```python
import tensorflow as tf
from keras.applications.vgg16 import VGG16
from keras.applications.resnet import ResNet50
import os
import numpy as np
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import pandas as pd
import matplotlib.pyplot as plt
```

```
WARNING:tensorflow:From C:\Users\Faiza Anan Noor\anaconda3\Lib\site-packages\keras\src\losses.py:2976: The name tf.losses.sparse_softmax_cross_entropy is deprecated. Please use tf.compat.v1.losses.sparse_softmax_cross_entropy instead.
```

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense, Dropout
from tensorflow.keras.applications import VGG16
from tensorflow.keras.optimizers import Adam

from tensorflow.keras.models import load_model
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
from scipy import interp
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
```

Here prepare the folder if does not exist

```python
# you need the current working directory NB: works both windows and linux
current_working_directory = os.getcwd()
current_working_directory = os.path.dirname(current_working_directory)

if not os.path.exists(f"{current_working_directory}/Datasets"):
    os.makedirs(f"{current_working_directory}/Datasets")

print(f"[DATASET] PUT THE DATASET here: {current_working_directory}/Datasets")
```

```
[DATASET] PUT THE DATASET here: C:\Users\Faiza Anan Noor\Computer Vision UTU/Datasets
```

```
# get the directory where I want to download the dataset
path_of_dataset = os.path.join(*['..', current_working_directory,
'Datasets', 'Most_Stolen_Cars'])
print(f"[DIR] The directory of the current dataset is
{path_of_dataset}")
```

```
[DIR] The directory of the current dataset is C:\Users\Faiza Anan
Noor\Computer Vision UTU\Datasets\Most_Stolen_Cars
```

Data prep

```
# here let s do some functions that we can re-use also for other
assignment
def load_the_data_and_the_labels(data_set_path: str, target_size:
tuple or None = None):
    """
    This function help you to load the data dynamically
    :param data_set_path: (str) put the path created in the previous
cell (is the dataset path)
    :param target_size: (tuple) the desired size of the images
    :return:
        - array of images
        - array with labels
        - list of labels name (this is used for better visualization)
    """
    try:
        dataset, labels, name_of_the_labels = list(), list(), list()
        # let s loop here and we try to discover how many class we
have
        for class_number, class_name in
enumerate(os.listdir(data_set_path)):
            full_path_the_data = os.path.join(*[data_set_path,
class_name])
            print(f"[WALK] I am walking into {full_path_the_data}")

            # add the list to nam _list
            name_of_the_labels.append(class_name)

            for single_image in os.listdir(f"{full_path_the_data}"):
                full_path_to_image =
os.path.join(*[full_path_the_data, single_image])

                # add the class number
                labels.append(class_number)

                if target_size is None:
                    # let s load the image
                    image =
tf.keras.utils.load_img(full_path_to_image)
                else:
```

```
                    image = 
tf.keras.utils.load_img(full_path_to_image, target_size=target_size)

                # transform PIL object in image
                image = tf.keras.utils.img_to_array(image)

                # add the image to the ds list
                dataset.append(image)

        return np.array(dataset, dtype='uint8'), np.array(labels, 
dtype='int'), name_of_the_labels
    except Exception as ex:
        print(f"[EXCEPTION] load the data and the labels throws 
exceptions {ex}")
```

Load the data

```
a. Target size: (112, 112, 3)
b. if for some reason your pc crash saying Out of Memory reduce half 
the target size

dataset, labels, 
label_names=load_the_data_and_the_labels(path_of_dataset, ((112, 112, 
3)))

[WALK] I am walking into C:\Users\Faiza Anan Noor\Computer Vision UTU\
Datasets\Most_Stolen_Cars\chevrolet_impala_2008
[WALK] I am walking into C:\Users\Faiza Anan Noor\Computer Vision UTU\
Datasets\Most_Stolen_Cars\chevrolet_silverado_2004
[WALK] I am walking into C:\Users\Faiza Anan Noor\Computer Vision UTU\
Datasets\Most_Stolen_Cars\dodge_ram_2001
[WALK] I am walking into C:\Users\Faiza Anan Noor\Computer Vision UTU\
Datasets\Most_Stolen_Cars\ford_f150_2006
[WALK] I am walking into C:\Users\Faiza Anan Noor\Computer Vision UTU\
Datasets\Most_Stolen_Cars\gmc_sierra_2012
[WALK] I am walking into C:\Users\Faiza Anan Noor\Computer Vision UTU\
Datasets\Most_Stolen_Cars\honda_accord_1997
[WALK] I am walking into C:\Users\Faiza Anan Noor\Computer Vision UTU\
Datasets\Most_Stolen_Cars\honda_civic_1998
[WALK] I am walking into C:\Users\Faiza Anan Noor\Computer Vision UTU\
Datasets\Most_Stolen_Cars\nissan_altima_2014
[WALK] I am walking into C:\Users\Faiza Anan Noor\Computer Vision UTU\
Datasets\Most_Stolen_Cars\toyota_camry_2014
[WALK] I am walking into C:\Users\Faiza Anan Noor\Computer Vision UTU\
Datasets\Most_Stolen_Cars\toyota_corolla_2013
```

normalize the data here

```
# do it here
dataset=dataset/255.0
```

Convert the data to one hot encoding (use the sklearn function)

```
# here we have to one hot encode the labes
def make_the_one_hot_encoding(labels_to_transform):
    try:
        enc = OneHotEncoder(handle_unknown='ignore')
        # this is a trick to figure the array as 2d array instead of
list
        temp = np.reshape(labels_to_transform, (-1, 1))
        labels_to_transform = enc.fit_transform(temp).toarray()
        print(f'[ONE HOT ENCODING] Labels are one-hot-encoded:
{(labels_to_transform.sum(axis=1) -
np.ones(labels_to_transform.shape[0])).sum() == 0}')
        return labels_to_transform
    except Exception as ex:
        print(f"[EXCEPTION] Make the one hot encoding throws exception
{ex}")

# do it here
one_hot_labels = make_the_one_hot_encoding(labels)
one_hot_labels
```

```
[ONE HOT ENCODING] Labels are one-hot-encoded: True
```

```
array([[1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 0., 1.]])
```

Every category in the dataset is represented as a binary vector in one-hot encoding, with each vector element corresponding to a distinct category.

split the data in train set and test set

```
a. use 0.3 as split factor
```

We divide the data into train and test labels and targets using the train_test_split function defining the test size as 30% of the whole data

```
X_train, X_test, y_train, y_test = train_test_split(dataset,
one_hot_labels, test_size=0.3, random_state=42)

# Print the shape of the training and testing sets for verification
print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
```

```
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
```

```
X_train shape: (4137, 112, 112, 3)
X_test shape: (1774, 112, 112, 3)
y_train shape: (4137, 10)
y_test shape: (1774, 10)
```

Create a CNN with the following characteristics

