 64 filters and ReLU activation.

**model.add(layers.Flatten()**: You flatten the output from the previous layer. This converts the 2D feature maps into a 1D vector, which is typically used as input for fully connected layers.

**model.add(layers.Dense(64, activation='relu')**: You add a fully connected layer with 64 units and ReLU activation.

**model.add(layers.Dense(10)**: You add the final fully connected layer with 10 units. This layer often has no activation function when used for classification tasks.

**model.summary()**: This line prints a summary of the model architecture, showing the number of parameters and the shapes of each layer's output.

**model.compile(optimizer='adam', loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True), metrics=['accuracy'])**: You compile the model. The **optimizer** is set to 'adam,' which is a popular optimizer for training deep learning models. The **loss** is specified as **tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True)**, which is used for multi-class classification tasks when the model's final layer does not apply a softmax activation. The **metrics** parameter specifies that you want to track the accuracy during training.

**epochs = 1**: You set the number of training epochs to 1. One epoch means the model will be trained once on the entire training dataset.

**h = model.fit(train\_images, train\_labels, epochs=epochs, validation\_data=(test\_images, test\_labels))**: This line starts the training process. The model is trained on the **train\_images** and **train\_labels** for 1 epoch. The **validation\_data** parameter is set to **(test\_images, test\_labels)** to evaluate the model's performance on the test dataset. The training history is stored in the **h** variable.

**Ass 4**

**features = data.drop(140, axis=1)**: You are creating a DataFrame **features** by dropping the column with index 140 from the original **data** DataFrame. This is a common operation to separate the input features from the target variable.

**target = data[140]**: You are creating a Series **target** by selecting the column with index 140 from the original **data** DataFrame. This column is typically the target variable or label.

**x\_train, x\_test, y\_train, y\_test = train\_test\_split(features, target, test\_size=0.2, stratify=target)**: You are using the **train\_test\_split** function from scikit-learn to split your dataset into training and testing sets. The input features are in **features**, and the target variable is in **target**. The **test\_size** parameter specifies the proportion of the dataset to include in the test split, and **stratify=target** ensures that the class distribution is maintained in both the training and testing sets. The resulting sets are **x\_train**, **x\_test**, **y\_train**, and **y\_test**.

**train\_index = y\_train[y\_train == 1].index**: You are identifying the indices in the training data where the target variable **y\_train** has a value of 1. This is commonly done to isolate instances associated with a specific class, often used for binary classification.

**train\_data = x\_train.loc[train\_index]**: You are selecting the corresponding input features from the training data using the indices identified in **train\_index**. This creates a subset of the training data containing only the instances where the target variable is 1, which can be useful for training a binary classification model.

**class AutoEncoder(Model)**: You define a custom class **AutoEncoder** that inherits from **tf.keras.Model**, indicating that this class is a Keras model.

In the **\_\_init\_\_** method, you define the architecture of the autoencoder. An autoencoder consists of two parts: an encoder and a decoder.

* + The **encoder** is a Keras **Sequential** model that defines the encoding part of the autoencoder. It includes a series of densely connected layers with ReLU activation functions and dropout layers.
  + The **decoder** is another Keras **Sequential** model that defines the decoding part of the autoencoder. It also includes a series of densely connected layers with ReLU activation functions and dropout layers. The final layer uses the sigmoid activation function, which is typical for autoencoders to ensure output values are in the range [0, 1].

In the **call** method, you define the forward pass of the autoencoder. It takes **inputs** as the input data and passes it through the **encoder** to obtain the encoded representation (**encoded**) and then through the **decoder** to reconstruct the original input (**decoded**).

**model = AutoEncoder(output\_units=x\_train\_scaled.shape[1])**: You create an instance of your **AutoEncoder** class, specifying the **output\_units** based on the shape of your input data. This is a common practice to ensure the autoencoder can reconstruct the input data.

**model.compile(loss='msle', metrics=['mse'], optimizer='adam')**: You compile the model. You set the loss function to 'msle', which stands for Mean Squared Logarithmic Error, and specify that you want to track the mean squared error ('mse') as a metric. The optimizer is set to 'adam', which is a commonly used optimization algorithm.

**epochs = 20**: You set the number of training epochs to 20. This determines how many times the model will be trained on the entire training dataset.

