AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Color detection using deep learning

Programmer: Maciej Skrobisz
Tester: Michał Janik

1. Project Overview

The main goal of this project is to create a Machine Learning system capable of classifying the dominant color in images. The project focuses on "Edge AI" – meaning the model is designed to be lightweight and efficient enough to run on microcontrollers or small devices. I used a custom Convolutional Neural Network (CNN) and converted it to TensorFlow Lite with INT8 quantization.

2. Project Structure

The code is divided into several modules to keep the logic clean:

- **config.py**: Holds global settings like image size (180x180), batch size (32), and paths to datasets or models.
- **data_cleaning.py**: A script I wrote to clean the dataset. It converts PNG files to JPG and deletes corrupted images and files that could crash the training script.
- **Dataset.py**: Handles loading images from folders and splitting them into training (80%) and validation (20%) sets.
- **model.py**: Contains the architecture of my neural network (CNN).
- **visualize.py**: A helper script to show a batch of images and their labels. I use it to make sure the data is loaded correctly.
- **train.py**: The main script for training. It includes logic for handling unbalanced classes - if there are too many photos of one color.
- **predict.py**: Used for testing the model on new images. It prints the result and confidence score.
- **convert.tflite.py**: Converts the trained Keras model into a .tflite file optimized for hardware (INT8 quantization).

3. Algorithm Description

3.1. Dataset Overview

The dataset was collected manually by downloading images from the internet. It consists of **10 distinct color classes**.

- **Size:** The dataset is relatively small, containing approximately **20 to 30 JPG images per color**.
- **Challenge:** Because the dataset is small, the model relies heavily on Data Augmentation (described below) to learn effectively and avoid overfitting.

3.2. Data Preprocessing

Before training, I use data_cleaning.py to ensure the data is valid.

1. **Format Check**: It converts all images to JPG.
2. **Integrity Check**: It uses the PIL library and TensorFlow's read_file to find and remove broken files. This prevents runtime errors during the actual training.

3.3. Network Architecture (CNN)

I decided to build a custom CNN instead of using a pre-trained giant like ResNet because speed was a priority. The architecture (in model.py) includes:

- **Data Augmentation**: Layers that randomly flip, rotate, and zoom images to prevent overfitting.
- **Convolutional Layers**: Three blocks of Conv2D + MaxPooling to extract features.
- **GlobalAveragePooling2D**: I used this instead of a Flatten layer. It drastically reduces the model size and parameter count.
- **Dropout**: A regularization layer to help the model generalize better.

3.4. Training Process

The training uses the Adam optimizer and Sparse Categorical Crossentropy.

- **Class Weights**: To solve the problem of unbalanced data, I calculate class weights. I also added a manual penalty for the 'black' class (reducing its weight) because it often acts as background noise.
- **Checkpoints**: The code saves the model only when the validation accuracy improves.

3.5. Quantization

To make the model ready for microcontrollers, I implemented Post-training Quantization in convert.tflite.py. It uses a "Representative Dataset" generator to calibrate the model and converts all weights to 8-bit integers (INT8).

4. Comparison: My Approach vs. Alternatives

4.1. vs. Classical Machine Learning (Scikit-learn)

I compared my Deep Learning implementation with a classic Machine Learning approach (like SVM or KNN available in Scikit-learn).

- **Feature Extraction**: Scikit-learn models require manual feature engineering (like creating color histograms). My CNN learns these features automatically from the pixels.
- **Invariance**: Classic algorithms struggle if the object moves or rotates in the picture. My CNN uses Pooling layers and Data Augmentation, so it handles position changes much better.
- **Size and Deployment**: A Random Forest model trained on raw pixels would be huge. My CNN using Global Average Pooling is small, and after TFLite conversion, it is perfect for embedded devices.

