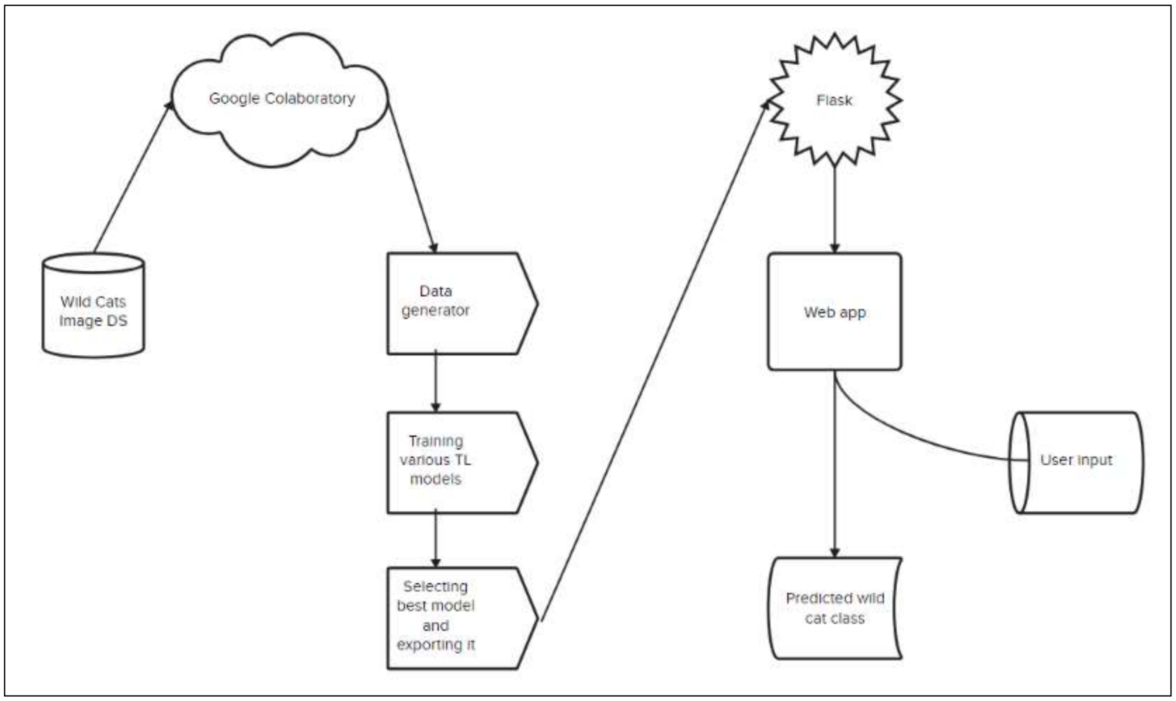
Jungle Detectives: AI-Powered Image Classification of Wild Big Cats

Introduction

Jungle Detectives is a transfer-learning based web app to identify the 10 big cats based on uploaded image. The app leverages transfer learning model to distinguish the majestic big cats. The big cats that can be identified are awe-inspiring Bengal tiger, the elusive snow leopard, the powerful African lion, the agile cheetah, the mysterious clouded leopard, the majestic jaguar, the enigmatic cougar, the endangered Amur leopard, the graceful serval, and the formidable leopard. By combining image recognition techniques with extensive training data, this state-of-the-art model provides conservationists, wildlife researchers, and enthusiasts with a powerful tool to safeguard and appreciate the beauty and diversity of these magnificent creatures, contributing to their preservation and understanding in their natural habitats.

With the advancement of technology, many new techniques of Deep learning have contributed towards classification of images in a more efficient way such as ResNet, VGG16, Inception V3 etc. Here in the given project, we are using the ResNet50V2 to classify the microorganisms into their original classes.

Technical Architecture



Pre-requisites

The following software's, concepts, and packages are required for the solution:

- Google Colaboratory for cloud-based python environment. It is used to train and test various models.
- Anaconda Navigator is a free and open-source distribution of the Python and R programming languages for data science and machine learning related applications. Is is an open-source, cross-platform, package management system. It is used for construct the deployment using Flask.

Following packages are required for machine learning model:

- Numpy: It is an open-source numerical Python library. It contains a multidimensional array and matrix data structures and can be used to perform mathematical operations.
- Scikit-learn: It is a free machine learning library for Python. It features various algorithms like support vector machine, random forests, and k-neighbors.
- TensorFlow: It is a free and open-source software library for machine learning and artificial intelligence.
- Flask: Web framework used for building Web applications.

Installing python packages:

1. Open anaconda prompt as administrator or use colab
2. Type "pip install numpy" and click enter.
3. Type "pip install pandas" and click enter.
4. Type "pip install scikit-learn" and click enter.
5. Type "pip install tensorflow==2.3.2" and click enter.
6. Type "pip install keras==2.3.1" and click enter.
7. Type "pip install Flask" and click enter.

Deep Learning Concepts

Deep learning is a subset of machine learning, which is essentially a neural network with three or more layers. These neural networks attempt to simulate the behavior of the human brain—albeit far from matching its ability—allowing it to "learn" from large amounts of data. CNN (convolutional neural network) is a class of deep neural networks, most commonly applied to analyzing visual imagery. ResNet50 is an implementation of CNN. ResNet-50 has an architecture based on the CNN, but with one important difference. The 50-layer ResNet uses a bottleneck design for the building block. A bottleneck residual block uses 1×1 convolution, known as a “bottleneck”, which reduces the number of parameters and matrix multiplications. This enables much faster training of each layer. It uses a stack of three layers rather than two layers.

Project Objectives

By the end of this project, the following concepts would be known:

- Fundamental concepts and techniques of Convolutional Neural Network.
- Understanding of image data.
- Pre-process/clean the data using different data pre-processing techniques.
- Build a web application using the Flask framework.

User experience:

- The user interacts with the UI (User Interface) to choose the image.
- The chosen image analyzed by the model which is integrated with flask application.
- ResNet50V2 Model analyses the image, then prediction is showcased on the Flask UI.

Project Flow

To accomplish objectives, the following tasks are required:

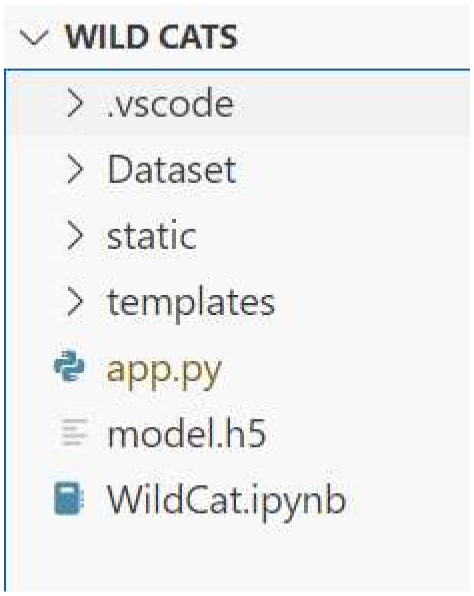
- Data Collection
    - Create Train and Test Folders
- Data Pre-processing
    - Import the ImageDataGenerator library
- Configure ImageDataGenerator class
    - Apply ImageDataGenerator functionality to Train Set, Test set and Valid set
- Model Building
    - Import the model building Libraries
    - Initializing the model
    - Adding Input Layer
    - Adding Hidden Layer
    - Adding Output Layer
    - Configure the Learning Process

- Training and testing the model
- Comparing performances of various models and selecting the best model
- Save the Model
- Application Building using Flask
    - Create an HTML file
    - Build Python Code

Project Structure

Overall project folder will contain the files and folders are shown in the below image.