```
    a. Input layer
    b. As base model use VGG16:
        i. Weights: imagenet
        ii. Include_top: False
        iii. Input_shape the target shape described in point 1.
    c. Add a flatten layer
    d. Add a Dense layer with 512 units and a dropout layer with 0.2
unit.
    e. Add a Dense layer with 256 units and a dropout layer with 0.2
unit.
    f. Add the final classifier with the correct number of units and
the suitable activation.
```

alt text

In the following code snippet, we create the CNN model according to the instructions above. We defined the input shape as (112, 112, 3) as target size of the images use (112, 112) and has 3 color channels. The following custom convolutional neural network (CNN) called "custom_cnn" architecture is built upon the VGG16 pre-trained model, excluding its top classifier. Specific layers, including 'block5_conv2', 'block5_conv3', and 'block5_pool', are selected for fine-tuning, while others are frozen to retain pre-trained weights. The model incorporates a Flatten layer to transform the output of the base model into a vector, followed by dense layers with rectified linear unit (ReLU) activation functions and dropout regularization for feature extraction and regularization, respectively. These additional layers aim to capture more abstract features from the flattened output before the final softmax classifier, which predicts probabilities across the specified number of classes, enabling effective classification. This architecture balances feature extraction from pre-trained layers with task-specific learning through fine-tuning and newly added layers, optimizing performance for the given classification task.

```
# do it here

from tensorflow.keras.models import Model
num_classes=10

# Define input shape and number of classes
input_shape = (112, 112, 3)  # Target shape described in point 1
num_classes = 10  # Assuming 10 classes for classification
```

```
# Load the VGG16 model with imagenet weights and without the top
classifier
base_model = VGG16(weights='imagenet', include_top=False,
input_shape=input_shape)

# Set the specified layers as trainable
for layer in base_model.layers:
    if layer.name in ['block5_conv2', 'block5_conv3', 'block5_pool']:
        layer.trainable = True
    else:
        layer.trainable = False

# Flatten the output of the base model
x = Flatten()(base_model.output)

# Add a Dense layer with 512 units and dropout
x = Dense(512, activation='relu')(x)
x = Dropout(0.2)(x)

# Add a Dense layer with 256 units and dropout
x = Dense(256, activation='relu')(x)
x = Dropout(0.2)(x)

# Add the final classifier with the correct number of units and
softmax activation
predictions = Dense(num_classes, activation='softmax')(x)  # Assuming
num_classes is defined

# Combine the base model and the classifier
custom_cnn = Model(inputs=base_model.input, outputs=predictions)
custom_cnn
```

```
WARNING:tensorflow:From C:\Users\Faiza Anan Noor\anaconda3\Lib\site-
packages\keras\src\backend.py:1398: The name
tf.executing_eagerly_outside_functions is deprecated. Please use
tf.compat.v1.executing_eagerly_outside_functions instead.

WARNING:tensorflow:From C:\Users\Faiza Anan Noor\anaconda3\Lib\site-
packages\keras\src\layers\pooling\max_pooling2d.py:161: The name
tf.nn.max_pool is deprecated. Please use tf.nn.max_pool2d instead.
```

```
<keras.src.engine.functional.Functional at 0x230db050190>
```

```
# Define input shape and number of classes
input_shape = (112, 112, 3)  # Target shape described in point 1
num_classes = 10  # Assuming 10 classes for classification

# Create the model
```

```python
# Print model summary
custom_cnn.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_1 (InputLayer)        [(None, 112, 112, 3)]     0         
                                                                 
 block1_conv1 (Conv2D)       (None, 112, 112, 64)      1792      
                                                                 
 block1_conv2 (Conv2D)       (None, 112, 112, 64)      36928     
                                                                 
 block1_pool (MaxPooling2D)  (None, 56, 56, 64)        0         
                                                                 
 block2_conv1 (Conv2D)       (None, 56, 56, 128)       73856     
                                                                 
 block2_conv2 (Conv2D)       (None, 56, 56, 128)       147584    
                                                                 
 block2_pool (MaxPooling2D)  (None, 28, 28, 128)       0         
                                                                 
 block3_conv1 (Conv2D)       (None, 28, 28, 256)       295168    
                                                                 
 block3_conv2 (Conv2D)       (None, 28, 28, 256)       590080    
                                                                 
 block3_conv3 (Conv2D)       (None, 28, 28, 256)       590080    
                                                                 
 block3_pool (MaxPooling2D)  (None, 14, 14, 256)       0         
                                                                 
 block4_conv1 (Conv2D)       (None, 14, 14, 512)       1180160   
                                                                 
 block4_conv2 (Conv2D)       (None, 14, 14, 512)       2359808   
                                                                 
 block4_conv3 (Conv2D)       (None, 14, 14, 512)       2359808   
                                                                 
 block4_pool (MaxPooling2D)  (None, 7, 7, 512)         0         
                                                                 
 block5_conv1 (Conv2D)       (None, 7, 7, 512)         2359808   
                                                                 
 block5_conv2 (Conv2D)       (None, 7, 7, 512)         2359808   
                                                                 
 block5_conv3 (Conv2D)       (None, 7, 7, 512)         2359808   
                                                                 
 block5_pool (MaxPooling2D)  (None, 3, 3, 512)         0         
                                                                 
 flatten (Flatten)           (None, 4608)              0         
                                                                 
 dense (Dense)               (None, 512)               2359808   
                                                                 
 dropout (Dropout)           (None, 512)               0         
```

```
 dense_1 (Dense)              (None, 256)               131328

 dropout_1 (Dropout)          (None, 256)               0

 dense_2 (Dense)              (None, 10)                2570

=================================================================
Total params: 17208394 (65.64 MB)
Trainable params: 7213322 (27.52 MB)
Non-trainable params: 9995072 (38.13 MB)
_________________________________________________________________
```

Set the layer block5_conv2, block5_conv3, block5_pool trainable

```
Important: you can make a function when you create a CNN within the
option of make layers trainable or not is up to you!

