Universitatea POLITEHNICA din București

Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Aplicație de detecție și identificare a semnelor de circulație

Proiect de Diplomă

Prezentat ca cerință parțială pentru obținerea

titlului de *Inginer*

în domeniul *Electronică și Telecomunicații*

programul de studii *Tehnologii și Sisteme de Telecomunicații*

Conducător științific

Conf.Dr.Ing. Ionuţ PIRNOG

Absolvent

Popescu Ervin-Adrian

Anul 2022

Universitatea “Politehnica” din Bucureşti **Anexa 1**
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei
Program de studiu **TST**

TEMA PROIECTULUI DE DIPLOMĂ

a studentului **POPESCU A. Ervin-Adrian, 444C**

1. Titlul temei: Aplicație de detecție și identificare a semnelor de circulație

2. Descrierea temei și a contribuției personale a studentului (în afara părții de documentare):
Se va implementa o aplicație de detecție și identificare a semnelor de circulație în imagini și secvențe video. Aplicația se poate implementa în Matlab, C , Python, Java. Se pot folosi librării specifice și algoritmi dedicați prelucrării imaginilor/video: OpenCV, YOLOv4, Pytorch, Tensorflow, Python Tesseract, etc..

3. Discipline necesare pt. proiect:
PDS; POO; TCSM

4. Data înregistrării temei: 2023-02-03 18:43:47

Conducător(i) lucrare,
Conf.Dr.Ing. Ionuţ PIRNOG

Student,
POPESCU A. Ervin-Adrian

Director departament,
Conf. dr. ing. Șerban OBREJA

Decan,
Prof. dr. ing. Mihnea UDREA

Cod Validare: **6434b356e1**

Declaraţie de onestitate academică

Prin prezenta declare că lucrarea cu titlul *Aplicație de detecție și identificare a semnelor de circulație*, prezentată în cadrul Facultăţii de Electronică, Telecomunicaţii şi Tehnologia Informaţiei a Universităţii "Politehnica" din Bucureşti ca cerinţă parţială pentru obţinerea titlului de *Inginer* în domeniul Inginerie Electronică şi Telecomunicaţii/ Calculatoare şi Tehnologia Informaţiei, programul de studii *Tehnologii și Sisteme de Telecomunicații* este scrisă de mine şi nu a mai fost prezentată niciodată la o facultate sau instituţie de învăţământ superior din ţară sau străinătate. Declare că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referinţe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi fac referinţă la sursă. Reformularea în cuvinte proprii a textelor scrise de către alţi autori face referinţă la sursă. Înţeleg că plagiatul constituie infracţiune şi se sancţionează conform legilor în vigoare. Declare că toate rezultatele simulărilor, experimentelor şi măsurătorilor pe care le prezint ca fiind făcute de mine, precum şi metodele prin care au fost obţinute, sunt reale şi provin din respectivele simulări, experimente şi măsurători. Înţeleg că falsificarea datelor şi rezultatelor constituie fraudă şi se sancţionează conform regulamentelor în vigoare.

Bucureşti, Iulie 2022.

Absolvent: Popescu Ervin-Adrian

....

Cuprins

Lista figurilor

Lista tabelelor

Lista acronimelor

CNN: Convolutional Neural Network

Capitolul 1

Introducere

Semnele de circulație joacă un rol vital în menținerea siguranței rutiere și a fluidității traficului. Detectarea și identificarea acestor semne poate fi o sarcină dificilă și de multe ori costisitoare, deoarece necesită o analiză vizuală atentă a imaginilor și secvențelor video. În această lucrare de diploma, se va propune o aplicație de detecție și identificare a semnelor de circulație utilizând diverse librării și algoritmi dedicați prelucrării imaginilor/video, cum ar fi OpenCV, YOLOv4, Pytorch, TensorFlow și Python Tesseract. Această aplicație va permite o detectare precisă și rapidă a semnelor de circulație, îmbunătățind astfel siguranța rutieră și eficiența traficului.

Contribuția personală a acestui proiect constă în implementarea și optimizarea unui sistem de recunoaștere a semnelor de circulație în imagini și secvențe video. Proiectul va fi implementat în Python, utilizând diferite librării și algoritmi specifici de prelucrare a imaginilor și algoritmici de machine learning. Acest sistem va fi capabil să detecteze și să identifice semnele de circulație cu o precizie ridicată, prin utilizarea unor modele de învățare profundă, cum ar fi rețele neuronale convoluționale (CNN). În plus, aplicația va fi optimizată pentru a asigura o viteză de procesare ridicată, ceea ce va permite utilizarea sa în timp real în diferite situații de trafic.

În concluzie, această lucrare de diplomă va prezenta o aplicație inovatoare de detecție și identificare a semnelor de circulație, care va îmbunătăți siguranța rutieră și eficiența traficului. Prin implementarea și optimizarea unui sistem de recunoaștere a semnelor de circulație în imagini și secvențe video, acest proiect va reprezenta o contribuție semnificativă la domeniul prelucrării imaginilor și al recunoașterii de modele.

Capitolul 2

Descrierea aplicației

În A.1 avem 139 .