- The Dataset folder contains the training, testing and validation images for training our model.
- The python notebook WildCat.ipynb was used for training the models and exporting the best model. The best model's weight has been exported as model.h5.
- We are building a Flask Application that needs HTML pages stored in the templates folder and a python script app.py for server-side scripting.
- Static folder is for storing the images uploaded for prediction by users.



```
WILD CATS
  > .vscode
  > Dataset
  > static
  > templates
  app.py
  model.h5
  WildCat.ipynb
```

Milestone 1 - Collection of Data

The dataset contains different images of big wild cats. It is downloaded from Kaggle data repository. There are three folders for train, test and validation separately. Within each folder, there are 10 different folders representing 10 classes of big wild cats. The given dataset has 10 different types of big cats, they are following:

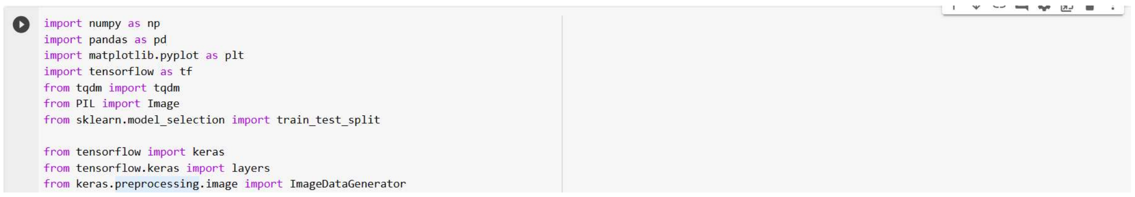
- Cheetah
- Lions
- Snow Leopard
- Caracal
- Tiger
- Clouded Leopard
- Puma
- Jaguar
- Ocelot
- African Leopard

Link to download the dataset: [10 Big Cats of the Wild - Image Classification | Kaggle]()

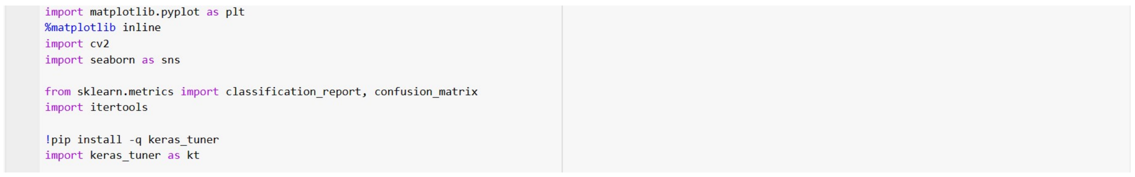
Milestone 2 – Model building

The dataset is loaded and Data generator will be used. Some geometric transformations of images like rescaling, rotating, width shifting, height shifting, horizontal flipping is performed. The following actions are required:

1. Import the ImageDataGenerator library
    - Image data augmentation is a technique that can be used to artificially expand the size of a training dataset by creating modified versions of images in the dataset.
    - The Keras deep learning neural network library provides the capability to fit models using image data augmentation via the ImageDataGenerator class. Let us import the ImageDataGenerator class from TensorFlow Keras

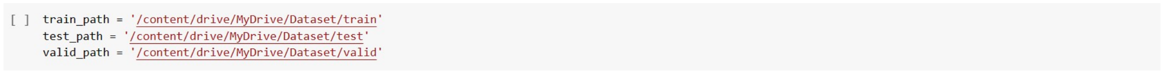


```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tqdm import tqdm
from PIL import Image
from sklearn.model_selection import train_test_split

from tensorflow import keras
from tensorflow.keras import layers
from keras.preprocessing.image import ImageDataGenerator
```

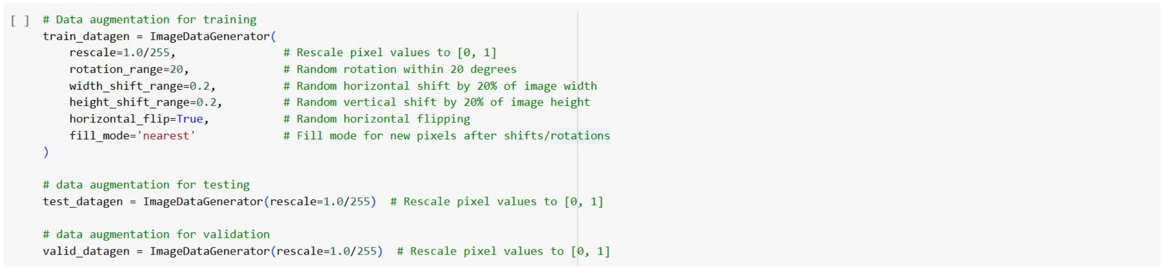


```
import matplotlib.pyplot as plt
%matplotlib inline
import cv2
import seaborn as sns

from sklearn.metrics import classification_report, confusion_matrix
import itertools

!pip install -q keras_tuner
import keras_tuner as kt
```

2. Configure ImageDataGenerator class
    - ImageDataGenerator class is instantiated and the configuration for the types of data augmentation. This repeated for train, test and validation.



```
train_path = '/content/drive/MyDrive/Dataset/train'
test_path = '/content/drive/MyDrive/Dataset/test'
valid_path = '/content/drive/MyDrive/Dataset/valid'
```

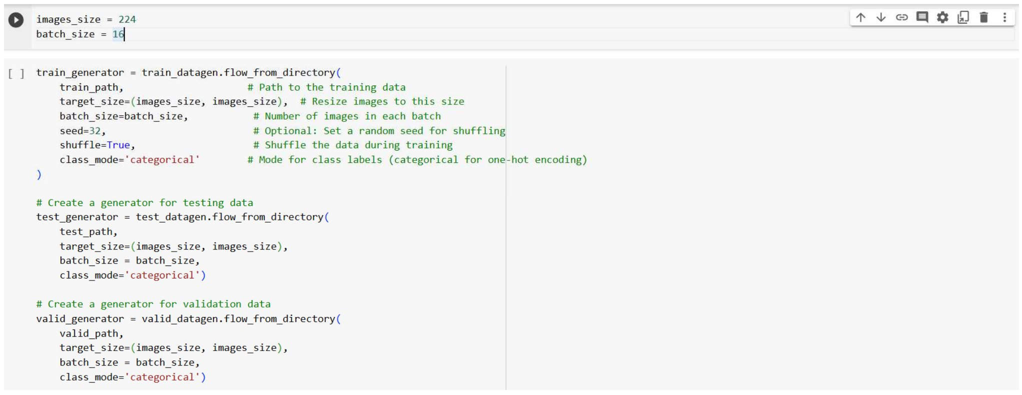


```
# Data augmentation for training
train_datagen = ImageDataGenerator(
    rescale=1.0/255,                # Rescale pixel values to [0, 1]
    rotation_range=20,              # Random rotation within 20 degrees
    width_shift_range=0.2,          # Random horizontal shift by 20% of image width
    height_shift_range=0.2,         # Random vertical shift by 20% of image height
    horizontal_flip=True,           # Random horizontal flipping
    fill_mode='nearest'             # Fill mode for new pixels after shifts/rotations
)

# data augmentation for testing
test_datagen = ImageDataGenerator(rescale=1.0/255)  # Rescale pixel values to [0, 1]

# data augmentation for validation
valid_datagen = ImageDataGenerator(rescale=1.0/255)  # Rescale pixel values to [0, 1]
```

- Five main types of data augmentation techniques were used:
    - Image rescaling via rescale.
    - Image rotations via the rotation_range argument.
    - Image shifts via the width_shift_range and height_shift_range arguments.
    - The image flips via the horizontal_flip.
- An instance of the ImageDataGenerator class can be constructed for train, test and validation.

3. Apply ImageDataGenerator functionality to Train, Test and Validation set
    - Let us apply ImageDataGenerator functionality.
    - Arguments:
        - directory: Directory where the data is located. If labels are "inferred", it should contain subdirectories, each containing images for a class. Otherwise, the directory structure is ignored.
        - batch_size: Size of the batches of data which is 64.
        - target_size: Size to resize images after they are read from disk.
        - Shuffleing true or false.
        - class_mode:
            - 'int' means that the labels are encoded as integers (e.g., for sparse_categorical_crossentropy loss).