#do it here

# Verify the layers' trainable status
for layer in custom_cnn.layers:
    print(f'{layer.name}: Trainable={layer.trainable}')

input_1: Trainable=False
block1_conv1: Trainable=False
block1_conv2: Trainable=False
block1_pool: Trainable=False
block2_conv1: Trainable=False
block2_conv2: Trainable=False
block2_pool: Trainable=False
block3_conv1: Trainable=False
block3_conv2: Trainable=False
block3_conv3: Trainable=False
block3_pool: Trainable=False
block4_conv1: Trainable=False
block4_conv2: Trainable=False
block4_conv3: Trainable=False
block4_pool: Trainable=False
block5_conv1: Trainable=False
block5_conv2: Trainable=True
block5_conv3: Trainable=True
block5_pool: Trainable=True
flatten: Trainable=True
dense: Trainable=True
dropout: Trainable=True
dense_1: Trainable=True
dropout_1: Trainable=True
dense_2: Trainable=True
```

```
checkpoint_path = "training_first_model/cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

# Create a callback that saves the model's weights
cp_callback =
tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,

save_weights_only=True,

                                                 verbose=1)
```

Train the model

```
a. set the batch size 32 (if your PC go Out of memory lower this
number half)
b. set epochs to 15
```

We also used Callback to periodically save the Keras model or model weights. The ModelCheckpoint callback is utilized in combination with model.fit() training to save a model or weights at regular intervals (in a checkpoint file). This allows the model or weights to be loaded at a later time to resume training from the saved state. We used the CategoricalCrossentropy loss function to train, Adam optimizer, and set our metric as accuracy. and we also set epoch =15 and batch size 32.

```
# do it here



# Compile the model
custom_cnn.compile(
    loss =tf.keras.losses.CategoricalCrossentropy(from_logits = True),
    optimizer = "Adam",
    metrics =["accuracy"],
    )

# Train the model
history = custom_cnn.fit(X_train, y_train,
                         batch_size=32,
                         epochs=15,
                           callbacks=[cp_callback],
                         validation_data=(X_test, y_test)
                        )

# Evaluate the model
loss, accuracy = custom_cnn.evaluate(X_test, y_test)
print(f'Test Loss: {loss}, Test Accuracy: {accuracy}')
```

```
WARNING:tensorflow:From C:\Users\Faiza Anan Noor\anaconda3\Lib\site-
packages\keras\src\optimizers\__init__.py:309: The name
tf.train.Optimizer is deprecated. Please use
tf.compat.v1.train.Optimizer instead.

Epoch 1/15
WARNING:tensorflow:From C:\Users\Faiza Anan Noor\anaconda3\Lib\site-
packages\keras\src\utils\tf_utils.py:492: The name
tf.ragged.RaggedTensorValue is deprecated. Please use
tf.compat.v1.ragged.RaggedTensorValue instead.


C:\Users\Faiza Anan Noor\anaconda3\Lib\site-packages\keras\src\
backend.py:5575: UserWarning: "`categorical_crossentropy` received
`from_logits=True`, but the `output` argument was produced by a
Softmax activation and thus does not represent logits. Was this
intended?
  output, from_logits = _get_logits(

WARNING:tensorflow:From C:\Users\Faiza Anan Noor\anaconda3\Lib\site-
packages\keras\src\engine\base_layer_utils.py:384: The name
tf.executing_eagerly_outside_functions is deprecated. Please use
tf.compat.v1.executing_eagerly_outside_functions instead.

130/130 [==============================] - ETA: 0s - loss: 1.8571 -
accuracy: 0.2956
Epoch 1: saving model to training_first_model\cp.ckpt
130/130 [==============================] - 237s 2s/step - loss: 1.8571
- accuracy: 0.2956 - val_loss: 1.5378 - val_accuracy: 0.3844
Epoch 2/15
130/130 [==============================] - ETA: 0s - loss: 1.4573 -
accuracy: 0.4008
Epoch 2: saving model to training_first_model\cp.ckpt
130/130 [==============================] - 248s 2s/step - loss: 1.4573
- accuracy: 0.4008 - val_loss: 1.4414 - val_accuracy: 0.4053
Epoch 3/15
130/130 [==============================] - ETA: 0s - loss: 1.2454 -
accuracy: 0.4733
Epoch 3: saving model to training_first_model\cp.ckpt
130/130 [==============================] - 263s 2s/step - loss: 1.2454
- accuracy: 0.4733 - val_loss: 1.1861 - val_accuracy: 0.5395
Epoch 4/15
130/130 [==============================] - ETA: 0s - loss: 1.0223 -
accuracy: 0.5584
Epoch 4: saving model to training_first_model\cp.ckpt
130/130 [==============================] - 238s 2s/step - loss: 1.0223
- accuracy: 0.5584 - val_loss: 1.1281 - val_accuracy: 0.5378
Epoch 5/15
130/130 [==============================] - ETA: 0s - loss: 0.8984 -
accuracy: 0.6198
```

```
Epoch 5: saving model to training_first_model\cp.ckpt
130/130 [==============================] - 221s 2s/step - loss: 0.8984
- accuracy: 0.6198 - val_loss: 1.1737 - val_accuracy: 0.5631
Epoch 6/15
130/130 [==============================] - ETA: 0s - loss: 0.7534 -
accuracy: 0.6749
Epoch 6: saving model to training_first_model\cp.ckpt
130/130 [==============================] - 223s 2s/step - loss: 0.7534
- accuracy: 0.6749 - val_loss: 1.1476 - val_accuracy: 0.6043
Epoch 7/15
130/130 [==============================] - ETA: 0s - loss: 0.6555 -
accuracy: 0.7276
Epoch 7: saving model to training_first_model\cp.ckpt
130/130 [==============================] - 230s 2s/step - loss: 0.6555
- accuracy: 0.7276 - val_loss: 1.1113 - val_accuracy: 0.6167
Epoch 8/15
130/130 [==============================] - ETA: 0s - loss: 0.5221 -
accuracy: 0.7880
Epoch 8: saving model to training_first_model\cp.ckpt
130/130 [==============================] - 224s 2s/step - loss: 0.5221
- accuracy: 0.7880 - val_loss: 1.3000 - val_accuracy: 0.5992
Epoch 9/15
130/130 [==============================] - ETA: 0s - loss: 0.4672 -
accuracy: 0.8243
Epoch 9: saving model to training_first_model\cp.ckpt
130/130 [==============================] - 245s 2s/step - loss: 0.4672
- accuracy: 0.8243 - val_loss: 1.3072 - val_accuracy: 0.6240
Epoch 10/15
130/130 [==============================] - ETA: 0s - loss: 0.4220 -
accuracy: 0.8378
Epoch 10: saving model to training_first_model\cp.ckpt
130/130 [==============================] - 247s 2s/step - loss: 0.4220
- accuracy: 0.8378 - val_loss: 1.2359 - val_accuracy: 0.6539
Epoch 11/15
130/130 [==============================] - ETA: 0s - loss: 0.2843 -
accuracy: 0.8973
Epoch 11: saving model to training_first_model\cp.ckpt
130/130 [==============================] - 223s 2s/step - loss: 0.2843
- accuracy: 0.8973 - val_loss: 1.3743 - val_accuracy: 0.6319
Epoch 12/15
130/130 [==============================] - ETA: 0s - loss: 0.2785 -
accuracy: 0.9028
Epoch 12: saving model to training_first_model\cp.ckpt
130/130 [==============================] - 252s 2s/step - loss: 0.2785
- accuracy: 0.9028 - val_loss: 1.3110 - val_accuracy: 0.6697
Epoch 13/15
130/130 [==============================] - ETA: 0s - loss: 0.1592 -
accuracy: 0.9437
Epoch 13: saving model to training_first_model\cp.ckpt
```