Anexa A

Cod sursă

A.1: Main project file

```python
import argparse
import os
from datetime import datetime
import sys
from timeit import default_timer as timer

from modules.config import logger

logger.info("$BOLD$GREENStart: $BLUE%s", datetime.now().strftime("%d/%m/%Y, %T"))

import pandas as pd

start = timer()
from keras.applications import (
    VGG16,
    VGG19,
    # MobileNetV3Large,
    # MobileNetV3Small,
    ResNet50,
    ResNet50V2,
    # ResNet152,
    # ResNet152V2,
)
stop = timer()
logger.info(
    f"$BOLD$BLUEImporting `keras.applications` took $RED{stop-start:.2f}$BLUE seconds"
)
from matplotlib import pyplot as plt

from modules.config import input_videos_filenames, output_path
from modules.custom_model import CustomModel


class HelpAction(argparse.Action):
    def __call__(self, parser, *args, **kwargs):
        parser.print_help()
        stop = timer()
        logger.info(
            f"$BOLD$BLUEProgram execution took $RED{stop-start:.2f}$BLUE seconds"
        )
        sys.exit(0)


def main():
    parser = argparse.ArgumentParser(
        description="Program that trains and tests a model for road sign detection",
        add_help=False,
    )
    parser.add_argument("-h", "--help", nargs=0, action=HelpAction)
    parser.add_argument(
        "--test", action="store_true", help="test model on both images and videos"
    )
    parser.add_argument(
        "--test-images", action="store_true", help="test model on images"
    )
    parser.add_argument(
        "--test-videos", action="store_true", help="test model on videos"
    )
    parser.add_argument(
        "--lite", action="store_true", help="convert HDF5 model to `tf.lite` model"
    )
    args = parser.parse_args()
    if args.test:
        args.test_images = True
        args.test_videos = True
    models = {
        # "MobilenetV3large": MobileNetV3Large,
        # "MobilenetV3small": MobileNetV3Small,
        # "resnet152": ResNet152,
        # "resnet152v2": ResNet152V2,
        "resnet50": ResNet50,
        "resnet50v2": ResNet50V2,
        "vgg16": VGG16,
        "vgg19": VGG19,
    }
    model_benchmarks = {
        "model_name": [],
        "num_model_params": [],
        "label_validation_accuracy": [],
    }
    for name, model in models.items():
        logger.info(f"$BLUE$BOLDBase model: $RED{name}$RESET")

        saved_model_path: str = os.path.join(output_path, name, "model.h5")
        trained: bool = os.path.exists(saved_model_path)

        custom_model_instance = CustomModel(
            base_model_function=model, trained=trained, lite_model_required=args.lite
        )

        if args.test_images:
            custom_model_instance.test_model_images(include_random=False)
        if args.test_videos:
            for input_filename in input_videos_filenames:
                custom_model_instance.test_model_videos(input_video_fn=input_filename)

        custom_model = custom_model_instance.model
        history = custom_model_instance.history

        model_benchmarks["model_name"].append(name)
        model_benchmarks["num_model_params"].append(custom_model.count_params())
        model_benchmarks["label_validation_accuracy"].append(
            float(history["val_class_label_accuracy"][-1]) * 100
        )

    benchmark_df = pd.DataFrame(model_benchmarks)
    benchmark_df.sort_values("label_validation_accuracy", inplace=True)
    benchmark_df["label_validation_accuracy"] = benchmark_df[
        "label_validation_accuracy"
    ].transform(lambda x: f"{x:.2f}%")
    print(benchmark_df.keys())
    benchmark_df.to_csv(output_path + "/benchmark_df.csv", index=False)

    # save plot to file
    markers = [".", ",", "o", "v", "^", "<", ">", "*", "+", "|", "_"]
    plt.figure(figsize=(10, 8))
    for row in benchmark_df.itertuples():
        plt.scatter(
            x=row.num_model_params,
            y=row.label_validation_accuracy,
            # y=row.random_accuracy,
            label=row.model_name,
            marker=markers[row.Index],
            s=150,
            linewidths=2,
        )
    plt.xscale("log")
    plt.xlabel("Number of Parameters in Model")
    plt.ylabel("Validation Accuracy after 10 Epochs")
    plt.title("Accuracy vs Model Size")
    plt.legend(bbox_to_anchor=(1, 1), loc="upper left")
    plt.tight_layout()
    plt.savefig(output_path + "/plot.png")


if __name__ == "__main__":
    main()
    stop = timer()
    logger.info(f"$BOLD$BLUEProgram execution took $RED{stop-start:.2f}$BLUE seconds")
```

```python
import os
import pathlib
from logging import Logger, getLogger

from modules.logger import init_log

RED = "\033[1;31m"
GREEN = "\033[1;32m"
BLUE = "\033[1;34m"
RESET = "\033[0m"

main_file_path = pathlib.Path(__file__).parent.parent
input_path = os.path.join(main_file_path, "input")
output_path = os.path.join(main_file_path, "output")


# Define the location of the dataset
training_data_dir = os.path.join(input_path, "images", "Training")
test_data_dir = os.path.join(input_path, "images", "Test")
input_videos_dir = os.path.join(input_path, "videos")
input_videos_filenames = os.listdir(input_videos_dir)
labels_path = os.path.join(input_path, "labels.json")

# Define the image size and number of classes
IMG_SIZE = (64, 64)
VIDEO_SIZE = (1024, 1024)
NUM_CLASSES = 43
INIT_LR = 1e-2
NUM_EPOCHS = 5
BATCH_SIZE = 64

logger: Logger = getLogger("main.py")
init_log(
    logger,
    mode="a",
    log_path=os.path.join(pathlib.Path(__file__).parent.parent.resolve(), "main.log"),
    format_str="%(message)s",
    log_level="INFO",
)
```