- 'categorical' means that the labels are encoded as a categorical vector (e.g., for categorical_crossentropy loss).
- 'binary' means that the labels (there can be only 2) are encoded as float32 scalars with values 0 or 1 (e.g., for binary_crossentropy).
- None (no labels).

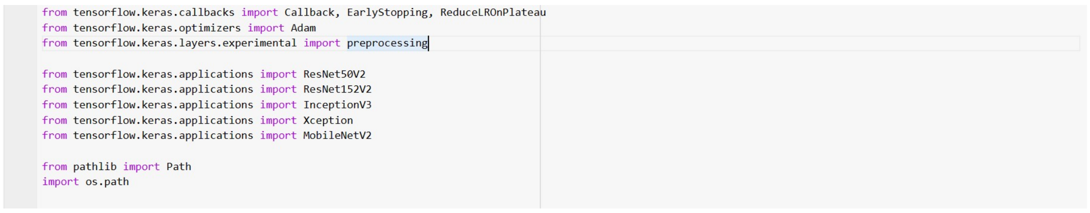


```
images_size = 224
batch_size = 16
```

```
train_generator = train_datagen.flow_from_directory(
    train_path,                   # Path to the training data
    target_size=(images_size, images_size),  # Resize images to this size
    batch_size=batch_size,         # Number of images in each batch
    seed=32,                       # Optional: Set a random seed for shuffling
    shuffle=True,                  # Shuffle the data during training
    class_mode='categorical'       # Mode for class labels (categorical for one-hot encoding)
)

# Create a generator for testing data
test_generator = test_datagen.flow_from_directory(
    test_path,
    target_size=(images_size, images_size),
    batch_size = batch_size,
    class_mode='categorical')

# Create a generator for validation data
valid_generator = valid_datagen.flow_from_directory(
    valid_path,
    target_size=(images_size, images_size),
    batch_size = batch_size,
    class_mode='categorical')
```

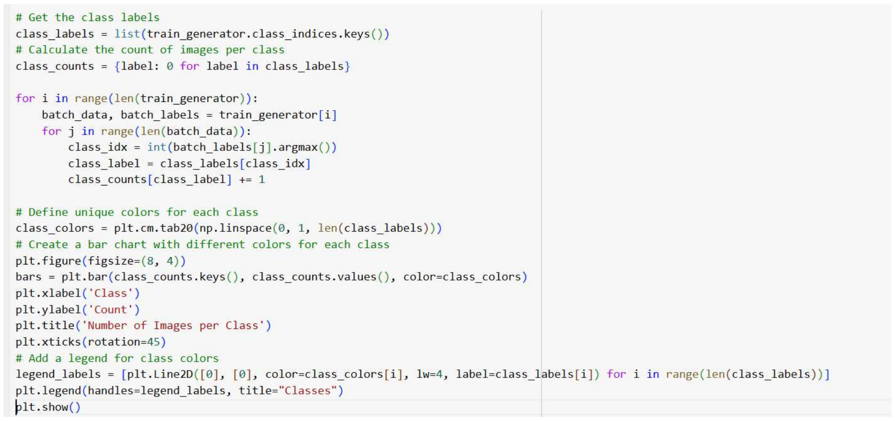
4. Model Selection and training

- A subset of 1000 images are taken to train 5 different transfer learning models to find which one is better in performance.
- The following tasks are required:
    i. Importing the Model Building Libraries

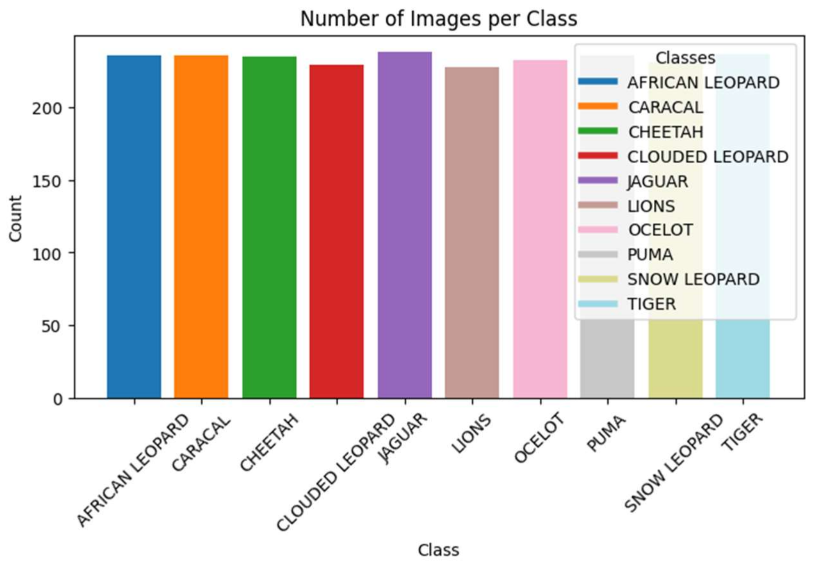


```
from tensorflow.keras.callbacks import Callback, EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers.experimental import preprocessing

from tensorflow.keras.applications import ResNet50V2
from tensorflow.keras.applications import ResNet152V2
from tensorflow.keras.applications import InceptionV3
from tensorflow.keras.applications import Xception
from tensorflow.keras.applications import MobileNetV2

from pathlib import Path
import os.path
```

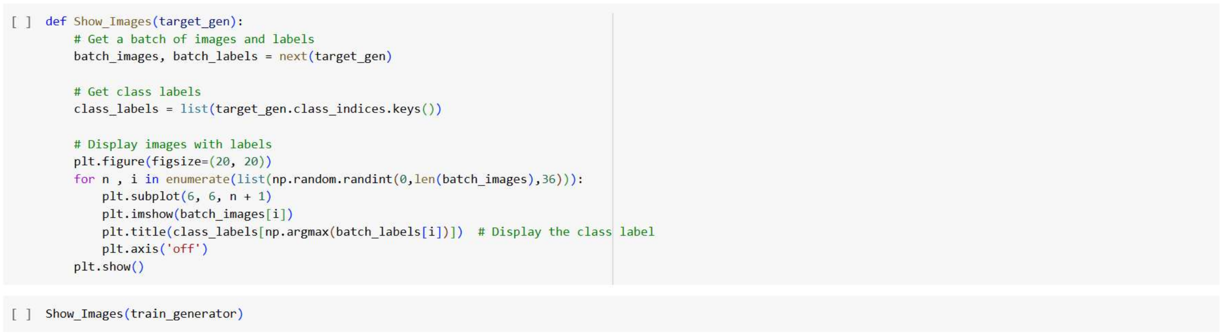
    ii. Exploratory Data Analysis



```
# Get the class labels
class_labels = list(train_generator.class_indices.keys())
# Calculate the count of images per class
class_counts = {label: 0 for label in class_labels}

for i in range(len(train_generator)):
    batch_data, batch_labels = train_generator[i]
    for j in range(len(batch_data)):
        class_idx = int(batch_labels[j].argmax())
        class_label = class_labels[class_idx]
        class_counts[class_label] += 1

# Define unique colors for each class
class_colors = plt.cm.tab20(np.linspace(0, 1, len(class_labels)))
# Create a bar chart with different colors for each class
plt.figure(figsize=(8, 4))
bars = plt.bar(class_counts.keys(), class_counts.values(), color=class_colors)
plt.xlabel('Class')
plt.ylabel('Count')
plt.title('Number of Images per Class')
plt.xticks(rotation=45)
# Add a legend for class colors
legend_labels = [plt.Line2D([0], [0], color=class_colors[i], lw=4, label=class_labels[i]) for i in range(len(class_labels))]
plt.legend(handles=legend_labels, title="Classes")
plt.show()
```