**history = model.fit(x\_train\_scaled, x\_train\_scaled, epochs=epochs, batch\_size=512, validation\_data=(x\_test\_scaled, x\_test\_scaled))**: This line starts the training process. You provide the scaled training data (**x\_train\_scaled**) as both the input and target for reconstruction. The **batch\_size** is set to 512, which means the data is divided into batches of 512 samples during training. The **validation\_data** parameter is set to the scaled testing data (**x\_test\_scaled**) to monitor the model's performance during training.

**find\_threshold(model, x\_train\_scaled)**: This function calculates a threshold for anomaly detection.

* + **reconstructions = model.predict(x\_train\_scaled)**: The autoencoder model is used to reconstruct the input data, and the resulting reconstructions are stored in the **reconstructions** variable.
  + **reconstruction\_errors = tf.keras.losses.msle(reconstructions, x\_train\_scaled)**: Mean Squared Logarithmic Error (MSLE) is calculated for each data point by comparing the reconstructions to the original input data.
  + **threshold = np.mean(reconstruction\_errors.numpy()) + np.std(reconstruction\_errors.numpy())**: The threshold is set as the mean of the reconstruction errors plus one standard deviation. This is a common approach for anomaly detection, where data points with reconstruction errors significantly above this threshold are considered anomalies.

**get\_predictions(model, x\_test\_scaled, threshold)**: This function makes predictions on the testing data using the calculated threshold.

* + **predictions = model.predict(x\_test\_scaled)**: The model is used to make predictions on the testing data.
  + **errors = tf.keras.losses.msle(predictions, x\_test\_scaled)**: MSLE is calculated for the testing data reconstructions.
  + **anomaly\_mask = pd.Series(errors) > threshold**: A boolean mask is created where **True** indicates that a data point is an anomaly (its reconstruction error is greater than the threshold).
  + **preds = anomaly\_mask.map(lambda x: 0.0 if x == True else 1.0)**: The boolean mask is mapped to binary predictions, where **0.0** indicates an anomaly and **1.0** indicates normal data.

**threshold = find\_threshold(model, x\_train\_scaled)**: You call the **find\_threshold** function to calculate the threshold using the training data.

**print(f"Threshold: {threshold}")**: You print the calculated threshold.

**predictions = get\_predictions(model, x\_test\_scaled, threshold)**: You call the **get\_predictions** function with the autoencoder model, scaled testing data (**x\_test\_scaled**), and the calculated threshold. This function returns binary predictions, where 0 indicates anomalies and 1 indicates normal data.

**accuracy\_score(predictions, y\_test)**: You use the **accuracy\_score** function from scikit-learn to evaluate the accuracy of the anomaly detection. The **predictions** are compared to the actual labels from **y\_test**. This will give you an accuracy score, which represents the proportion of correctly classified anomalies and normal data.

**Ass 5**

**from keras.preprocessing import text**: This statement imports modules related to text preprocessing in Keras. These modules provide tools for text tokenization, sequence padding, and other text-related tasks.

**from keras.utils import np\_utils**: This imports the **np\_utils** module from Keras' **utils**. **np\_utils** is used for various utilities, including one-hot encoding of class labels.

**from keras.preprocessing import sequence**: This imports the **sequence** module from Keras' **preprocessing**. The **sequence** module is used for sequence data preprocessing, including padding sequences to a fixed length

.

**from keras.utils import pad\_sequences**: This imports the **pad\_sequences** function from Keras' **utils**. **pad\_sequences** is often used to pad sequences to a specified length.

**import numpy as np**: You import the NumPy library and alias it as **np**. NumPy is a popular library for numerical and array operations in Python.

**import pandas as pd**: You import the Pandas library and alias it as **pd**. Pandas is a powerful library for data manipulation and analysis, particularly for structured data in tabular form.

**dl\_data = data.split()**

In this code, you've taken a text data containing multiple sentences and used the **split()** method to split it into a list of words. Each word in the text has been separated into individual elements in the list. The default delimiter for the **split()** method is whitespace (space or newline).

Now, **dl\_data** contains a list of words, where each word is an element in the list. You can further process and analyze this data as needed for your specific tasks, such as natural language processing or text analysis.

**tokenizer = text.Tokenizer()**: You create a **Tokenizer** instance from Keras. The **Tokenizer** is used to tokenize text data into words.