A.2: Config file

```python
import os
from typing import Tuple

import numpy as np
import pandas as pd
from keras.utils import img_to_array, load_img
from modules.config import IMG_SIZE, NUM_CLASSES
from sklearn.preprocessing import LabelBinarizer


# Function to load the images and labels from the dataset
def load_training_data(data_dir):
    images = []
    labels = []
    bboxes = []
    image_paths = []

    # loop over all 42 classes
    for c in range(0, NUM_CLASSES):
        prefix = os.path.join(data_dir, format(c, "05d"))  # subdirectory for class
        with open(os.path.join(prefix, "GT-" + format(c, "05d") + ".csv")) as gtFile:
            annotations = pd.read_csv(gtFile, sep=";")
            # loop over all images in current annotations file
            for _, row in annotations.iterrows():
                impath = os.path.join(prefix, row[0])
                image = img_to_array(load_img(impath, target_size=IMG_SIZE))
                label = row[7]
                w = int(row[1])
                h = int(row[2])
                xmin = int(row[3]) / w
                ymin = int(row[6]) / h
                xmax = int(row[5]) / w
                ymax = int(row[4]) / h
                images.append(image)  # the 1st column is the filename
                labels.append(label)  # the 8th column is the label
                bboxes.append((xmin, ymin, xmax, ymax))
                image_paths.append(impath)

    # one-hot encoding
    lb = LabelBinarizer()
    labels = lb.fit_transform(labels)

    # normalize -> from [0-255] to [0-1]
    images = np.array(images, dtype="float32") / 255.0

    # convert to np arrays
    labels = np.array(labels)
    bboxes = np.array(bboxes, dtype="float32")
    image_paths = np.array(image_paths)

    return images, labels, bboxes, image_paths



def load_test_data(data_dir):
    images = []
    bboxes = []
    image_paths = []

    with open(os.path.join(data_dir, "GT-final_test.test.csv")) as csvFile:
        annotations = pd.read_csv(csvFile, sep=";")
        # loop over all images in current annotations file
        for _, row in annotations.iterrows():
            impath = os.path.abspath(os.path.join(data_dir, row[0]))
            image = img_to_array(load_img(impath, target_size=IMG_SIZE))
            w = int(row[1])
            h = int(row[2])
            xmin = int(row[3]) / w
            ymin = int(row[6]) / h
            xmax = int(row[5]) / w
            ymax = int(row[4]) / h
            images.append(image)  # the 1st column is the filename
            bboxes.append((xmin, ymin, xmax, ymax))
            image_paths.append(impath)

    # normalize -> from [0-255] to [0-1]
    images = np.array(images, dtype="float32") / 255.0
    bboxes = np.array(bboxes, dtype="float32")
    image_paths = np.array(image_paths)

    return images, bboxes, image_paths
```