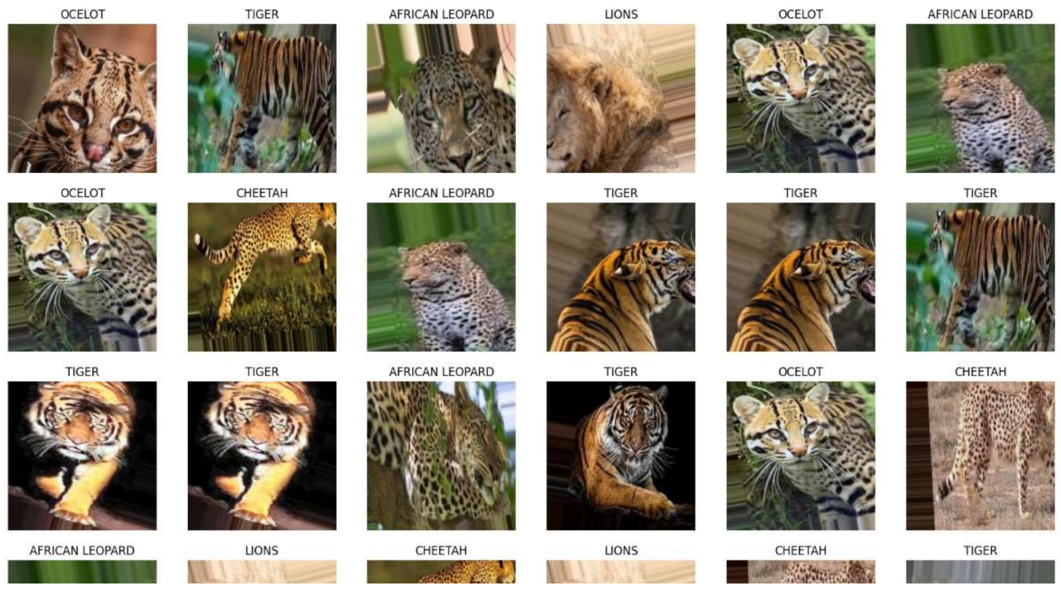


iii. Visualizing the images:

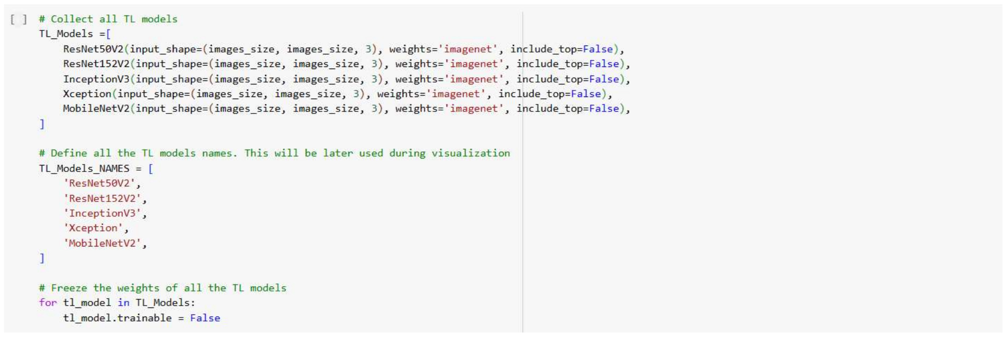


```
[ ] def Show_Images(target_gen):
        # Get a batch of images and labels
        batch_images, batch_labels = next(target_gen)

        # Get class labels
        class_labels = list(target_gen.class_indices.keys())

        # Display images with labels
        plt.figure(figsize=(20, 20))
        for n , i in enumerate(list(np.random.randint(0,len(batch_images),36))):
            plt.subplot(6, 6, n + 1)
            plt.imshow(batch_images[i])
            plt.title(class_labels[np.argmax(batch_labels[i])])  # Display the class label
            plt.axis('off')
        plt.show()
```

```
[ ] Show_Images(train_generator)
```