```
130/130 [==============================] - 228s 2s/step - loss: 0.1592
- accuracy: 0.9437 - val_loss: 1.7679 - val_accuracy: 0.6387
Epoch 14/15
130/130 [==============================] - ETA: 0s - loss: 0.1778 -
accuracy: 0.9398
Epoch 14: saving model to training_first_model\cp.ckpt
130/130 [==============================] - 253s 2s/step - loss: 0.1778
- accuracy: 0.9398 - val_loss: 1.6196 - val_accuracy: 0.6516
Epoch 15/15
130/130 [==============================] - ETA: 0s - loss: 0.2116 -
accuracy: 0.9335
Epoch 15: saving model to training_first_model\cp.ckpt
130/130 [==============================] - 255s 2s/step - loss: 0.2116
- accuracy: 0.9335 - val_loss: 1.6085 - val_accuracy: 0.6595
56/56 [==============================] - 62s 1s/step - loss: 1.6085 -
accuracy: 0.6595
Test Loss: 1.608549952507019, Test Accuracy: 0.6595264673233032
```

I evaluated the model using model.evaluate passing in our test labels and targets. And then from that variable containing the evaluation results, we print losses, accuracy of our model on the test set

```
# Save the entire model as a `.keras` zip archive.
custom_cnn.save('First_model.keras')
```

evaluate the model and record the accuracy score.

```
custom_cnn= tf.keras.models.load_model('First_model.keras')

# do it here

# Evaluate the model on the test data
loss, accuracy = custom_cnn.evaluate(X_test, y_test)

# Print the test accuracy score
print(f'Test Accuracy: {accuracy}')

56/56 [==============================] - 62s 981ms/step - loss: 1.6085
- accuracy: 0.6595
Test Accuracy: 0.6595264673233032
```

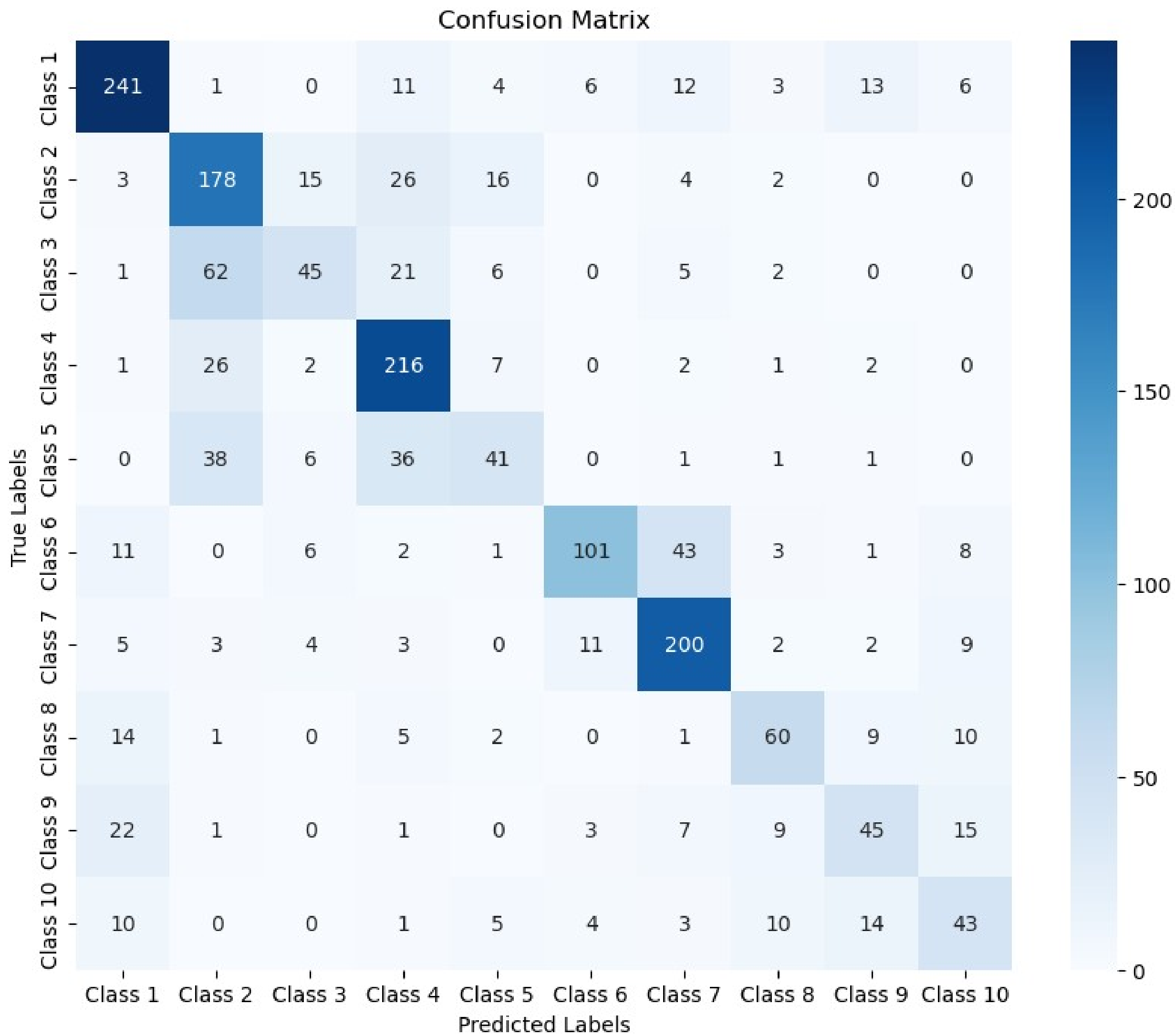
For our first model, ie, "custom_cnn" model, the accuracy came out to be 65%. We can see that almost 65% of the time, it gets correct results.

Since we get softmax probabilities as outputs, we converted the predicted probabilities into class labels taking their argmax of the predictions and test classes.