A.3: Data loading module

```python
import gzip
import json
import math
import os
import pathlib
import pickle
import random
from timeit import default_timer as timer

import cv2
import ffmpeg
import jsonpickle
import numpy as np
import pandas as pd
from keras.callbacks import History
from keras.layers import Dense, Dropout, Flatten, Input
from keras.models import Model, load_model

# from keras.optimizers.adam import Adam
from keras.optimizers.adamw import AdamW
from keras.utils import img_to_array, load_img
from keras.utils.vis_utils import plot_model
from PIL import Image, ImageDraw, ImageFont
from sklearn.model_selection import train_test_split
from tensorflow import lite

from modules.config import (
    BATCH_SIZE,
    IMG_SIZE,
    INIT_LR,
    NUM_CLASSES,
    NUM_EPOCHS,
    input_path,
    input_videos_dir,
    labels_path,
    logger,
    output_path,
    test_data_dir,
    training_data_dir,
)
from modules.load_data import load_test_data, load_training_data
from modules.videowriter import vidwrite


class LiteModel:
    def __init__(self, interpreter):
        self.interpreter: lite.Interpreter = interpreter
        self.interpreter.allocate_tensors()
        input_det = self.interpreter.get_input_details()[0]
        output_det = self.interpreter.get_output_details()[1]
        self.input_index = input_det["index"]
        self.output_index = output_det["index"]
        self.input_shape = input_det["shape"]
        self.output_shape = output_det["shape"]
        self.input_dtype = input_det["dtype"]
        self.output_dtype = output_det["dtype"]

    def predict(self, inp: np.ndarray):
        inp = inp.astype(self.input_dtype)
        count = inp.shape[0]
        out = np.zeros((count, self.output_shape[1]), dtype=self.output_dtype)
        for i in range(count):
            self.interpreter.set_tensor(self.input_index, inp[i : i + 1])
            self.interpreter.invoke()
            out[i] = self.interpreter.get_tensor(self.output_index)[0]
        return out


class CustomModel:
    def __init__(
        self, base_model_function: Model, trained: bool, lite_model_required: bool
    ) -> None:
        self.model: Model = None
        self.history: dict = None
        self.lite_model_required = lite_model_required
        self.tflite_model = None
        input_shape = IMG_SIZE + tuple([3])
        input_tensor = Input(shape=IMG_SIZE + tuple([3]))
        base_model_args = dict(
            input_shape=input_shape,
            weights="imagenet",
            include_top=False,
            input_tensor=input_tensor,
        )
        self.base_model: Model = base_model_function(**base_model_args)
        self.saved_model_path = os.path.join(
            output_path, self.base_model.name, "model.h5"
        )
        self.history_path = os.path.join(
            output_path, self.base_model.name, "training_history.json"
        )
        self.scores_path = os.path.join(output_path, self.base_model.name, "scores.txt")
```

```python
        self.lb_path = os.path.join(output_path, self.base_model.name, "lb.pickle")
        self.predicted_labels_path = os.path.join(
            output_path, self.base_model.name, "predicted_labels.pickle"
        )
        self.accuracies_path = os.path.join(
            output_path, self.base_model.name, "accuracies.txt"
        )
        if not trained:
            self.define_model()
            self.history = self.train().history
        else:
            self.model = load_model(self.saved_model_path)
            with open(self.history_path, "r") as f:
                self.history = jsonpickle.decode(f.read())
        if self.lite_model_required:
            lite_model_path = pathlib.Path(self.saved_model_path).with_suffix(".tflite")
            if os.path.exists(lite_model_path):
                with open(lite_model_path, "rb") as f:
                    self.tflite_model = f.read()
                    self.tflite_model_instance = LiteModel(
                        lite.Interpreter(model_path=str(lite_model_path))
                    )
            else:
                try:
                    self.tflite_model = lite.TFLiteConverter.from_keras_model(
                        self.model
                    ).convert()
                    self.tflite_model_instance = LiteModel(
                        lite.Interpreter(model_content=self.tflite_model)
                    )
                    with open(lite_model_path, "wb") as f:
                        f.write(self.tflite_model)
                except Exception:
                    logger.error(
                        f"$REDCould not convert model to `tf.lite` model$RESET",
                        exc_info=1,
                    )
                    exit(0)

    def train(self) -> History:
        # Load the data
        images, labels, bboxes, _ = load_training_data(training_data_dir)
        split = train_test_split(images, labels, bboxes, test_size=0.2, random_state=12)

        (x_train, x_validation) = split[0:2]
        (y_train, y_validation) = split[2:4]
        (bboxes_train, bboxes_validation) = split[4:6]

        train_targets = {"class_label": y_train, "bounding_box": bboxes_train}
        validation_targets = {
            "class_label": y_validation,
            "bounding_box": bboxes_validation,
        }

        # self.model.summary()

        if not os.path.exists(f"../../figuri/{self.base_model.name}/model_plot.png"):
            plot_model(
                self.model,
                to_file=f"../../figuri/{self.base_model.name}/model_plot.png",
                dpi=192,
                show_shapes=True,
                show_layer_names=True,
                show_layer_activations=True,
                show_trainable=True,
            )

        logger.info(f"\t$BLUEstarting training...$RESET")
        start = timer()
        # Train the model
        history = self.model.fit(
            x_train,
            train_targets,
            validation_data=(x_validation, validation_targets),
            epochs=NUM_EPOCHS,
            batch_size=BATCH_SIZE,
            verbose=0,
        )
        logger.info(f"\t$BLUEending training...$RESET")
        logger.info(f"\t$BLUEtraining took $RED{timer()-start:.6f} $BLUEseconds$RESET")
        self.model.save(self.saved_model_path)
        with open(self.history_path, "w") as f:
            f.write(jsonpickle.encode(history.history))
        return history

    def test_model_images(self, include_random: bool = False):
        start = timer()
        images, bboxes, image_paths = load_test_data(test_data_dir)
        if os.path.exists(self.predicted_labels_path):
            with open(self.predicted_labels_path, "rb") as f:
                predicted_labels = pickle.load(f)
        else:
            logger.info(f"\t$BLUEpredicting labels for images...$RESET")
            predicted_labels = self.model.predict(
                images,
                batch_size=BATCH_SIZE,
                verbose=0,
            )[1]
            f"\t$BLUEpredicting labels took $RED{timer()-start:.6f} $BLUEseconds$RESET"
            with open(self.predicted_labels_path, "wb") as f:
                pickle.dump(predicted_labels, f)
        predicted_labels = np.array(predicted_labels)
        with open(os.path.join(test_data_dir, "Test.csv")) as f:
            correct_labels = pd.read_csv(f, sep=",")["ClassId"].to_numpy(dtype="uint32")
        with open(labels_path, "r") as f:
            labels_json = json.load(f)

        testTargets = {"class_label": predicted_labels, "bounding_box": bboxes}
        metrics_names: list[str] = self.model.metrics_names
        correct = 0

        if not os.path.exists(self.scores_path):
            logger.info(
                f"\t$BLUEevaluating model $GREEN{self.base_model.name} $BLUEand saving scores...$RESET"
            )
            scores = self.model.evaluate(
                images,
                testTargets,
                batch_size=BATCH_SIZE,
                verbose=0,
            )

            for image_path in image_paths:
                index = np.where(image_paths == image_path)[0][0]
                image = load_img(image_path, target_size=IMG_SIZE)
                image = img_to_array(image) / 255.0
                image = np.expand_dims(image, axis=0)