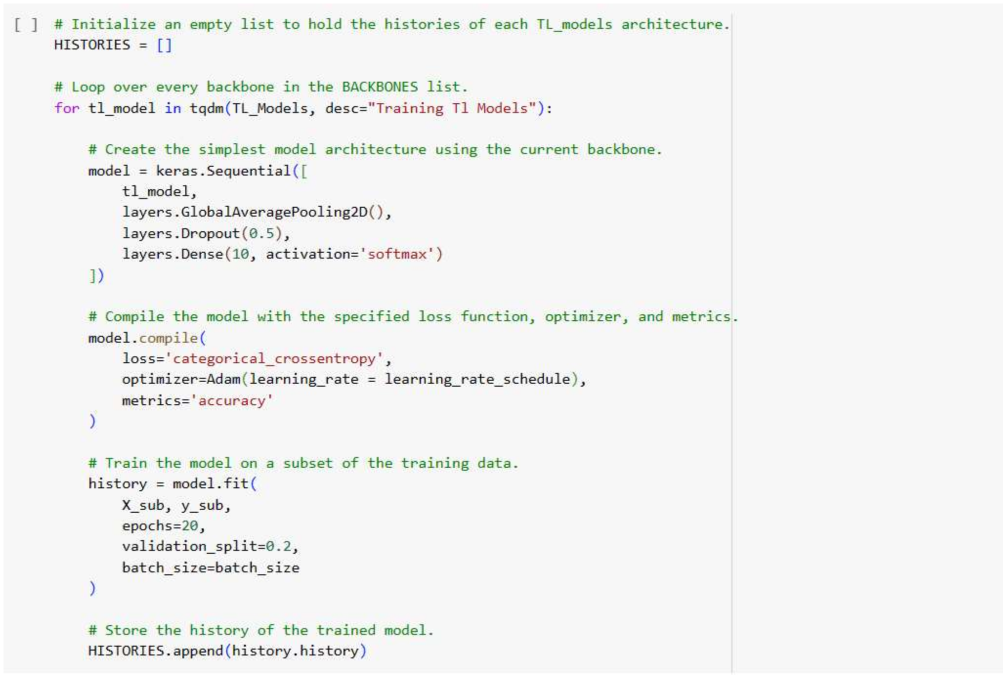


iv. Initializing the models

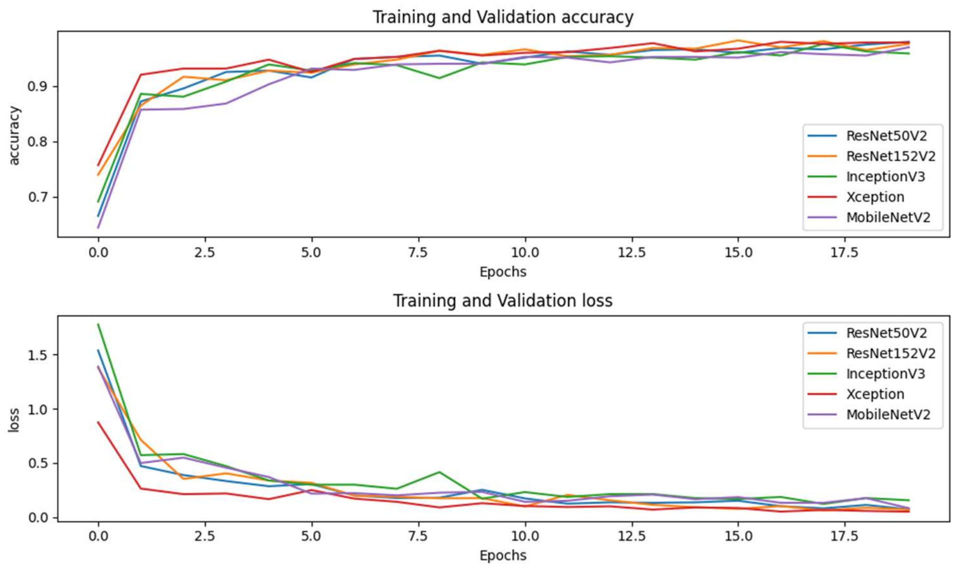


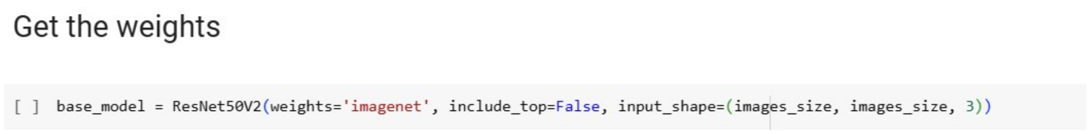
```
[ ] # Collect all TL models
    TL_Models =[
        ResNet50V2(input_shape=(images_size, images_size, 3), weights='imagenet', include_top=False),
        ResNet152V2(input_shape=(images_size, images_size, 3), weights='imagenet', include_top=False),
        InceptionV3(input_shape=(images_size, images_size, 3), weights='imagenet', include_top=False),
        Xception(input_shape=(images_size, images_size, 3), weights='imagenet', include_top=False),
        MobileNetV2(input_shape=(images_size, images_size, 3), weights='imagenet', include_top=False),
    ]

    # Define all the TL models names. This will be later used during visualization
    TL_Models_NAMES = [
        'ResNet50V2',
        'ResNet152V2',
        'InceptionV3',
        'Xception',
        'MobileNetV2',
    ]

    # Freeze the weights of all the TL models
    for tl_model in TL_Models:
        tl_model.trainable = False
```

v. Training all and comparing. Saving weights of best model.



```
[ ] # Initialize an empty list to hold the histories of each TL_models architecture.
    HISTORIES = []

    # Loop over every backbone in the BACKBONES list.
    for tl_model in tqdm(TL_Models, desc="Training Tl Models"):

        # Create the simplest model architecture using the current backbone.
        model = keras.Sequential([
            tl_model,
            layers.GlobalAveragePooling2D(),
            layers.Dropout(0.5),
            layers.Dense(10, activation='softmax')
        ])

        # Compile the model with the specified loss function, optimizer, and metrics.
        model.compile(
            loss='categorical_crossentropy',
            optimizer=Adam(learning_rate = learning_rate_schedule),
            metrics='accuracy'
        )

        # Train the model on a subset of the training data.
        history = model.fit(
            X_sub, y_sub,
            epochs=20,
            validation_split=0.2,
            batch_size=batch_size
        )

        # Store the history of the trained model.
        HISTORIES.append(history.history)
```

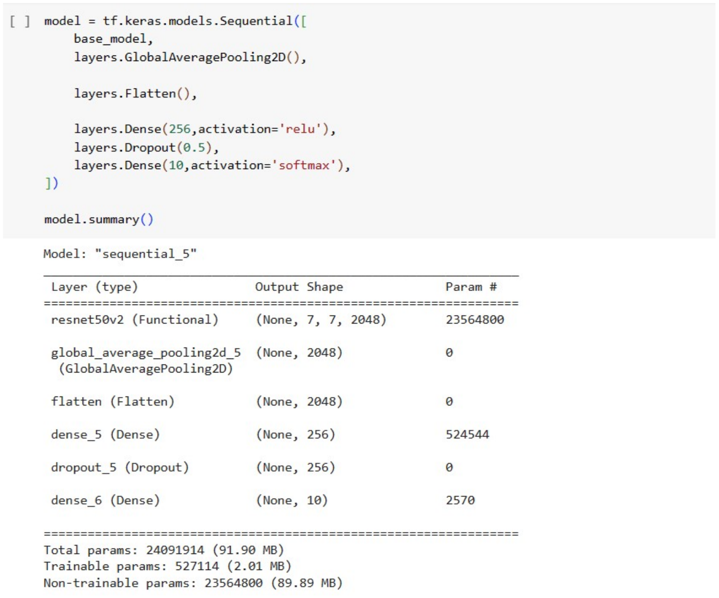



Get the weights

```
[ ]  base_model = ResNet50V2(weights='imagenet', include_top=False, input_shape=(images_size, images_size, 3))
```

vi. Training the model

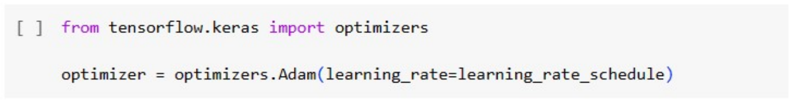
- The better performing model among five is ResNet50V2.
- It is selected for further training.
- Extra layers are added and model is compiled with required metrics.



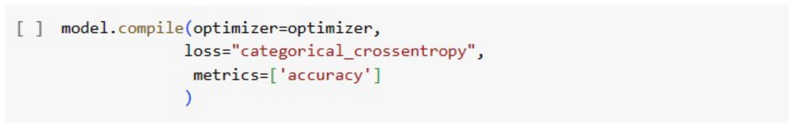
```
[ ]  model = tf.keras.models.Sequential([
         base_model,
         layers.GlobalAveragePooling2D(),

         layers.Flatten(),

         layers.Dense(256,activation='relu'),
         layers.Dropout(0.5),
         layers.Dense(10,activation='softmax'),
     ])

     model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50v2 (Functional)     (None, 7, 7, 2048)        23564800

 global_average_pooling2d_5  (None, 2048)              0
  (GlobalAveragePooling2D)

 flatten (Flatten)           (None, 2048)              0

 dense_5 (Dense)             (None, 256)               524544

 dropout_5 (Dropout)         (None, 256)               0

 dense_6 (Dense)             (None, 10)                2570

=================================================================
Total params: 24091914 (91.90 MB)
Trainable params: 527114 (2.01 MB)
Non-trainable params: 23564800 (89.89 MB)
```

- adam optimizer is used.



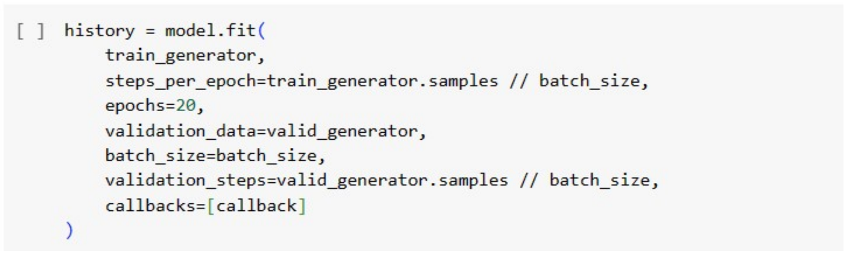
```
[ ]  from tensorflow.keras import optimizers

     optimizer = optimizers.Adam(learning_rate=learning_rate_schedule)
```



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

- The model is trained for 20 epochs. Arguments:
    - steps_per_epoch: it specifies the total number of steps taken from the generator as soon as one epoch is finished and the next epoch has started. We can calculate the value of steps_per_epoch as the total number of samples in your dataset divided by the batch size.
    - Epochs: an integer and number of epochs we want to train our model for.