**tokenizer.fit\_on\_texts(dl\_data)**: You fit the **Tokenizer** on your text data, **dl\_data**, which means it processes the text and builds a vocabulary based on the words found in the data.

**word2id = tokenizer.word\_index**: You obtain a dictionary that maps words to their corresponding unique integer IDs. This is the word-to-ID mapping.

**word2id['PAD'] = 0**: You add a special token 'PAD' with an ID of 0 to represent padding. This is commonly used in sequence processing tasks.

**id2word = {v:k for k, v in word2id.items()}**: You create a dictionary that maps integer IDs to their corresponding words. This is the reverse mapping of word-to-ID.

**wids = [[word2id[w] for w in text.text\_to\_word\_sequence(doc)] for doc in dl\_data]**: You convert the text data **dl\_data** into a list of lists of integer IDs, where each inner list represents a document or sentence. This step replaces words with their corresponding integer IDs.

**vocab\_size = len(word2id)**: You determine the vocabulary size, which is the total number of unique words in your text data.

**embed\_size = 100**: You set the embedding size to 100. This is the dimensionality of the word embeddings you'll create.

**window\_size = 2**: You define a window size, which is often used in the context of word embeddings (e.g., skip-gram or CBOW models). It specifies the context size for training word embeddings.

**print('Vocabulary Size:', vocab\_size)**: You print the vocabulary size, which tells you how many unique words are in your data.

**print('Vocabulary Sample:', list(word2id.items())[:10])**: You print a sample of the word-to-ID mapping to inspect the first 10 entries in the vocabulary.

**context\_length = window\_size \* 2**: You calculate the total context length by multiplying the **window\_size** by 2. This determines the number of words in the context around the target word.

**for words in corpus:**: You iterate through the input **corpus**, where each element represents a list of words in a sentence or document.

**sentence\_length = len(words)**: You calculate the length of the current sentence in terms of the number of words.

**for index, word in enumerate(words):**: Within each sentence, you iterate through the words, and for each word, you generate context-word pairs.

**context\_words = []** and **label\_word = []**: You initialize lists to store the context words and the label word (target word) for the current word in the sentence.

**start = index - window\_size** and **end = index + window\_size + 1**: You determine the starting and ending indices for the context window around the current word. This window is determined by the **window\_size**.

**context\_words.append([...])**: You populate the **context\_words** list with the words within the context window. The list comprehension iterates over the indices within the window and extracts the corresponding words, ensuring that the index is within the valid range.

**label\_word.append(word)**: You add the current word as the label word.

**x = pad\_sequences(context\_words, maxlen=context\_length)**: You use **pad\_sequences** to ensure that all context windows have the same length. This is necessary for training neural network models.

**y = np\_utils.to\_categorical(label\_word, vocab\_size)**: You convert the label word into a one-hot encoded format. This is a common representation for training neural networks in NLP tasks.

**yield (x, y)**: Instead of returning the context-word pairs, you yield them as a generator, allowing you to iterate through them one at a time.

**import keras.backend as K**: You import the Keras backend to access Keras backend operations.

**from keras.models import Sequential**: You import the **Sequential** class from Keras, which is used to define a sequential neural network model.

**from keras.layers import Dense, Embedding, Lambda**: You import the necessary layers for building the neural network, including **Dense** for the output layer, **Embedding** for word embeddings, and **Lambda** for custom layer operations.

**cbow = Sequential()**: You create a **Sequential** model called **cbow** to build your CBOW model.

**cbow.add(Embedding(input\_dim=vocab\_size, output\_dim=embed\_size, input\_length=window\_size\*2)**: You add an embedding layer to the model. This layer is responsible for mapping words to dense vector representations. **input\_dim** specifies the vocabulary size, **output\_dim** is the dimensionality of the word embeddings (previously set to 100), and **input\_length** indicates the size of the input context window.

**cbow.add(Lambda(lambda x: K.mean(x, axis=1), output\_shape=(embed\_size,)))**: You add a Lambda layer that computes the mean of the word embeddings in the context window. This step aggregates the context word vectors into a single vector, which is then used as input for the final classification layer.

**cbow.add(Dense(vocab\_size, activation='softmax'))**: You add a fully connected dense layer with a softmax activation function. This layer is responsible for predicting the target word within the vocabulary.