                # # finding class label with highest pred. probability
                i = np.argmax(predicted_labels[index], axis=0)
                predicted_label = labels_json[str(i)]
                correct_label = labels_json[str(correct_labels[index])]

                if predicted_label == correct_label:
                    correct += 1

            test_acc = f"Test accuracy: {correct/len(images)*100:.2f}%\n"
            with open(self.scores_path, "w") as f:
                for name, score in zip(metrics_names, scores):
                    name = name.split("_")
                    name[0] = name[0].capitalize()
                    joined_name = " ".join(name)
                    if "Loss" in joined_name or "loss" in joined_name:
                        line = "{}: {:.2f}\n".format(joined_name, score)
                    else:
                        line = "{}: {:.2f}%\n".format(joined_name, score * 100)
                    f.write(line)
                f.write(test_acc)
        if include_random:
            for i in range(100):
                correct = 0
                random.seed(random.random() * 50)
                random_choices = random.choices(
                    image_paths, k=int(len(image_paths) / 100)
                )
                for image_path in random_choices:
                    index = np.where(image_paths == image_path)[0][0]
                    i = np.argmax(predicted_labels[index], axis=0)
                    predicted_label = labels_json[str(i)]
                    correct_label = labels_json[str(correct_labels[index])]
                    if predicted_label == correct_label:
                        correct += 1
                random_acc = f"{correct/len(random_choices)*100:.2f}\n"
                with open(self.accuracies_path, "a") as f:
                    f.write(random_acc)
        f"\t$BLUEtesting images took $RED{timer()-start:.6f} $BLUEseconds$RESET"

    def test_model_videos(self, input_video_fn: str):
        logger.info(
            f"$BOLD$COLOR==> $BOLD$BLUEprocessing input video $GREEN{input_video_fn}$RESET"
        )
        input_video_path = os.path.join(input_videos_dir, input_video_fn)
        output_video_path = os.path.join(
            output_path,
            self.base_model.name,
            f'output-{input_video_fn.replace(".mp4","")}.mp4',
        )
        input_frames_path = os.path.join(
            input_path,
            "frames",
            self.base_model.name,
            f'{input_video_fn.replace(".mp4","")}.frames.npy.gz',
        )
        with open(labels_path, "r") as f:
            labels_json = json.load(f)
        video_stream = ffmpeg.probe(input_video_path)["streams"][0]
        ns = {"__builtins__": None}
        # frame_height = int(video_stream["height"])
        # frame_width = int(video_stream["width"])
        fps = math.ceil(float(eval(video_stream["avg_frame_rate"], ns)))
        # pix_fmt = video_stream["pix_fmt"]
        pix_fmt = "rgb24"
        ffmpeg_args = {
            "hide_banner": None,
            "loglevel": "quiet",
            "v": "quiet",
            "nostats": None,
        }
        if not os.path.exists(pathlib.Path(input_frames_path).parent):
            os.mkdir(pathlib.Path(input_frames_path).parent)
        if not os.path.exists(pathlib.Path(output_video_path).parent):
            os.mkdir(pathlib.Path(output_video_path).parent)
        generated_frames = os.path.exists(input_frames_path)
        generated_video = os.path.exists(output_video_path)
        if generated_frames:
            if generated_video:
                logger.info(
                    f"\t$BLUEalready generated frames and video for video $GREEN{input_video_fn}$RESET"
                )
            else:
                logger.info(
                    f"\t$BLUEreading generated frames for video $GREEN{input_video_fn}$RESET"
                )
                with gzip.GzipFile(input_frames_path, "r") as f:
                    resized_frames = np.load(f)
                logger.info(f"writing output video $GREEN{output_video_path}$RESET")
                vidwrite(
                    output_video_path,
                    resized_frames,
                    fps=fps // 4,
                    in_pix_fmt=pix_fmt,
                    input_args=ffmpeg_args,
                    output_args={
                        i: ffmpeg_args[i] for i in ffmpeg_args if i != "hide_banner"
                    },
                )
                return
        else:
```

```python
        logger.info(f"\t$BLUEgenerating frames")
        vidcap = cv2.VideoCapture(input_video_path)
        resized_frames = []
        frames = []
        count = 0
        while vidcap.isOpened():
            success, frame = vidcap.read()
            if success:
                img = Image.fromarray(frame)
                frames.append(img)
                resized_frame = cv2.resize(frame, (64, 64))
                resized_frames.append(resized_frame)
                count += fps
                vidcap.set(cv2.CAP_PROP_POS_FRAMES, count)
            else:
                vidcap.release()
                break
        resized_frames = np.array(resized_frames)
        start = timer()
        if not self.lite_model_required:
            label_predictions = self.model.predict(
                resized_frames, verbose=0, batch_size=BATCH_SIZE
            )
        else:
            label_predictions = self.tflite_model_instance.predict(resized_frames)
        logger.info(
            f"\t$BLUEpredicting labels took $RED{timer()-start:.6f} $BLUEseconds$RESET"
        )
        start = timer()
        for index, frame in zip(range(resized_frames.shape[0]), frames):
            label = labels_json[str(np.argmax(label_predictions[index]))]
            font = ImageFont.truetype(
                "/usr/share/fonts/OTF/intelone-mono-font-family-regular.otf",
                size=20,
            )
            margin = 10
            left, top, right, bottom = font.getbbox(label)
            width, height = right - left, bottom - top
            button_size = (width + 2 * margin, height + 3 * margin)
            button_img = Image.new("RGBA", button_size, "black")
            button_draw = ImageDraw.Draw(button_img)
            button_draw.text((10, 10), label, fill=(0, 255, 0), font=font)
            frame.paste(button_img, (0, 0))
            frames[index] = np.array(frame, dtype=np.uint8)
        logger.info(
            f"\t$BLUEmodifying images took $RED{timer()-start:.6f} $BLUEseconds$RESET"
        )
        logger.info(
            f"\t$BLUEsaving $RED{len(resized_frames)}$BLUE  frames in $GREEN{input_frames_path} $RESET"
        )
        with gzip.GzipFile(input_frames_path, mode="w", compresslevel=3) as f:
            np.save(f, resized_frames)
        logger.info(f"\t$BLUEwriting output video $GREEN{output_video_path}$RESET")
        vidwrite(
            output_video_path,
            resized_frames,
            fps=fps // 4,
            in_pix_fmt=pix_fmt,
            input_args=ffmpeg_args,
            output_args={
                i: ffmpeg_args[i] for i in ffmpeg_args if i != "hide_banner"
            },
        )
        return

def define_model(self):
    # freeze training any of the layers of the base model
    for layer in self.base_model.layers:
        layer.trainable = False

    flatten = self.base_model.output
    flatten = Flatten()(flatten)

    bboxHead = Dense(128, activation="relu")(flatten)
    bboxHead = Dense(64, activation="relu")(bboxHead)
    bboxHead = Dense(32, activation="relu")(bboxHead)
    bboxHead = Dense(4, activation="sigmoid", name="bounding_box")(bboxHead)
    # 4 neurons correspond to 4 co-ords in output bbox

    softmaxHead = Dense(512, activation="relu")(flatten)
    if self.lite_model_required == False:
        softmaxHead = Dropout(0.5)(softmaxHead)
    softmaxHead = Dense(512, activation="relu")(softmaxHead)
    if self.lite_model_required == False:
        softmaxHead = Dropout(0.5)(softmaxHead)
    softmaxHead = Dense(512, activation="relu")(softmaxHead)
    if self.lite_model_required == False:
        softmaxHead = Dropout(0.5)(softmaxHead)
    softmaxHead = Dense(NUM_CLASSES, activation="softmax", name="class_label")(
        softmaxHead
    )
    self.model = Model(
```