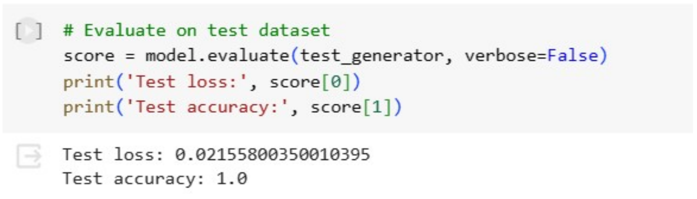
- validation_data can be either: an inputs and targets list a generator an inputs, targets, and sample_weights list which can be used to evaluate the loss and metrics for any model after any epoch has ended.
- validation_steps: only if the validation_data is a generator then only this argument can be used. It specifies the total number of steps taken from the generator before it is stopped at every epoch and its value is calculated as the total number of validation data points in your dataset divided by the validation batch size.



```
[ ] history = model.fit(
        train_generator,
        steps_per_epoch=train_generator.samples // batch_size,
        epochs=20,
        validation_data=valid_generator,
        batch_size=batch_size,
        validation_steps=valid_generator.samples // batch_size,
        callbacks=[callback]
    )
```

vii. Analyzing performance metrics

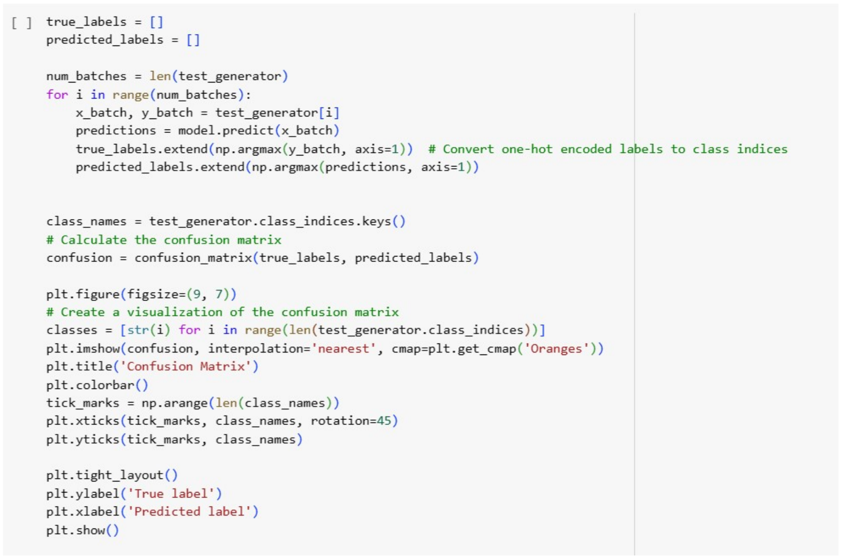
- Evaluating on test data.



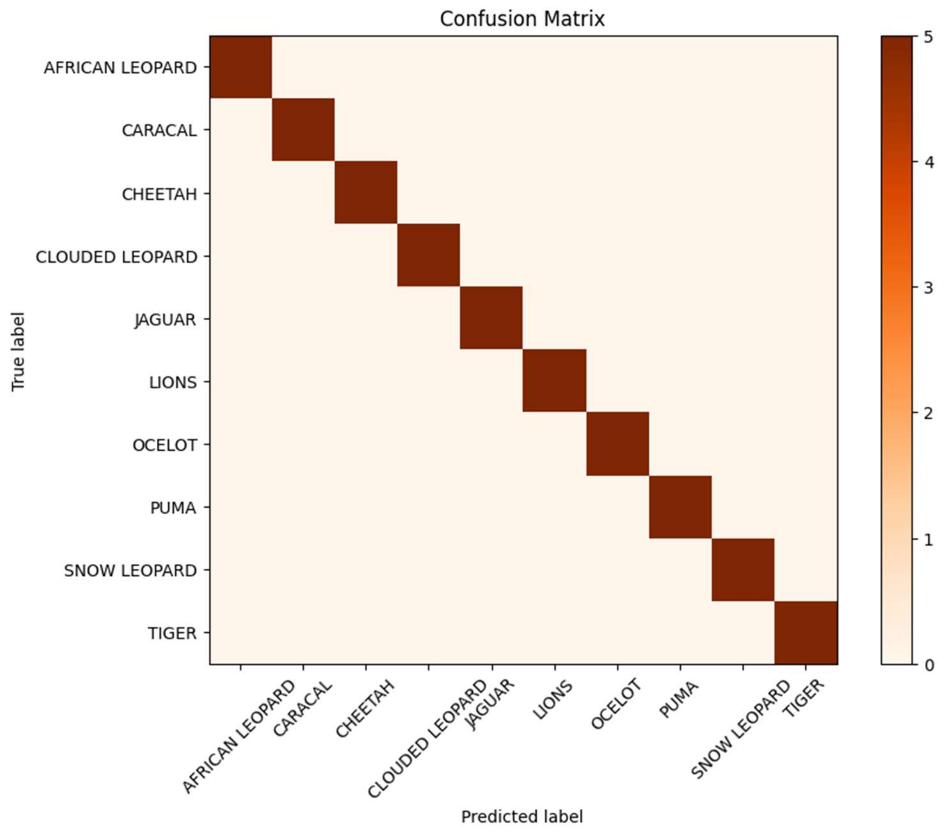
```
[ ] # Evaluate on test dataset
    score = model.evaluate(test_generator, verbose=False)
    print('Test loss:', score[0])
    print('Test accuracy:', score[1])
```

```
Test loss: 0.021558003500010395
Test accuracy: 1.0
```

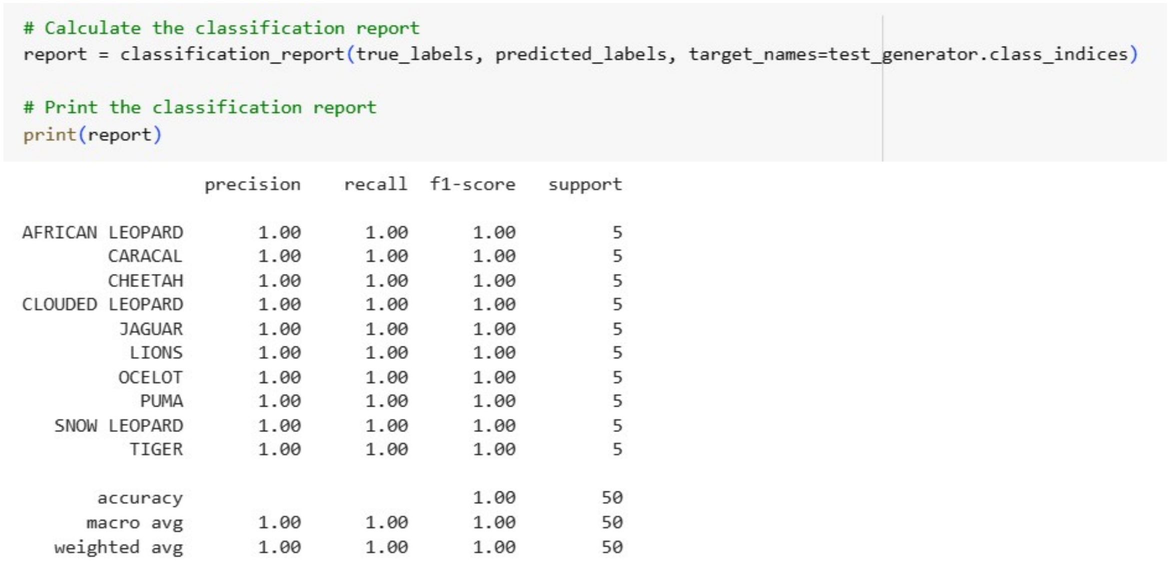
- Observing the conclusion matrix to see how well the model is able to classify correctly. All the images have been correctly classified.



```
[ ] true_labels = []
    predicted_labels = []

    num_batches = len(test_generator)
    for i in range(num_batches):
        x_batch, y_batch = test_generator[i]
        predictions = model.predict(x_batch)
        true_labels.extend(np.argmax(y_batch, axis=1))  # Convert one-hot encoded labels to class indices
        predicted_labels.extend(np.argmax(predictions, axis=1))


    class_names = test_generator.class_indices.keys()
    # Calculate the confusion matrix
    confusion = confusion_matrix(true_labels, predicted_labels)

    plt.figure(figsize=(9, 7))
    # Create a visualization of the confusion matrix
    classes = [str(i) for i in range(len(test_generator.class_indices))]
    plt.imshow(confusion, interpolation='nearest', cmap=plt.get_cmap('Oranges'))
    plt.title('Confusion Matrix')
    plt.colorbar()
    tick_marks = np.arange(len(class_names))
    plt.xticks(tick_marks, class_names, rotation=45)
    plt.yticks(tick_marks, class_names)

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()
```