**cbow.compile(loss='categorical\_crossentropy', optimizer='rmsprop')**: You compile the model. The loss function is set to 'categorical\_crossentropy', which is appropriate for multi-class classification tasks like predicting the next word in a sentence. The optimizer is 'rmsprop', a popular choice for training neural networks.

**print(cbow.summary())**: You print the summary of the CBOW model, which provides information about the model's architecture, layer configurations, and the number of parameters.

**for epoch in range(1, 6)**: You have a loop that iterates over a specified number of epochs, in this case, from 1 to 5.

**loss = 0.**: You initialize the loss to 0 for each epoch. This variable will accumulate the total loss during training.

**i = 0**: You initialize a counter **i** to keep track of the number of context-word pairs processed during the epoch.

The inner loop iterates through the generated context-word pairs, which are provided by the **generate\_context\_word\_pairs** function.

* + **i += 1**: You increment the counter for each context-word pair processed.
  + **loss += cbow.train\_on\_batch(x, y)**: You use the **train\_on\_batch** method of the CBOW model to train on the current context-word pair (input **x** and target **y**). The loss from this batch is added to the total loss for the epoch.
  + **if i % 100000 == 0:**: This condition checks if 100,000 context-word pairs have been processed, and if so, it prints a progress update to keep track of training progress.

**weights = cbow.get\_weights()[0]**: You retrieve the weights from the CBOW model, specifically from the first layer (the embedding layer). This layer contains the word embeddings. The **[0]** index is used to access the weights of the first layer.

**weights = weights[1:]**: You exclude the first row of weights (index 0), which corresponds to the 'PAD' token. This is because 'PAD' is often a special token and is not included in the word embeddings for actual words. By removing it, you ensure that the **weights** variable only contains the embeddings for actual words.

**print(weights.shape)**: You print the shape of the **weights** array, which represents the dimensionality of the word embeddings and the size of the vocabulary.

**pd.DataFrame(weights, index=list(id2word.values())[1:]).head()**: You create a Pandas DataFrame from the weights, using the words (from **id2word**) as the index. The **[1:]** slicing ensures that you skip the 'PAD' token. Finally, you use **.head()** to display the first few rows of the DataFrame.

**from sklearn.metrics.pairwise import euclidean\_distances**: You import the **euclidean\_distances** function from scikit-learn. This function is used to compute the pairwise Euclidean distances between rows (word embeddings) in the input matrix.

**distance\_matrix = euclidean\_distances(weights)**: You calculate the Euclidean distances between all pairs of word embeddings in the **weights** matrix. This results in a square matrix where each element **(i, j)** represents the Euclidean distance between the embeddings for word **i** and word **j**.

**print(distance\_matrix.shape)**: You print the shape of the **distance\_matrix**, which should be a square matrix with a size equal to the vocabulary size.

**similar\_words = {search\_term: [id2word[idx] for idx in distance\_matrix[word2id[search\_term] - 1].argsort()[1:6] + 1] for search\_term in ['deep']}**: You create a dictionary **similar\_words** that stores similar words for a given search term.

* + **search\_term** is set to 'deep' in this example.
  + **word2id[search\_term] - 1** is used to obtain the index of the search term in the vocabulary (minus 1 because the index starts at 0).
  + **distance\_matrix[word2id[search\_term] - 1]** retrieves the row of distances for the search term.
  + **argsort()** returns the indices that would sort the distances in ascending order.
  + **[1:6]** selects the top 5 nearest neighbors (excluding the search term itself).
  + **+ 1** is used to adjust the indices since they are 0-based, but word IDs are 1-based.

**ASS 6**

**import matplotlib.pyplot as plt**: You import the **matplotlib** library, which is commonly used for data visualization. The alias **plt** is used to create plots and charts.

**import numpy as np**: You import the NumPy library and alias it as **np**. NumPy is a popular library for numerical and array operations in Python.

**import os**: You import the **os** module, which provides functions for interacting with the operating system, such as file and directory operations.

**import PIL**: This is the Python Imaging Library. It's often used for image processing tasks. Specific functions and modules from PIL may be imported later in your code.

**import tensorflow as tf**: You import the TensorFlow library, a popular deep learning framework. It is commonly used for building and training neural networks.

**from tensorflow import keras**: You import the Keras API from TensorFlow, which provides high-level interfaces for building and training neural networks.

**from tensorflow.keras import layers**: You import the **layers** module from Keras. This module provides various types of layers used to build neural network architectures.