A.4: Custom model module

```python
import os
import warnings
from logging import Formatter, Logger, LogRecord, captureWarnings, getLogger
from logging.handlers import RotatingFileHandler


class ColorFormatter(Formatter):
    """Logging formatter adding console colors to the output."""

    black, red, green, yellow, blue, magenta, cyan, white = range(8)
    colors = {
        "WARNING": yellow,
        "INFO": green,
        "DEBUG": blue,
        "CRITICAL": yellow,
        "ERROR": red,
        "RED": red,
        "GREEN": green,
        "YELLOW": yellow,
        "BLUE": blue,
        "MAGENTA": magenta,
        "CYAN": cyan,
        "WHITE": white,
    }
    reset_seq = "\033[0m"
    color_seq = "\033[%dm"
    bold_seq = "\033[1m"

    def format(self, record: LogRecord) -> str:
        """Format the record with colors."""
        color = self.color_seq % (30 + self.colors[record.levelname])
        message = Formatter.format(self, record)
        message = (
            message.replace("$RESET", self.reset_seq)
            .replace("$BOLD", self.bold_seq)
            .replace("$COLOR", color)
        )
        for color, value in self.colors.items():
            message = (
                message.replace("$" + color, self.color_seq % (value + 30))
                .replace("$BG" + color, self.color_seq % (value + 40))
                .replace("$BG-" + color, self.color_seq % (value + 40))
            )
        return message + self.reset_seq
```

```python
def init_log(
    logger: Logger,
    log_path: str,
    mode: str = "a",
    format_str: str = "%(message)s",
    log_level="INFO",
) -> None:
    for handler in logger.handlers:
        logger.removeHandler(handler)
    should_roll_over = os.path.exists(log_path)
```