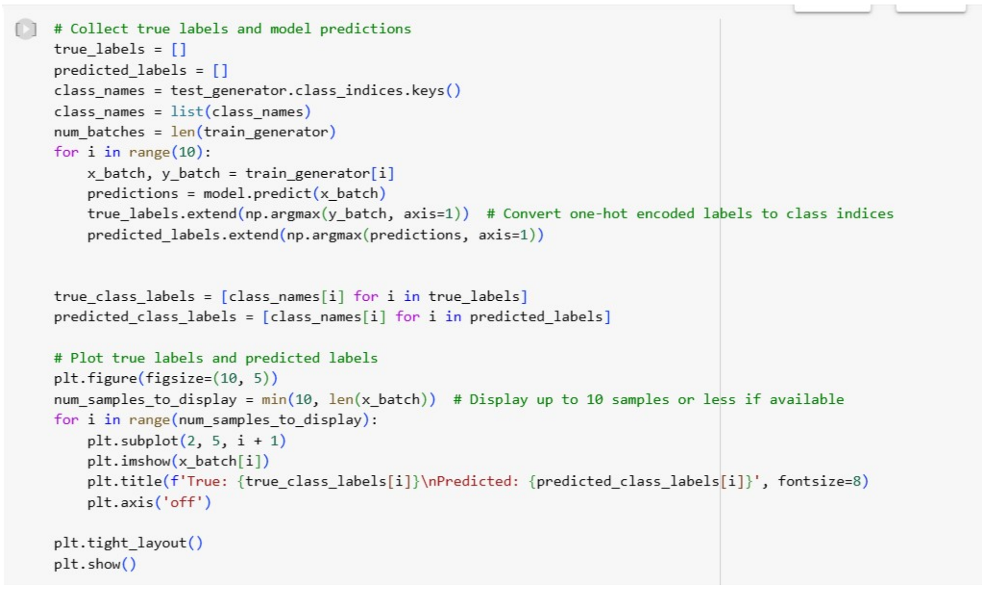
- Studying the classification report. The model is performing highly precise for the test data.



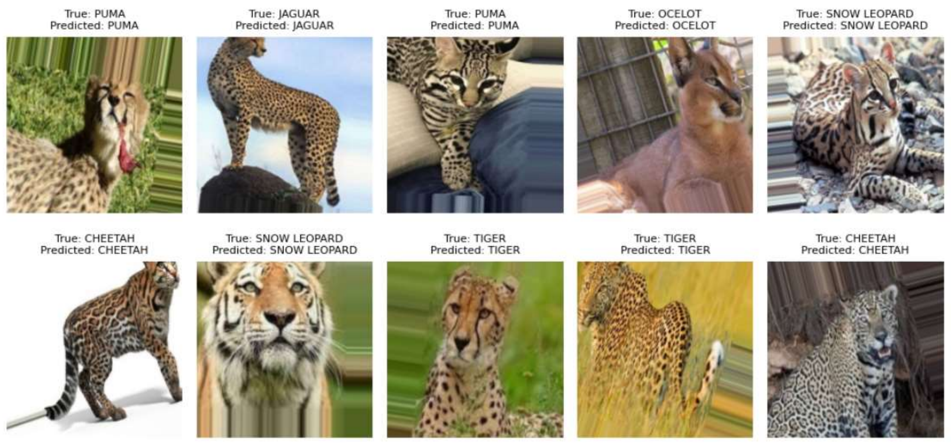
```
# Calculate the classification report
report = classification_report(true_labels, predicted_labels, target_names=test_generator.class_indices)

# Print the classification report
print(report)
```

```
                 precision    recall  f1-score   support

AFRICAN LEOPARD       1.00      1.00      1.00         5
        CARACAL       1.00      1.00      1.00         5
        CHEETAH       1.00      1.00      1.00         5
CLOUDED LEOPARD       1.00      1.00      1.00         5
         JAGUAR       1.00      1.00      1.00         5
          LIONS       1.00      1.00      1.00         5
         OCELOT       1.00      1.00      1.00         5
           PUMA       1.00      1.00      1.00         5
   SNOW LEOPARD       1.00      1.00      1.00         5
          TIGER       1.00      1.00      1.00         5

       accuracy                           1.00        50
      macro avg       1.00      1.00      1.00        50
   weighted avg       1.00      1.00      1.00        50
```

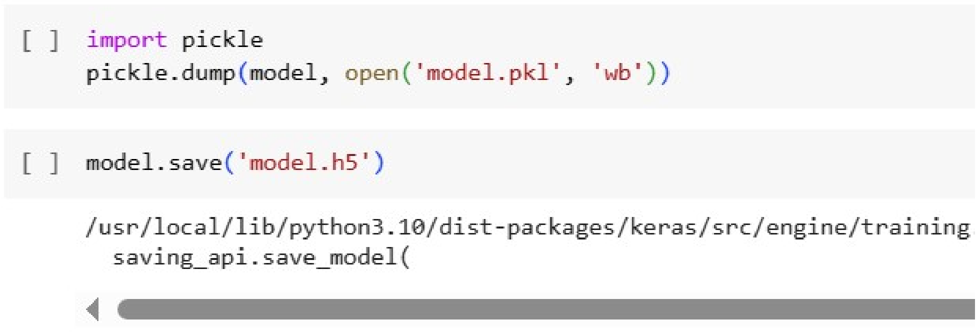
viii. Testing some samples



```
# Collect true labels and model predictions
true_labels = []
predicted_labels = []
class_names = test_generator.class_indices.keys()
class_names = list(class_names)
num_batches = len(train_generator)
for i in range(10):
    x_batch, y_batch = train_generator[i]
    predictions = model.predict(x_batch)
    true_labels.extend(np.argmax(y_batch, axis=1))  # Convert one-hot encoded labels to class indices
    predicted_labels.extend(np.argmax(predictions, axis=1))


true_class_labels = [class_names[i] for i in true_labels]
predicted_class_labels = [class_names[i] for i in predicted_labels]

# Plot true labels and predicted labels
plt.figure(figsize=(10, 5))
num_samples_to_display = min(10, len(x_batch))  # Display up to 10 samples or less if available
for i in range(num_samples_to_display):
    plt.subplot(2, 5, i + 1)
    plt.imshow(x_batch[i])
    plt.title(f'True: {true_class_labels[i]}\nPredicted: {predicted_class_labels[i]}', fontsize=8)
    plt.axis('off')

plt.tight_layout()
plt.show()
```



ix. Saving the model in appropriate format to integrate using Flask



```
[ ]  import pickle
     pickle.dump(model, open('model.pkl', 'wb'))
```

```
[ ]  model.save('model.h5')
```

```
     /usr/local/lib/python3.10/dist-packages/keras/src/engine/training
       saving_api.save_model(
```