```python
# Get the model's predictions on the test data
predictions = custom_cnn.predict(X_test)

# Convert the predicted probabilities to class labels
predicted_labels = np.argmax(predictions, axis=1)

# Convert one-hot encoded true labels to class labels
true_labels = np.argmax(y_test, axis=1)

# Print the true and predicted labels
print("True Labels:", true_labels)
print("Predicted Labels:", predicted_labels)
```

```
56/56 [==============================] - 65s 1s/step
True Labels: [8 2 6 ... 0 5 0]
Predicted Labels: [8 1 6 ... 0 5 0]
```

```python
# Define class labels
class_labels = ["Class 1", "Class 2", "Class 3", "Class 4", "Class 5",
                "Class 6", "Class 7", "Class 8", "Class 9", "Class
10"]

conf_matrix = confusion_matrix(true_labels, predicted_labels)


# Plot confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues",
xticklabels=class_labels, yticklabels=class_labels)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix")
plt.show()
```



Then we analyzed the confusion matrix for showing true classes for all 10 classes. To cite a few cases, we can see that 241 of class 1 has been corrrectly identified as class 1, 216 of class 4 has been corrrectly identified as class 4, 200 of class 7 has been corrrectly identified as class 7. So the model performs relatively well for these classes.

```
import numpy as np
import matplotlib.pyplot as plt

# X_test contains the test images and y_test contains the true labels

# Make predictions
predictions = custom_cnn.predict(X_test)

# Convert one-hot encoded labels back to categorical labels
true_labels = np.argmax(y_test, axis=1)
predicted_labels = np.argmax(predictions, axis=1)
```

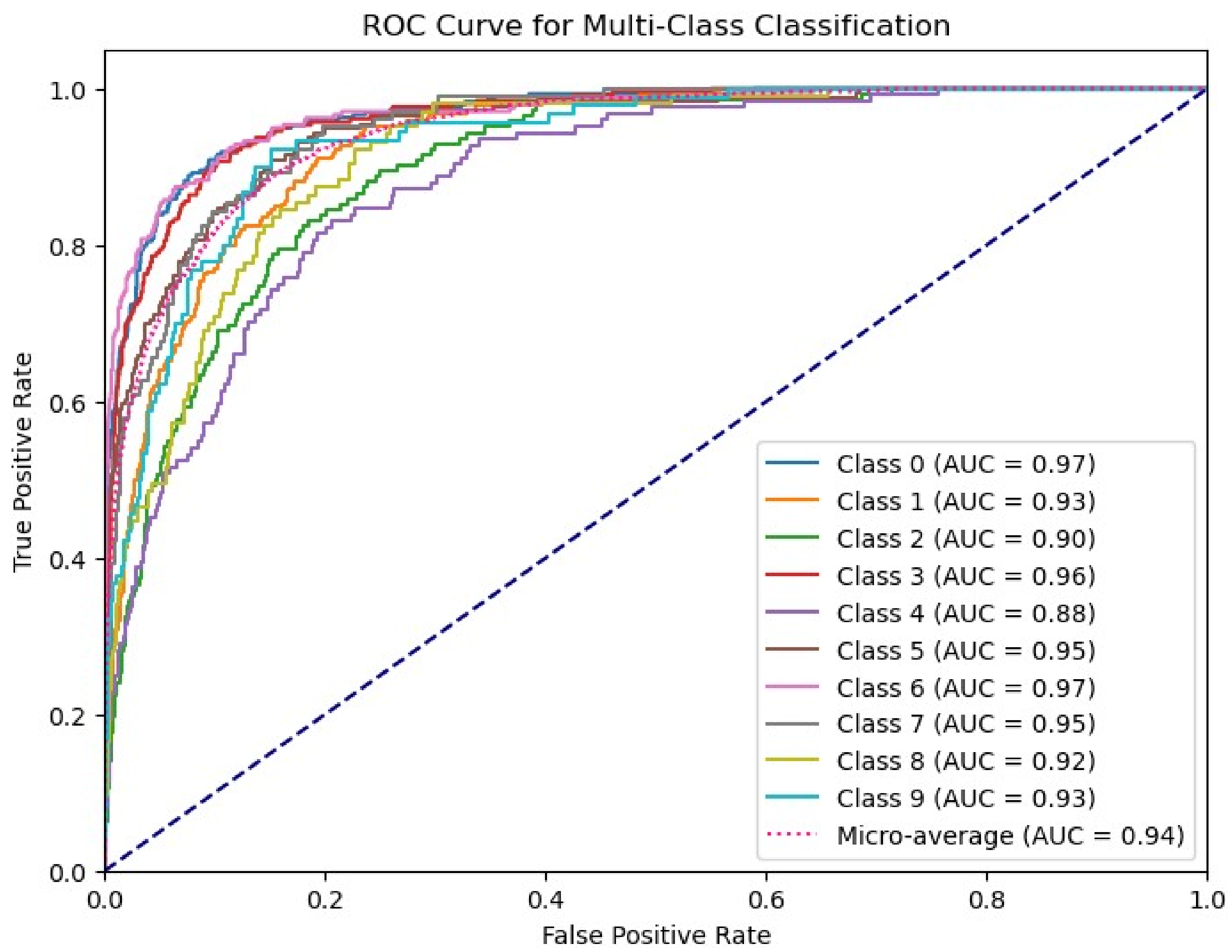
```python
# Visualize some predictions
num_images = 4  # Number of images to visualize
indices = np.random.choice(len(X_test), num_images, replace=False)

plt.figure(figsize=(15, 8))
for i, idx in enumerate(indices):
    plt.subplot(1, num_images, i+1)
    plt.imshow(X_test[idx])
    plt.title(f'True Label: {true_labels[idx]}, Predicted Label:
{predicted_labels[idx]}')
    plt.axis('off')
plt.show()
```

```
56/56 [==============================] - 61s 1s/step
```

Then we tried to show some predictions by plotting a test image against our predicted Label Results. For our 4 cases as depicted above, our model predicted a case with actual label 9 as 9, actual label 3 as 3, and actual label 7 as 7. However, it wrongly identified a class with true label 1 as 4 wrongly.

```python
# Convert true labels to one-hot encoded format
y_test_bin = label_binarize(y_test, classes=np.arange(10))  # Assuming
10 classes

# Compute ROC curve and ROC area for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(10):  # Assuming 10 classes
    fpr[i], tpr[i], _ = roc_curve(y_test_bin[:, i], predictions[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_bin.ravel(),
predictions.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plot ROC curve for each class
plt.figure(figsize=(8, 6))
for i in range(10):  # Assuming 10 classes
```

```
    plt.plot(fpr[i], tpr[i], label=f'Class {i} (AUC =
{roc_auc[i]:.2f})')
plt.plot(fpr["micro"], tpr["micro"], label='Micro-average (AUC =
{0:0.2f})'.format(roc_auc["micro"]), color='deeppink', linestyle=':')
plt.plot([0, 1], [0, 1], color='navy', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for Multi-Class Classification')
plt.legend(loc="lower right")
plt.show()
```



A true positive rate of 1 (all positives are correctly identified) and a false positive rate of 0 (no negatives are wrongly classified) are the benchmarks for flawless classification performance, which is represented by a value of 1. When the model's performance is equal to chance, a value of 0.5 indicates random classification. Higher numbers show stronger discriminating between positive and negative samples. numbers between 0.5 and 1 reflect varied degrees of categorization performance.

With different ROC AUC values of 0.97, 0.93, 0.95, 0.92, 0.88, 0.96 etc for three distinct classes in my instance, it is known that the model does a good job of differentiating between positive and negative instances for these classes; higher values suggest stronger discrimination and good efficiency for the model. For our case, the highest AUC belongs to class 0 and class 6, followed by class 3. The poorest come from class 4 which means that the model is least capable of differentiating this class.