A.5: Logger module

```python
import gzip
import json
import math
import os
import pathlib
import pickle
import random
from timeit import default_timer as timer

import cv2
import ffmpeg
import jsonpickle
import numpy as np
import pandas as pd
from keras.callbacks import History
from keras.layers import Dense, Dropout, Flatten, Input
from keras.models import Model, load_model

# from keras.optimizers.adam import Adam
from keras.optimizers.adamw import AdamW
from keras.utils import img_to_array, load_img
from keras.utils.vis_utils import plot_model
from PIL import Image, ImageDraw, ImageFont
from sklearn.model_selection import train_test_split
from tensorflow import lite

from modules.config import (
    BATCH_SIZE,
    IMG_SIZE,
    INIT_LR,
    NUM_CLASSES,
    NUM_EPOCHS,
    input_path,
    input_videos_dir,
    labels_path,
    logger,
    output_path,
    test_data_dir,
    training_data_dir,
)
from modules.load_data import load_test_data, load_training_data
from modules.videowriter import vidwrite


class LiteModel:
    def __init__(self, interpreter):
        self.interpreter: lite.Interpreter = interpreter
        self.interpreter.allocate_tensors()
        input_det = self.interpreter.get_input_details()[0]
        output_det = self.interpreter.get_output_details()[1]
        self.input_index = input_det["index"]
        self.output_index = output_det["index"]
        self.input_shape = input_det["shape"]
        self.output_shape = output_det["shape"]
        self.input_dtype = input_det["dtype"]
        self.output_dtype = output_det["dtype"]

    def predict(self, inp: np.ndarray):
        inp = inp.astype(self.input_dtype)
        count = inp.shape[0]
        out = np.zeros((count, self.output_shape[1]), dtype=self.output_dtype)
        for i in range(count):
            self.interpreter.set_tensor(self.input_index, inp[i : i + 1])
            self.interpreter.invoke()
            out[i] = self.interpreter.get_tensor(self.output_index)[0]
        return out


class CustomModel:
    def __init__(
        self, base_model_function: Model, trained: bool, lite_model_required: bool
    ) -> None:
        self.model: Model = None
        self.history: dict = None
        self.lite_model_required = lite_model_required
        self.tflite_model = None
        input_shape = IMG_SIZE + tuple([3])
        input_tensor = Input(shape=IMG_SIZE + tuple([3]))
        base_model_args = dict(
            input_shape=input_shape,
            weights="imagenet",
            include_top=False,
            input_tensor=input_tensor,
        )
        self.base_model: Model = base_model_function(**base_model_args)
        self.saved_model_path = os.path.join(
            output_path, self.base_model.name, "model.h5"
        )
        self.history_path = os.path.join(
            output_path, self.base_model.name, "training_history.json"
        )
        self.scores_path = os.path.join(output_path, self.base_model.name, "scores.txt")
```

```python
        self.lb_path = os.path.join(output_path, self.base_model.name, "lb.pickle")
        self.predicted_labels_path = os.path.join(
            output_path, self.base_model.name, "predicted_labels.pickle"
        )
        self.accuracies_path = os.path.join(
            output_path, self.base_model.name, "accuracies.txt"
        )
        if not trained:
            self.define_model()
            self.history = self.train().history
        else:
            self.model = load_model(self.saved_model_path)
            with open(self.history_path, "r") as f:
                self.history = jsonpickle.decode(f.read())
        if self.lite_model_required:
            lite_model_path = pathlib.Path(self.saved_model_path).with_suffix(".tflite")
            if os.path.exists(lite_model_path):
                with open(lite_model_path, "rb") as f:
                    self.tflite_model = f.read()
                    self.tflite_model_instance = LiteModel(
                        lite.Interpreter(model_path=str(lite_model_path))
                    )
            else:
                try:
                    self.tflite_model = lite.TFLiteConverter.from_keras_model(
                        self.model
                    ).convert()
                    self.tflite_model_instance = LiteModel(
                        lite.Interpreter(model_content=self.tflite_model)
                    )
                    with open(lite_model_path, "wb") as f:
                        f.write(self.tflite_model)
                except Exception:
                    logger.error(
                        f"$REDCould not convert model to `tf.lite` model$RESET",
                        exc_info=1,
                    )
                    exit(0)

    def train(self) -> History:
        # Load the data
        images, labels, bboxes, _ = load_training_data(training_data_dir)
        split = train_test_split(images, labels, bboxes, test_size=0.2, random_state=12)

        (x_train, x_validation) = split[0:2]
        (y_train, y_validation) = split[2:4]
        (bboxes_train, bboxes_validation) = split[4:6]

        train_targets = {"class_label": y_train, "bounding_box": bboxes_train}
        validation_targets = {
            "class_label": y_validation,
            "bounding_box": bboxes_validation,
        }

        # self.model.summary()

        if not os.path.exists(f"../../figuri/{self.base_model.name}/model_plot.png"):
            plot_model(
                self.model,
                to_file=f"../../figuri/{self.base_model.name}/model_plot.png",
                dpi=192,
                show_shapes=True,
                show_layer_names=True,
                show_layer_activations=True,
                show_trainable=True,
            )

        logger.info(f"\t$BLUEstarting training...$RESET")
        start = timer()
        # Train the model
        history = self.model.fit(
            x_train,
            train_targets,
            validation_data=(x_validation, validation_targets),
            epochs=NUM_EPOCHS,
            batch_size=BATCH_SIZE,
            verbose=0,
        )
        logger.info(f"\t$BLUEending training...$RESET")
        logger.info(f"\t$BLUEtraining took $RED{timer()-start:.6f} $BLUEseconds$RESET")
        self.model.save(self.saved_model_path)
        with open(self.history_path, "w") as f:
            f.write(jsonpickle.encode(history.history))
        return history

    def test_model_images(self, include_random: bool = False):
        start = timer()
        images, bboxes, image_paths = load_test_data(test_data_dir)
        if os.path.exists(self.predicted_labels_path):
            with open(self.predicted_labels_path, "rb") as f:
                predicted_labels = pickle.load(f)
        else:
```

```python
            logger.info(f"\t$BLUEpredicting labels for images...$RESET")
            predicted_labels = self.model.predict(
                images,
                batch_size=BATCH_SIZE,
                verbose=0,
            )[1]
            f"\t$BLUEpredicting labels took $RED{timer()-start:.6f} $BLUEseconds$RESET"
            with open(self.predicted_labels_path, "wb") as f:
                pickle.dump(predicted_labels, f)
        predicted_labels = np.array(predicted_labels)
        with open(os.path.join(test_data_dir, "Test.csv")) as f:
            correct_labels = pd.read_csv(f, sep=",")["ClassId"].to_numpy(dtype="uint32")
        with open(labels_path, "r") as f:
            labels_json = json.load(f)

        testTargets = {"class_label": predicted_labels, "bounding_box": bboxes}
        metrics_names: list[str] = self.model.metrics_names
        correct = 0

        if not os.path.exists(self.scores_path):
            logger.info(
                f"\t$BLUEevaluating model $GREEN{self.base_model.name} $BLUEand saving scores...$RESET"
            )
            scores = self.model.evaluate(
                images,
                testTargets,
                batch_size=BATCH_SIZE,
                verbose=0,
            )

            for image_path in image_paths:
                index = np.where(image_paths == image_path)[0][0]
                image = load_img(image_path, target_size=IMG_SIZE)
                image = img_to_array(image) / 255.0
                image = np.expand_dims(image, axis=0)

                # # finding class label with highest pred. probability
