Neural Networks Project

Image Classifier

1. Introduction:
   First, let's start with what the project does. An image classifier is a neural network model that is trained to recognize a single object in a picture. The model presented in this project is trained to detect between objects that would be commonly found on a street: bicycles, busses, cars, humans, traffic lights or traffic signs. This model is quite important, as it is the first step to an object detection model, which is made to locate and detect all objects in an image, or live footage. Such a model is used in autonomous driving cars.

2. Setup:
   a. Prerequisites:
      This model requires any IDE you like, I use Visual Studio Code, Python 3.10 specifically, Anaconda, TensorFlow 2.10, OpenCv, Matplotlib and Numpy. The installation of these will be provided a bit later.
   b. Python 3.10:
      - Python can be downloaded from the following link: Python Release Python 3.10.0 | Python.org
      - When installing, make sure to check the box that says "Add Python3 to PATH".
   c. Anaconda and Anaconda Environment:
      - Anaconda can be downloaded from the following link: Download Anaconda Distribution | Anaconda
      - After installing, open Anaconda Prompt and write the following commands:

```
conda create -n "any name you want" python=3.10
conda activate "your environment name"
conda install -c conda-forge cudatoolkit=11.2
cudnn=8.1.0
python -m pip install "tensorflow<2.11"
python -m pip install "opencv-python"
python -m pip install "matplotlib"
python -m pip install "numpy"
```

      - Now, your anaconda environment should be complete.

3. Dataset:
   The dataset I have used was made by myself using an extension of Chrome to download all images Google Images returned for each object. I then filtered the data with a script inside the classifiers code.

4. Project file structure:
   The file structure of project is as follows:
   a. Data directory: stores the directories: bicycle, bus, car, human, traffic light and traffic sign. Each subdirectory contains the photos of each object.
   b. Images_to_use directory: here are images that should not be in the data directory to test the model.
   c. Model directory: here is the model file after it was trained and compiled.
   d. Classifier.py file: here lies the code of the model.
   e. GUI.py: makes a GUI so the user can use test the model without training it with every run.

5. Code structure
   Here I will attach the code and explain what it does and how it works:
   a. Imports:

```
import tensorflow as tf

import cv2, imghdr
import os
from matplotlib import pyplot as plt
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dense, Flatten
from keras.metrics import Precision, Recall, CategoricalAccuracy
from keras.callbacks import EarlyStopping
import numpy as np
```

b. GPU setup:
The code is going to search for all GPUs in the system and tell TensorFlow to limit the memory usage and not allow it to use it all, resulting in an out of memory exception.

```
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
tf.config.list_logical_devices('GPU')
```

c. Image Filtering:
The quality and format of the images is crucial when training a model. Considering the images were downloaded off Google Images, there are a lot of images that “lie” about their format, are corrupted, are way too small or are not of a format that TensorFlow can work with, so the code goes through all the data subdirectories and checks each image if: are of correct format, then if they are of the minimum allowed size and are not corrupted and deletes all the images that either do not pass the filter, or raise an exception when loaded.

```
allowed_formats = ['jpeg', 'jpg', 'png', 'bmp']
base_folder = "data"
size_threshold_kb = 10
for folder in os.listdir(base_folder):
    folder_path = os.path.join(base_folder, folder)
    if os.path.isdir(folder_path):
        for file in os.listdir(folder_path):
            file_path = os.path.join(folder_path, file)
            file_ext = file.split('.')[-1].lower()
            file_size_kb = os.path.getsize(file_path) / 1024
            try:
                img = cv2.imread(file_path)
                tip = imghdr.what(file_path)
                if file_size_kb < size_threshold_kb:
                    print(f'File {file_path} is under {size_threshold_kb} KB')
                    os.remove(file_path)
                elif tip not in allowed_formats:
                    print(f'Image not in ext list {file_path} (detected as {tip})')
                    os.remove(file_path)
                elif img is None:
                    print(f'Image is corrupted {file_path}')
                    os.remove(file_path)
            except Exception as e:
                print(f'Issue with image {file_path}: {e}')
```

d. Data loading:
This bit of code creates a dataset with the images from the subdirectories

```
data = tf.keras.utils.image_dataset_from_directory('data', batch_size=32, image_size=(256, 256))
data = data.map(lambda x, y: (x / 255.0, y))
```

of the data directory, then it splits the images in batches of 32 images and a size of 256px by 256px.

e. Splitting the data:
In order to accurately train and evaluate the model, it is important to split the dataset into 3 sets: train, validate and test set, then use the 3 to evaluate the training correctly.

```
dataset_size = len(data)
train_size = int(0.7 * dataset_size)
val_size = int(0.2 * dataset_size)
test_size = int(0.1 * dataset_size)

train = data.take(train_size)
val = data.skip(train_size).take(val_size)
test = data.skip(train_size + val_size).take(test_size)
```