```
from tensorflow.keras.models import load_model

# Load the previously defined model

base_model2 = VGG16(weights='imagenet', include_top=False,
input_shape=input_shape)

# Set the specified layers as trainable
for layer in base_model2.layers:
    layer.trainable = False


# Flatten the output of the base model
x = Flatten()(base_model2.output)

# Add a Dense layer with 512 units and dropout
x = Dense(512, activation='relu')(x)
x = Dropout(0.2)(x)

# Add a Dense layer with 256 units and dropout
x = Dense(256, activation='relu')(x)
x = Dropout(0.2)(x)

# Add the final classifier with the correct number of units and
softmax activation
predictions2 = Dense(num_classes, activation='softmax')(x)  # Assuming
num_classes is defined

# Combine the base model and the classifier
custom_cnn2 = Model(inputs=base_model2.input, outputs=predictions2)
custom_cnn2
```

```
<keras.src.engine.functional.Functional at 0x230daf5d890>
```

This code snippet loads a new custom CNN like previously but now its named "custom_cnn2" and then iterates through all layers of the model, setting their trainable attribute to False, effectively freezing them to prevent further training. This ensures that only the newly added layers (if any) will be trainable, while the pre-trained layers retain their learned representations.

```
for layer in custom_cnn2.layers[:]:
    layer.trainable = False
    if layer.name =='block5_pool':
```

```
        break
#custom_cnn2.summary()
```

Load again the CNN and set all the base model layers to not trainable.

```
# Verify the layers' trainable status
for layer in custom_cnn2.layers:
    print(f'{layer.name}: Trainable={layer.trainable}')

input_2: Trainable=False
block1_conv1: Trainable=False
block1_conv2: Trainable=False
block1_pool: Trainable=False
block2_conv1: Trainable=False
block2_conv2: Trainable=False
block2_pool: Trainable=False
block3_conv1: Trainable=False
block3_conv2: Trainable=False
block3_conv3: Trainable=False
block3_pool: Trainable=False
block4_conv1: Trainable=False
block4_conv2: Trainable=False
block4_conv3: Trainable=False
block4_pool: Trainable=False
block5_conv1: Trainable=False
block5_conv2: Trainable=False
block5_conv3: Trainable=False
block5_pool: Trainable=False
flatten_1: Trainable=True
dense_3: Trainable=True
dropout_2: Trainable=True
dense_4: Trainable=True
dropout_3: Trainable=True
dense_5: Trainable=True
```

Repeat the train and evaluation steps

What happen? Why?

Our second model with all the base layers set untrainable led to a much less accuracy of 62% compared to 65% like our first case. We can also see in upcoming cells that our first model led to a higher result for some classes using the confusion matrix and ROC AUC curve.

```
checkpoint_path2 = "training_second_model/cp.ckpt"
checkpoint_dir2 = os.path.dirname(checkpoint_path2)

# Create a callback that saves the model's weights
cp_callback2 =
tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path2,
```

```
save_weights_only=True,

                                                        verbose=1)
```

We trained and compiled our "cnn_model2" in same settings as our first model "custom_cnn".

```
# here

# Compile the model
custom_cnn2.compile(
    loss =tf.keras.losses.CategoricalCrossentropy(from_logits = True),
    optimizer ="Adam",
    metrics =["accuracy"],
    )

# Train the model
history = custom_cnn2.fit(X_train, y_train,
                         batch_size=32,
                         epochs=15,
                            callbacks=[cp_callback2],
                        validation_data=(X_test, y_test)

                         )

Epoch 1/15
130/130 [==============================] - ETA: 0s - loss: 1.3586 -
accuracy: 0.4786
Epoch 1: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 204s 2s/step - loss: 1.3586
- accuracy: 0.4786 - val_loss: 1.3114 - val_accuracy: 0.5147
Epoch 2/15
130/130 [==============================] - ETA: 0s - loss: 1.1202 -
accuracy: 0.5748
Epoch 2: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 218s 2s/step - loss: 1.1202
- accuracy: 0.5748 - val_loss: 1.1364 - val_accuracy: 0.5800
Epoch 3/15
130/130 [==============================] - ETA: 0s - loss: 0.9483 -
accuracy: 0.6423
Epoch 3: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 270s 2s/step - loss: 0.9483
- accuracy: 0.6423 - val_loss: 1.1084 - val_accuracy: 0.5902
Epoch 4/15
130/130 [==============================] - ETA: 0s - loss: 0.7814 -
accuracy: 0.7143
Epoch 4: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 269s 2s/step - loss: 0.7814
- accuracy: 0.7143 - val_loss: 1.1419 - val_accuracy: 0.5941
Epoch 5/15
```

```
130/130 [==============================] - ETA: 0s - loss: 0.6762 -
accuracy: 0.7489
Epoch 5: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 217s 2s/step - loss: 0.6762
- accuracy: 0.7489 - val_loss: 1.1932 - val_accuracy: 0.5874
Epoch 6/15
130/130 [==============================] - ETA: 0s - loss: 0.5575 -
accuracy: 0.8001
Epoch 6: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 184s 1s/step - loss: 0.5575
- accuracy: 0.8001 - val_loss: 1.3271 - val_accuracy: 0.5603
Epoch 7/15
130/130 [==============================] - ETA: 0s - loss: 0.4484 -
accuracy: 0.8344
Epoch 7: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 185s 1s/step - loss: 0.4484
- accuracy: 0.8344 - val_loss: 1.1438 - val_accuracy: 0.6268
Epoch 8/15
130/130 [==============================] - ETA: 0s - loss: 0.3595 -
accuracy: 0.8695
Epoch 8: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 184s 1s/step - loss: 0.3595
- accuracy: 0.8695 - val_loss: 1.2994 - val_accuracy: 0.6206
Epoch 9/15
130/130 [==============================] - ETA: 0s - loss: 0.3281 -
accuracy: 0.8828
Epoch 9: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 182s 1s/step - loss: 0.3281
- accuracy: 0.8828 - val_loss: 1.3742 - val_accuracy: 0.6116
Epoch 10/15
130/130 [==============================] - ETA: 0s - loss: 0.3138 -
accuracy: 0.8832
Epoch 10: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 185s 1s/step - loss: 0.3138
- accuracy: 0.8832 - val_loss: 1.6238 - val_accuracy: 0.5964
Epoch 11/15
130/130 [==============================] - ETA: 0s - loss: 0.2895 -
accuracy: 0.9002
Epoch 11: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 183s 1s/step - loss: 0.2895
- accuracy: 0.9002 - val_loss: 1.3330 - val_accuracy: 0.6201
Epoch 12/15
130/130 [==============================] - ETA: 0s - loss: 0.2017 -
accuracy: 0.9287
Epoch 12: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 183s 1s/step - loss: 0.2017
- accuracy: 0.9287 - val_loss: 1.4926 - val_accuracy: 0.6082
Epoch 13/15
130/130 [==============================] - ETA: 0s - loss: 0.1842 -
```