**from tensorflow.python.keras.layers import Dense, Flatten**: You import specific layers from Keras. **Dense** is used for fully connected layers, and **Flatten** is used to flatten the output from convolutional layers.

**from tensorflow.keras.models import Sequential**: You import the **Sequential** model from Keras. **Sequential** is a type of neural network model that allows you to define a linear stack of layers.

**from tensorflow.keras.optimizers import Adam**: You import the Adam optimizer, which is a popular optimization algorithm used for training neural networks. It is part of the Keras optimizers.

**import pathlib**: You import the **pathlib** module, which is part of the Python standard library and is used for working with file paths and directories.

**dataset\_url = "https://storage.googleapis.com/download.tensorflow.org/example\_images/flower\_photos.tgz"**: You define the URL where the flower photos dataset is located. This dataset contains various images of flowers.

**data\_dir = tf.keras.utils.get\_file('flower\_photos', origin=dataset\_url, untar=True)**: This line downloads the dataset from the specified URL and extracts it to a local directory. The **get\_file** function from TensorFlow's Keras utilities is used for this purpose. The function takes the following arguments:

* + **'flower\_photos'**: This is the name of the file or directory to create.
  + **origin=dataset\_url**: This specifies the URL to download the dataset from.
  + **untar=True**: This indicates that the downloaded file is a tarball archive and should be extracted.

**data\_dir = pathlib.Path(data\_dir)**: After downloading and extracting the dataset, you create a **Path** object using **pathlib** to represent the directory where the dataset is stored. This allows you to work with the files and directories in a platform-independent way.

**image\_dataset\_from\_directory** function to create a training dataset from a directory of images. This function is commonly used for creating a **tf.data.Dataset** from image files in a directory.

The code specifies the **img\_height** and **img\_width** variables as 180, sets the **batch\_size** to 32, and then utilizes these variables to create the training dataset.

Here's a breakdown of the code:

* **img\_height** and **img\_width** are defined as 180.
* **batch\_size** is set to 32.
* The **tf.keras.preprocessing.image\_dataset\_from\_directory** function is used to create a dataset from a directory named **data\_dir**.
* **validation\_split** is set to 0.2, indicating that 20% of the data will be used for validation.
* **subset** is specified as "training," indicating that this dataset is meant for training.
* **seed** is set to 123, providing a seed for reproducibility.
* **image\_size** is set to **(img\_height, img\_width)** which is (180, 180), specifying the height and width of the images in the dataset.
* **batch\_size** is set to **batch\_size** variable, which is 32.
* It's using the **tf.keras.preprocessing.image\_dataset\_from\_directory** function again.
* **validation\_split** is set to 0.2, indicating that 20% of the data will be used for validation. This ensures that the validation dataset contains 20% of the images from **data\_dir**.
* **subset** is specified as "validation," indicating that this dataset is intended for validation purposes.
* **seed** is set to 123 for reproducibility, similar to the training dataset.
* **image\_size** is set to **(img\_height, img\_width)**, which is (180, 180), specifying the height and width of the images in the dataset.
* **batch\_size** is set to **batch\_size**, which is 32.

**import matplotlib.pyplot as plt**: This line imports the Matplotlib library, which is commonly used for creating plots and visualizations in Python.

**plt.figure(figsize=(10, 10))**: This creates a new figure with a specified size of 10x10 inches.

**for images, labels in train\_ds.take(1):**: This loop iterates over the **train\_ds** dataset using **train\_ds.take(1)**, which means it takes one batch of data from the dataset. Each batch typically contains a batch of images and their corresponding labels.

**for i in range(6):**: This loop iterates six times, which means it will display six images in a 3x3 grid.

**ax = plt.subplot(3, 3, i + 1)**: This line creates a subplot in the 3x3 grid at position **i + 1**. The **ax** variable allows you to set properties for the individual subplots.

**plt.imshow(images[i].numpy().astype("uint8"))**: This line displays the image from the batch. **images[i]** is the i-th image in the batch. **.numpy()** converts the image tensor to a NumPy array, and **.astype("uint8")** ensures that the data type is **uint8**, which is expected by Matplotlib for displaying images.

**plt.title(class\_names[labels[i]])**: This sets the title for the subplot using the label associated with the image. **class\_names** is assumed to be a list or array containing the class names or labels for the dataset.