f. Model:
Here we give the instruction on how the model should be build

```
model = Sequential([
    Conv2D(16, (3, 3), activation='relu', input_shape=(256, 256, 3)),
    MaxPooling2D(),
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D(),
    Conv2D(16, (3, 3), activation='relu'),
    MaxPooling2D(),
    Flatten(),
    Dense(256, activation='relu'),
    Dense(6, activation='softmax')
])

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

I will now explain what everything means:

- Sequential is a model type that build a neural network layer by layer, in a linear fashion. It is optimal for image classifiers.

- Conv2D is a layer that performs 2D convolution. What it does is scan the image and “memorize the relevant information” so to speak.
    - 16 is the number of filters the layer has.
    - (3, 3) is the size of a filter in pixels.
    - 1 is the stride, meaning the filters move 1 pixel.
    - activation = ‘relu’: ReLU stands for Rectified Linear Unit and takes the output of the convolution and passes it through a filter that makes all negative values 0
    - input_shape: this is used only on the first layer. It tells the model that it needs to receive images of 256px-by-256px size and 3 color channels.
- MaxPooling2D reduces the width and height of the input volume by taking the maximum value of the input.
- Flatten converts the 3D output (width, height, channels) of the convolutions and converts them into a one-dimensional array.
- Dense creates a fully connected layer of x neurons:
    - First value represents the number of neurons the layer has.
    - The softmax activation outputs a probability distribution array of the classes.
- Model.Compile:
    - Optimizer = ‘adam’: adam is an adaptive learning rate optimization algorithm that's widely used for training deep learning models.
    - Loss = ‘sparse_categorical_crossentropy’: This loss function is used for classification problems where the target variable is an integer.
    - Metrics is the metric used to evaluate the model.

g. Train

Early_stopping: monitors the training and stops when it becomes redundant.

- Monitor = ‘accuracy’: checks the accuracy
- Patience = 2: after 2 consecutive trainings with the same accuracy, the training stops

Model.fit: starts the training

- Train: takes the training data to train on.
- Epochs = 20: number of trainings
- Validation_data = val: takes the validation data to validate training
- Callbacks = [early_stopping] tells the training to use the early_stopping function

```
# Early stopping callback
early_stopping = EarlyStopping(monitor = 'accuracy', patience = 2, restore_best_weights = True)

# Train
hist = model.fit(train, epochs=20, validation_data=val, callbacks=[early_stopping])
```

h. Evaluate and save the model:
The model takes the test data and evaluates its performance. After, it saves the model.

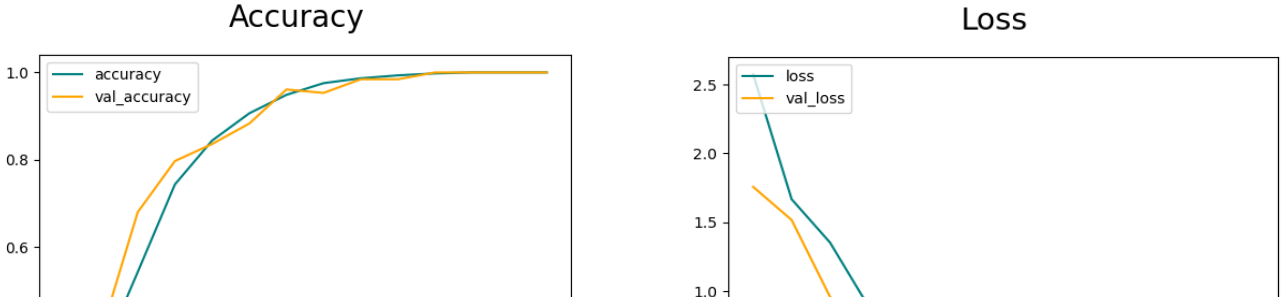
```
# Evaluate
pre = Precision()
re = Recall()
acc = CategoricalAccuracy()

for batch in test.as_numpy_iterator():
    X, y = batch
    yhat = model.predict(X)
    yhat = np.argmax(yhat, axis=1)
    pre.update_state(y, yhat)
    re.update_state(y, yhat)
    acc.update_state(y, yhat)

print(f'Precision: {pre.result().numpy()}, Recall: {re.result().numpy()}, Accuracy:
{acc.result().numpy()}')

#Save model
model.save(os.path.join('models','imageclassifier.h5'))
```

6. Graphs:
The following graphs show us the performance of the model.



Let’s start to interpret them. The blue line is the accuracy or loss of the training, and the yellow one is the validation for both. Now let’s take a look at the accuracy graph.

- Both the accuracy and validation accuracy have a rapid increase, showing the efficiency of the model’s training.
- Good performance is indicated by the convergence of the 2 lines at later stages of the training.

Same interpretation is valid for the loss graph, starting with high loss values that steeply decline over training.

By looking at the graphs, there is little to no evidence of overfitting, as the validation does not stagnate or decrease for neither accuracy nor loss as they improve.

7. Improvements:
    a. A larger dataset with high quality images, made specifically for training models.
    b. Data augmentation can be used to make the model more efficient on low quality or distorted images. Such augmentations are:
        - Blurring
        - Rotation
        - Noise
        - Brightness and contrast adjustment
        - Rotation
    c. Develop this image classifier into a full-blown object recognition model.

8. Documentation and help:
    - Many thanks to Nicholas Renotte for the code skeleton that I improved and modified to make this model in the classifier I needed. Here is the link

to the video: Build a Deep CNN Image Classifier with ANY Images (youtube.com)

- Also to KGP Talkie for the tutorial on how to make TensorFlow2 work on GPU for native Windows users: How to Install TensorFlow GPU on Windows 11 | GPU Setup on Windows | TensorFlow GPU Setup - YouTube
- TensorFlow documentation: API Documentation | TensorFlow v2.16.1
- CV2 documentation: OpenCV: OpenCV-Python Tutorials
- Python OS: os — Miscellaneous operating system interfaces — Python 3.12.3 documentation

9. Conclusion:
   This is a good start for a true object recognition model, made using Python and TensorFlow, in a protected Anaconda environment. The model is efficient and has a usable graphical interface to make things easier for the user. In this project we went over the code, understanding how it works, we've looked at graphs to understand how to interpret them to recognize and fix errors, if they appear and we've learned how to improve it.