```
accuracy: 0.9333
Epoch 13: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 183s 1s/step - loss: 0.1842
- accuracy: 0.9333 - val_loss: 1.6821 - val_accuracy: 0.6065
Epoch 14/15
130/130 [==============================] - ETA: 0s - loss: 0.1675 -
accuracy: 0.9432
Epoch 14: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 184s 1s/step - loss: 0.1675
- accuracy: 0.9432 - val_loss: 1.5852 - val_accuracy: 0.6201
Epoch 15/15
130/130 [==============================] - ETA: 0s - loss: 0.1699 -
accuracy: 0.9388
Epoch 15: saving model to training_second_model\cp.ckpt
130/130 [==============================] - 462s 4s/step - loss: 0.1699
- accuracy: 0.9388 - val_loss: 1.5076 - val_accuracy: 0.6206
```

```
custom_cnn2.save('Second_model.keras')
#custom_cnn2= tf.keras.models.load_model('Second_model.keras')
```

```
# Evaluate the model on the test data
loss, accuracy = custom_cnn2.evaluate(X_test, y_test)

# Print the test accuracy score
print(f'Test Accuracy: {accuracy}')
```

```
56/56 [==============================] - 81s 1s/step - loss: 1.5076 -
accuracy: 0.6206
Test Accuracy: 0.6206313371658325
```

For our second model, the accuracy fell down and became 62%.

Make and visualize some predictions.

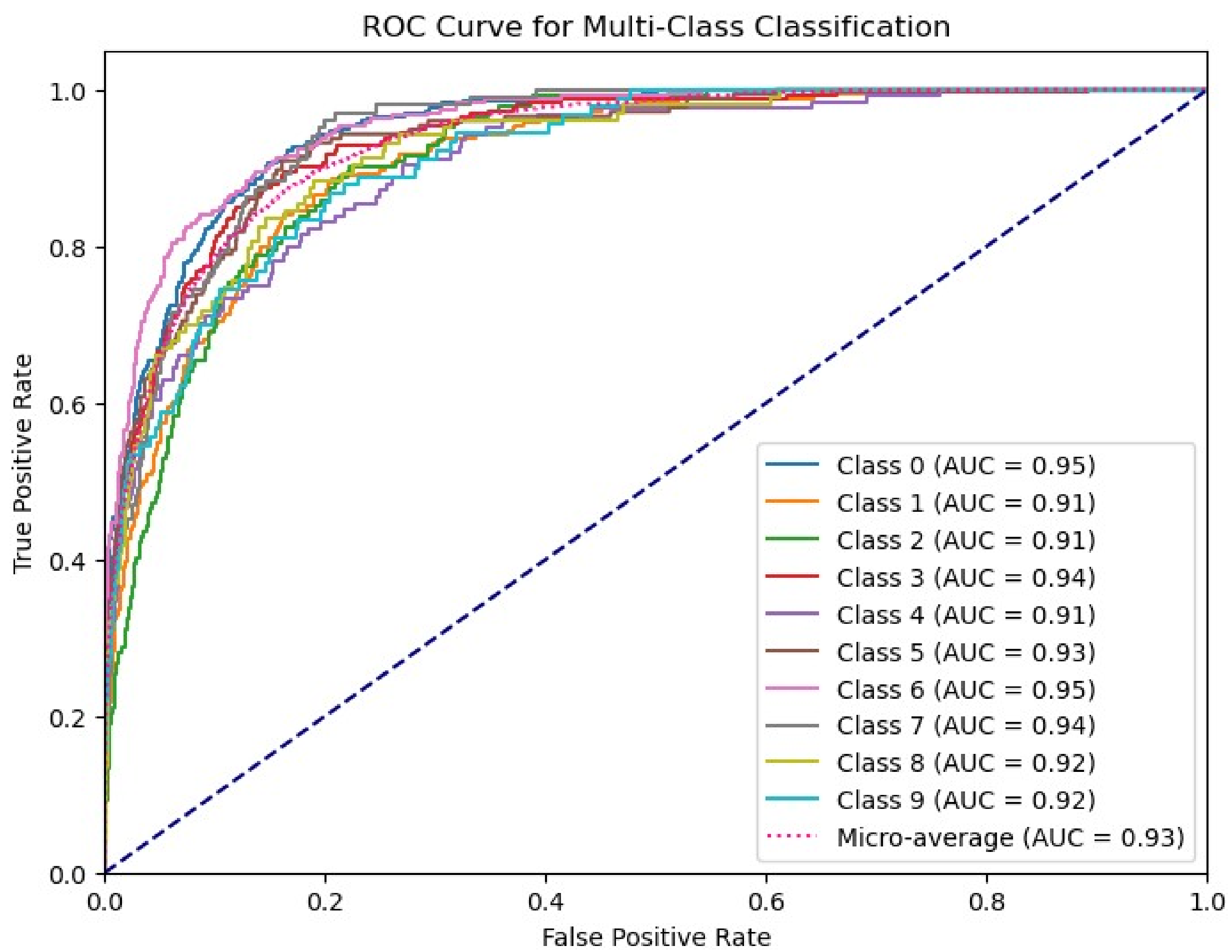
```
import numpy as np
import matplotlib.pyplot as plt

# X_test contains the test images and y_test contains the true labels

# Make predictions
predictions2 = custom_cnn2.predict(X_test)

# Convert one-hot encoded labels back to categorical labels
true_labels2 = np.argmax(y_test, axis=1)
predicted_labels2 = np.argmax(predictions2, axis=1)

# Visualize some predictions
num_images = 4  # Number of images to visualize
indices = np.random.choice(len(X_test), num_images, replace=False)
```

```
plt.figure(figsize=(15, 8))
for i, idx in enumerate(indices):
    plt.subplot(1, num_images, i+1)
    plt.imshow(X_test[idx])
    plt.title(f'True Label: {true_labels2[idx]}, Predicted Label:
{predicted_labels2[idx]}')
    plt.axis('off')
plt.show()

56/56 [==============================] - 67s 1s/step
```

Then we tried to show some predictions by plotting a test image against our predicted Label Results. For our 4 cases as depicted above, our model predicted a case with actual label 8 as 8, actual label 0 as 0 for two cases, and actual label 7 as 7.