4.2. vs. Pre-trained Deep Learning Models (e.g., MobileNetV2)

I also considered using Transfer Learning with popular models like MobileNetV2 or ResNet50, but I decided against it for several reasons:

- **Overkill**: MobileNet is trained on ImageNet to recognize 1000 complex classes (breeds of dogs, types of cars). For a simple task like color detection, using such a deep network is unnecessary.
- **Model Size**: Even the smallest MobileNet weighs several megabytes. My custom model weighs only a few kilobytes. This makes a huge difference when deploying to a microcontroller with very limited flash memory.
- **Computation Speed**: My model has significantly fewer parameters. On weak hardware, this translates to faster inference times and lower battery consumption.

Conclusion: My custom CNN is the "middle ground" – it is smarter than Scikit-learn but much lighter and faster than standard pre-trained Deep Learning models.

5. Installation

To run this project, you need to set up a Python virtual environment and install the required dependencies. The `.venv` folder is not included in the repository to keep the project lightweight.

Tensorflow library in my project supports python 3.12 or older!

Steps to set up the environment from scratch:

1. Open your terminal in the project folder.
2. Run the following commands to create the environment and install libraries:

```
python -m venv .venv
.\.venv\Scripts\Activate
pip install tensorflow numpy matplotlib scikit-learn pillow pyscaffold
```

Adding the dataset

Unzip the dataset folder and add it to the Color_detection folder.
You can also use your own dataset of .jpg files.

6. How to Run the Code

Here are the commands to execute the main parts of the project:

To train the model:

```
python -m src.color_detection.train
```

To predict a color from a specific image:

```
python -m src.color_detection.predict "C:\Path\To\Your\Photo.jpg"
```

To run other utility modules: You can execute other scripts (like `visualize`, `data_cleaning`, or `convert_tflite`) by following the same syntax pattern:

```
python -m src.color_detection.<module_name>
```

TESTS

Michał Janik

1. Unit Tests

The color_detection project includes a set of unit tests covering all key modules. These tests primarily ensure the correct structure and basic functionality of the project components.

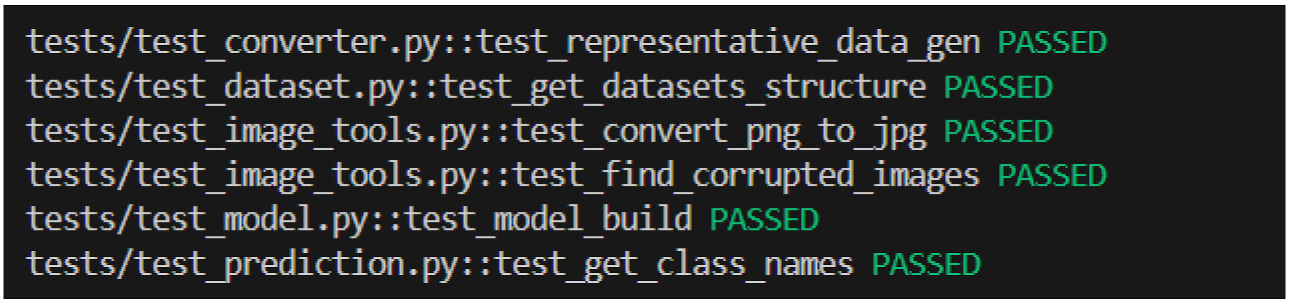
To run the tests, follow these steps:

```
# Create a virtual environment (Python < 11)
python -m venv .venv

# Activate the environment
.\.venv\Scripts\Activate

# Install dependencies
pip install tensorflow numpy matplotlib scikit-learn pillow pyscaffold pytest-cov

# Run all tests
pytest
```

1.1 Main objectives of the tests

- Verify the data pipeline – datasets are iterable, images are correctly converted, and corrupted files are detected.
- Check model construction – the CNN has the correct output shape and the forward pass runs without errors.
- Validate utility functions – image conversion and corrupted file detection behave as expected.
- Verify prediction configuration – the list of class names is returned correctly.
- Prepare data for TFLite – the representative data generator returns batches with the correct shape and type.