**plt.axis("off")**: This line turns off the axis for the current subplot, which is commonly done for image visualizations to remove the axis labels and ticks.

**resnet\_model = Sequential()**: This line initializes a Sequential model. A Sequential model is a linear stack of layers in Keras, where you can easily add layers one by one.

**pretrained\_model= tf.keras.applications.ResNet50(...)**: Here, you're using the ResNet-50 model provided by TensorFlow's **tf.keras.applications** module. You configure the ResNet-50 model as follows:

* + **include\_top=False**: This means that the final fully connected layers (classification layers) of ResNet-50 are not included. You'll add your own classification layers later.
  + **input\_shape=(180, 180, 3)**: Specifies the input shape for the model, which is (180, 180, 3) representing 180x180-pixel color images (RGB).
  + **pooling='avg'**: This specifies global average pooling as the pooling layer, which computes the average of all values in the feature maps.
  + **classes=5**: Sets the number of classes for your specific problem to 5.
  + **weights='imagenet'**: Initializes the model with pre-trained weights from the ImageNet dataset.

**for layer in pretrained\_model.layers:**: This loop iterates through all the layers in the pre-trained ResNet-50 model.

**layer.trainable=False**: It sets the **trainable** attribute of each layer in the pre-trained model to **False**, effectively freezing the weights of the pre-trained layers. This is a common practice when using pre-trained models for transfer learning, as you typically want to fine-tune only the added layers for your specific task.

**resnet\_model.add(pretrained\_model)**: This adds the pre-trained ResNet-50 model (with its layers frozen) to your **resnet\_model**.

**resnet\_model.add(Flatten())**: This adds a Flatten layer, which converts the 3D output from the previous layers into a 1D vector, preparing the data for fully connected layers.

**resnet\_model.add(Dense(512, activation='relu'))**: Adds a fully connected layer with 512 units and ReLU activation. This layer can capture complex patterns in the flattened feature vectors.

**resnet\_model.add(Dense(5, activation='softmax'))**: Adds the final fully connected layer with 5 units (assuming you have 5 classes) and softmax activation. This layer is responsible for the actual classification.

* **epochs = 10**: This sets the number of training epochs to 10, meaning that the model will go through the entire training dataset 10 times during training.
* **history = resnet\_model.fit(...)**: This is where you're fitting the model to the training data and monitoring its performance on the validation data.
* **train\_ds** is used as the training dataset.
* **val\_ds** is used as the validation dataset, allowing you to monitor how well the model generalizes to data it hasn't seen during training.

**epochs=epochs** specifies the number of training epochs, which is set to 10 as defined earlier.

**import cv2**: This line imports the OpenCV library, which is commonly used for image processing and computer vision tasks in Python.

**image = cv2.imread(str(roses[0]))**: This line reads an image from the file path specified by **roses[0]**. **roses** is assumed to be a list or collection of file paths to images. The **str()** function is used to ensure that the path is treated as a string.

**image\_resized = cv2.resize(image, (img\_height, img\_width))**: This line resizes the image to the dimensions specified by **img\_height** and **img\_width**. The resulting image is stored in the **image\_resized** variable.

**image = np.expand\_dims(image\_resized, axis=0)**: This line expands the dimensions of the resized image to create a batch with a single image. The **axis=0** argument adds a new dimension at the beginning, effectively turning the image into a 4D array with shape **(1, img\_height, img\_width, 3)**. This is often done when you want to feed a single image to a deep learning model, as many models expect batched input data.

**print(image.shape)**: This line prints the shape of the **image** array. It will show you the dimensions of the image, which should be **(1, img\_height, img\_width, 3)** if the expansion was successful.

**pred = resnet\_model.predict(image)**: This line uses the **resnet\_model** to make predictions on the input image, which has been preprocessed and expanded into the appropriate shape. The **predict** method of a Keras model returns the predicted class probabilities for the input data.

**print(pred)**: This line prints the predicted class probabilities to the console.

**output\_class = class\_names[np.argmax(pred)]**: This line uses **np.argmax(pred)** to find the index of the class with the highest predicted probability in the **pred** array. Then, it uses this index to access the corresponding class label from the **class\_names** list. The **output\_class** variable will store the predicted class label.

**print("The predicted class is", output\_class)**: This line prints the predicted class label to the console.