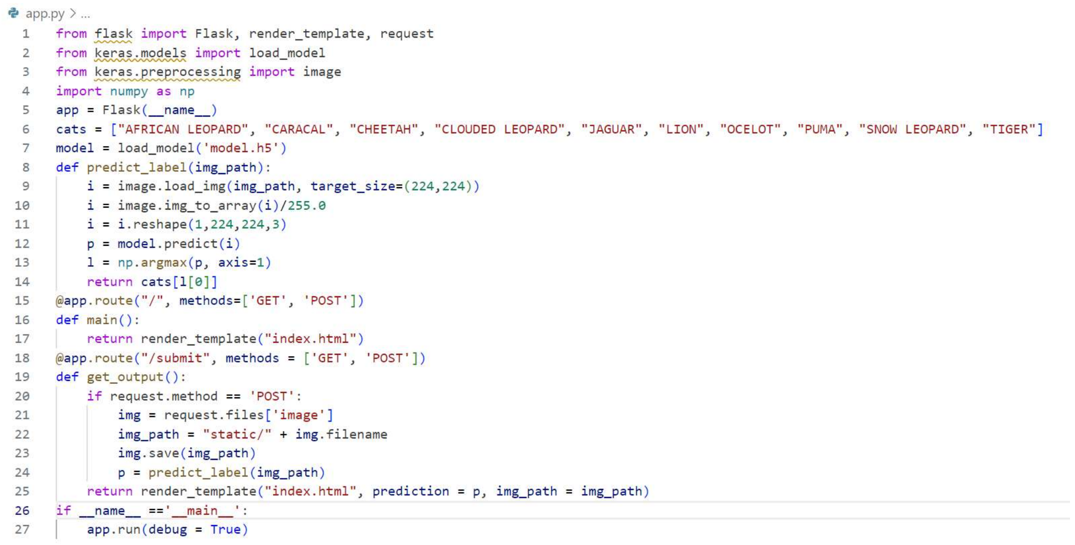
Milestone 3 – Application Building

- Now that we have trained our model, let us build our flask application which will be running in our local browser with a user interface.
- In the flask application, the input parameters are taken from the HTML page These factors are then given to the model to know to predict the type of wild cat and showcase on the HTML page to notify the user.

i. Create HTML Pages

- We use HTML to create the front-end part of the web page.
- We also use Java Script and CSS to enhance our functionality and view of HTML pages.

ii. Create app.py (Python Flask) file

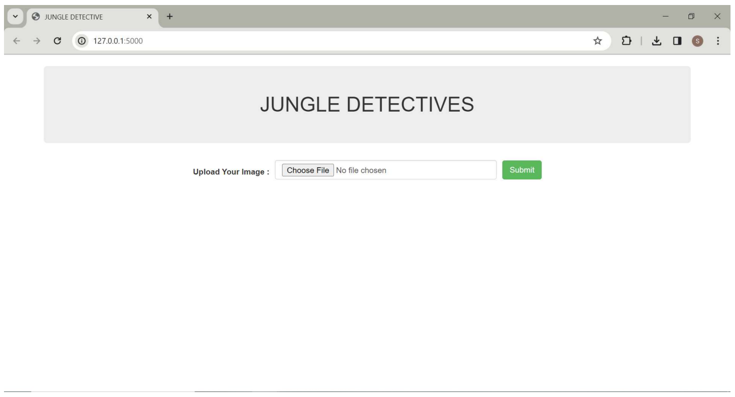


app.py > ...

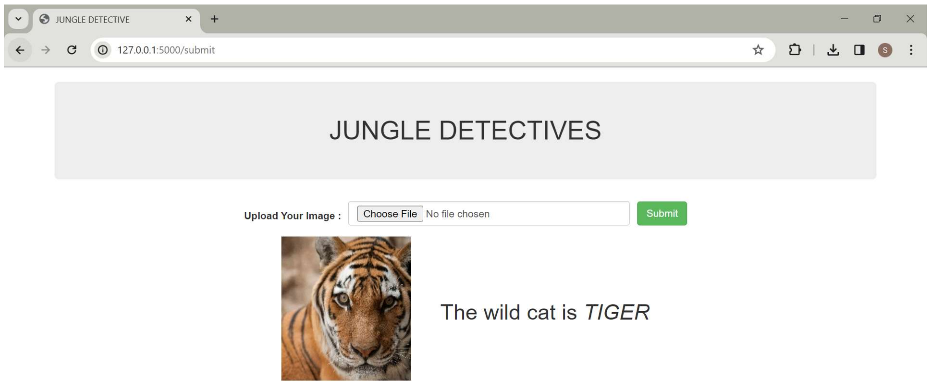
```python
from flask import Flask, render_template, request
from keras.models import load_model
from keras.preprocessing import image
import numpy as np
app = Flask(__name__)
cats = ["AFRICAN LEOPARD", "CARACAL", "CHEETAH", "CLOUDED LEOPARD", "JAGUAR", "LION", "OCELOT", "PUMA", "SNOW LEOPARD", "TIGER"]
model = load_model('model.h5')
def predict_label(img_path):
    i = image.load_img(img_path, target_size=(224,224))
    i = image.img_to_array(i)/255.0
    i = i.reshape(1,224,224,3)
    p = model.predict(i)
    l = np.argmax(p, axis=1)
    return cats[l[0]]
@app.route("/", methods=['GET', 'POST'])
def main():
    return render_template("index.html")
@app.route("/submit", methods = ['GET', 'POST'])
def get_output():
    if request.method == 'POST':
        img = request.files['image']
        img_path = "static/" + img.filename
        img.save(img_path)
        p = predict_label(img_path)
    return render_template("index.html", prediction = p, img_path = img_path)
if __name__ =='__main__':
    app.run(debug = True)
```

Output

Initial page.



Output for an image submitted.

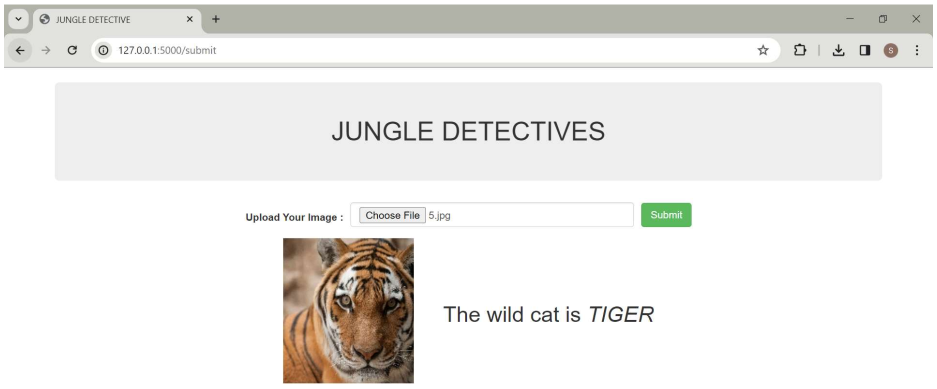


JUNGLE DETECTIVES

Upload Your Image : No file chosen

The wild cat is *TIGER*

Submitting another image.

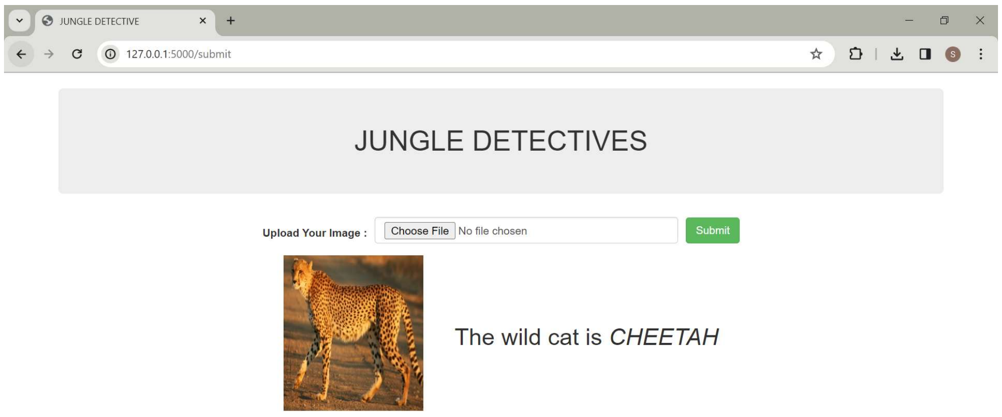


JUNGLE DETECTIVES

Upload Your Image : 5.jpg

The wild cat is *TIGER*

Output for second submitted image.



JUNGLE DETECTIVES

Upload Your Image : No file chosen

The wild cat is *CHEETAH*