```
# Convert true labels to one-hot encoded format
y_test_bin = label_binarize(y_test, classes=np.arange(10))  # Assuming
10 classes

# Compute ROC curve and ROC area for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(10):  # Assuming 10 classes
    fpr[i], tpr[i], _ = roc_curve(y_test_bin[:, i], predictions2[:,
i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_bin.ravel(),
predictions2.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plot ROC curve for each class
plt.figure(figsize=(8, 6))
for i in range(10):  # Assuming 10 classes
    plt.plot(fpr[i], tpr[i], label=f'Class {i} (AUC =
{roc_auc[i]:.2f})')
plt.plot(fpr["micro"], tpr["micro"], label='Micro-average (AUC =
```

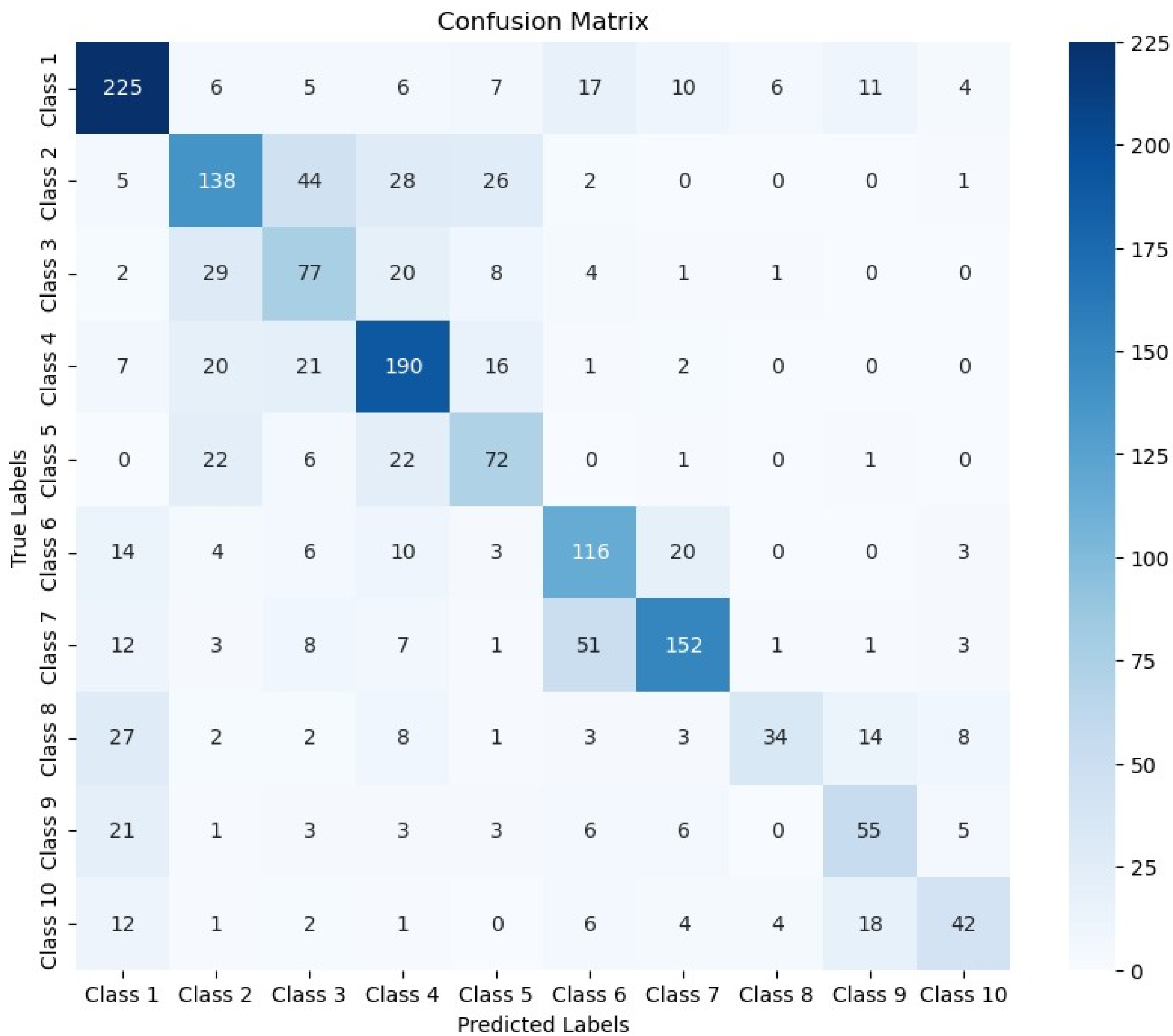
```
{0:0.2f})'.format(roc_auc["micro"]), color='deeppink', linestyle=':')
plt.plot([0, 1], [0, 1], color='navy', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for Multi-Class Classification')
plt.legend(loc="lower right")
plt.show()
```



For our case, the highest AUC belongs to class 0 and class 6, followed by class 3 and class 5. The poorest come from class 2 and class 4 which means that the model is least capable of differentiating these 2 classes. It also seems that for some classes, the result has fallen down for our new model. Previously for class 0, 1, 3,5, 7 the AUc was mugh higher than the second model as the second model has values 0.95, 0.91, 0.94, 0.93, 0.94 but the previous one had 0.97, 0.93, 0.96, 0.95. However the results improved for classes 2,4,6,9 and is the same for class 8.

```
# Define class labels
```

```
class_labels = ["Class 1", "Class 2", "Class 3", "Class 4", "Class 5",
                "Class 6", "Class 7", "Class 8", "Class 9", "Class
10"]

conf_matrix2 = confusion_matrix(true_labels2, predicted_labels2)


# Plot confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix2, annot=True, fmt="d", cmap="Blues",
xticklabels=class_labels, yticklabels=class_labels)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix")
plt.show()
```



Then we analyzed the confusion matrix for showing true classes for all 10 classes. To cite a few cases, we can see that 225 of class 1 has been corrrectly identified as class 1, 190 of class 4 has been corrrectly identified as class 4, 152 of class 7 has been corrrectly identified as class 7. But previously we see that our first model performed better for these classes as 241 of class 1 has been corrrectly identified as class 1, 216 of class 4 has been corrrectly identified as class 4, 200 of class 7 has been corrrectly identified as class 7. So the model performs relatively well for these classes. But for our second model, the performance improved for class 3 as it currently identified 77 class 3 classes perfectly compared to only 45 before. Results also improved visibly for class 5 as well.