1.2 Example tests



```
tests/test_converter.py::test_representative_data_gen PASSED
tests/test_dataset.py::test_get_datasets_structure PASSED
tests/test_image_tools.py::test_convert_png_to_jpg PASSED
tests/test_image_tools.py::test_find_corrupted_images PASSED
tests/test_model.py::test_model_build PASSED
tests/test_prediction.py::test_get_class_names PASSED
```

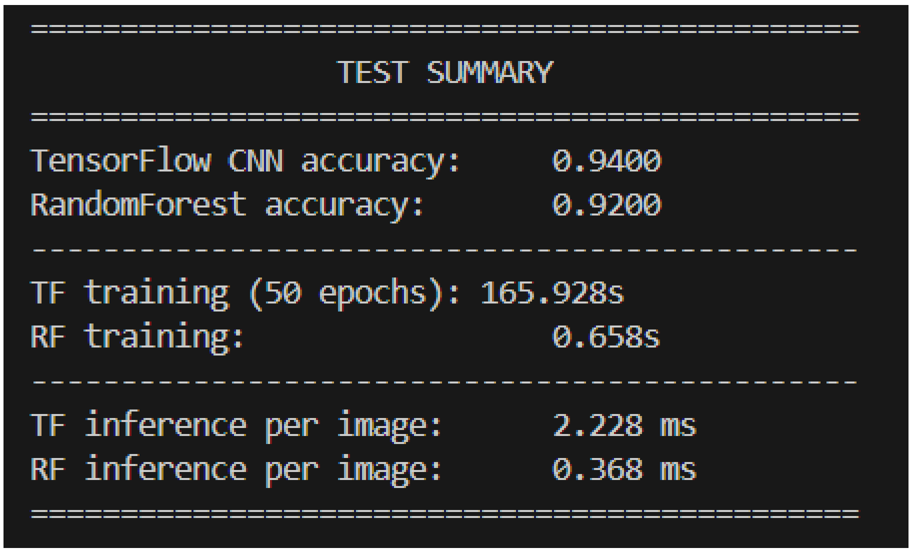
1.3 Conclusions

- The tests are fast to run and suitable for CI/CD pipelines.
- They ensure basic project integrity, protecting against errors in the data pipeline, model, and utility functions.
- They provide a solid baseline for further development and more extensive integration or performance testing.

2. Comparison of CNN (TensorFlow) and RandomForest (scikit-learn)

Two models were tested on the same color image dataset (10 classes):
a convolutional neural network (CNN) and a RandomForest (RF) classifier.

2.1 Results



```
==============================================
                 TEST SUMMARY
==============================================
TensorFlow CNN accuracy:     0.9400
RandomForest accuracy:       0.9200
----------------------------------------------
TF training (50 epochs): 165.928s
RF training:                 0.658s
----------------------------------------------
TF inference per image:      2.228 ms
RF inference per image:      0.368 ms
==============================================
```

CNN (TensorFlow)

- Accuracy: 94%
- Training: 166 s (50 epochs)
- Prediction: ~2.2 ms / image

CNN achieved the highest accuracy, learning visual structures and features from the images.

RandomForest (scikit-learn)

- Accuracy: 92%
- Training: 0.66 s
- Prediction: ~0.37 ms / image

RF performed surprisingly well because the task is mainly color-based, even though it does not analyze shapes or patterns.

2.2 Interpretation

- RandomForest works well for this dataset because images differ mostly in color, and RF captures simple pixel statistics.
- CNN outperforms RF by understanding image structure and generalizing features more effectively.
- RF serves as a strong baseline, but is not suitable for more complex computer vision tasks.

2.3 Conclusions

1. CNN is the best choice for image tasks, achieving the highest accuracy.
2. RandomForest is fast and effective for simple color-based tasks.
3. The 96% vs 90% difference shows that CNN learns visual features that RF cannot capture.
4. The comparison demonstrates the real advantage of deep learning, while classical ML can still provide a useful reference baseline.