                i = np.argmax(predicted_labels[index], axis=0)
                predicted_label = labels_json[str(i)]
                correct_label = labels_json[str(correct_labels[index])]

                if predicted_label == correct_label:
                    correct += 1

            test_acc = f"Test accuracy: {correct/len(images)*100:.2f}%\n"
            with open(self.scores_path, "w") as f:
                for name, score in zip(metrics_names, scores):
                    name = name.split("_")
                    name[0] = name[0].capitalize()
                    joined_name = " ".join(name)
                    if "Loss" in joined_name or "loss" in joined_name:
                        line = "{}: {:.2f}\n".format(joined_name, score)
                    else:
                        line = "{}: {:.2f}%\n".format(joined_name, score * 100)
                    f.write(line)
                f.write(test_acc)
        if include_random:
            for i in range(100):
                correct = 0
                random.seed(random.random() * 50)
                random_choices = random.choices(
                    image_paths, k=int(len(image_paths) / 100)
                )
                for image_path in random_choices:
                    index = np.where(image_paths == image_path)[0][0]
                    i = np.argmax(predicted_labels[index], axis=0)
                    predicted_label = labels_json[str(i)]
                    correct_label = labels_json[str(correct_labels[index])]
                    if predicted_label == correct_label:
                        correct += 1
                random_acc = f"{correct/len(random_choices)*100:.2f}\n"
                with open(self.accuracies_path, "a") as f:
                    f.write(random_acc)
        f"\t$BLUEtesting images took $RED{timer()-start:.6f} $BLUEseconds$RESET"

    def test_model_videos(self, input_video_fn: str):
        logger.info(
            f"$BOLD$COLOR==> $BOLD$BLUEprocessing input video $GREEN{input_video_fn}$RESET"
        )
        input_video_path = os.path.join(input_videos_dir, input_video_fn)
        output_video_path = os.path.join(
            output_path,
            self.base_model.name,
            f'output-{input_video_fn.replace(".mp4","")}.mp4',
        )
        input_frames_path = os.path.join(
            input_path,
            "frames",
            self.base_model.name,
            f'{input_video_fn.replace(".mp4","")}.frames.npy.gz',
        )
        with open(labels_path, "r") as f:
            labels_json = json.load(f)
        video_stream = ffmpeg.probe(input_video_path)["streams"][0]
        ns = {"__builtins__": None}
        # frame_height = int(video_stream["height"])
        # frame_width = int(video_stream["width"])
        fps = math.ceil(float(eval(video_stream["avg_frame_rate"], ns)))
        # pix_fmt = video_stream["pix_fmt"]
        pix_fmt = "rgb24"
        ffmpeg_args = {
            "hide_banner": None,
            "loglevel": "quiet",
            "v": "quiet",
            "nostats": None,
        }
        if not os.path.exists(pathlib.Path(input_frames_path).parent):
            os.mkdir(pathlib.Path(input_frames_path).parent)
        if not os.path.exists(pathlib.Path(output_video_path).parent):
            os.mkdir(pathlib.Path(output_video_path).parent)
        generated_frames = os.path.exists(input_frames_path)
        generated_video = os.path.exists(output_video_path)
        if generated_frames:
            if generated_video:
                logger.info(
                    f"\t$BLUEalready generated frames and video for video $GREEN{input_video_fn}$RESET"
                )
            else:
                logger.info(
                    f"\t$BLUEreading generated frames for video $GREEN{input_video_fn}$RESET"
                )
                with gzip.GzipFile(input_frames_path, "r") as f:
                    resized_frames = np.load(f)
                logger.info(f"writing output video $GREEN{output_video_path}$RESET")
                vidwrite(
                    output_video_path,
                    resized_frames,
                    fps=fps // 4,
                    in_pix_fmt=pix_fmt,
                    input_args=ffmpeg_args,
                    output_args={
                        i: ffmpeg_args[i] for i in ffmpeg_args if i != "hide_banner"
                    },
                )
                return
        else:
            logger.info(f"\t$BLUEgenerating frames")
            vidcap = cv2.VideoCapture(input_video_path)
            resized_frames = []
            frames = []
            count = 0
            while vidcap.isOpened():
                success, frame = vidcap.read()
                if success:
                    img = Image.fromarray(frame)
                    frames.append(img)
                    resized_frame = cv2.resize(frame, (64, 64))
                    resized_frames.append(resized_frame)
                    count += fps
                    vidcap.set(cv2.CAP_PROP_POS_FRAMES, count)
                else:
                    vidcap.release()
                    break
            resized_frames = np.array(resized_frames)
            start = timer()
            if not self.lite_model_required:
                label_predictions = self.model.predict(
                    resized_frames, verbose=0, batch_size=BATCH_SIZE
                )
            else:
                label_predictions = self.tflite_model_instance.predict(resized_frames)
            logger.info(
                f"\t$BLUEpredicting labels took $RED{timer()-start:.6f} $BLUEseconds$RESET"
            )
            start = timer()
            for index, frame in zip(range(resized_frames.shape[0]), frames):
                label = labels_json[str(np.argmax(label_predictions[index]))]
                font = ImageFont.truetype(
                    "/usr/share/fonts/OTF/intelone-mono-font-family-regular.otf",
                    size=20,
                )
                margin = 10
                left, top, right, bottom = font.getbbox(label)
                width, height = right - left, bottom - top
                button_size = (width + 2 * margin, height + 3 * margin)
                button_img = Image.new("RGBA", button_size, "black")
                button_draw = ImageDraw.Draw(button_img)
                button_draw.text((10, 10), label, fill=(0, 255, 0), font=font)
                frame.paste(button_img, (0, 0))
                frames[index] = np.array(frame, dtype=np.uint8)
            logger.info(
                f"\t$BLUEmodifying images took $RED{timer()-start:.6f} $BLUEseconds$RESET"
            )
            logger.info(
                f"\t$BLUEsaving $RED{len(resized_frames)}$BLUE  frames in $GREEN{input_frames_path} $RESET"
            )
            with gzip.GzipFile(input_frames_path, mode="w", compresslevel=3) as f:
                np.save(f, resized_frames)
            logger.info(f"\t$BLUEwriting output video $GREEN{output_video_path}$RESET")
            vidwrite(
                output_video_path,
                resized_frames,
                fps=fps // 4,
                in_pix_fmt=pix_fmt,
                input_args=ffmpeg_args,
                output_args={
                    i: ffmpeg_args[i] for i in ffmpeg_args if i != "hide_banner"
                },
            )
            return

    def define_model(self):
        # freeze training any of the layers of the base model
        for layer in self.base_model.layers:
            layer.trainable = False

        flatten = self.base_model.output
        flatten = Flatten()(flatten)

        bboxHead = Dense(128, activation="relu")(flatten)
        bboxHead = Dense(64, activation="relu")(bboxHead)
        bboxHead = Dense(32, activation="relu")(bboxHead)
        bboxHead = Dense(4, activation="sigmoid", name="bounding_box")(bboxHead)
        # 4 neurons correspond to 4 co-ords in output bbox

        softmaxHead = Dense(512, activation="relu")(flatten)
        if self.lite_model_required == False:
            softmaxHead = Dropout(0.5)(softmaxHead)
        softmaxHead = Dense(512, activation="relu")(softmaxHead)
        if self.lite_model_required == False:
            softmaxHead = Dropout(0.5)(softmaxHead)
        softmaxHead = Dense(512, activation="relu")(softmaxHead)
        if self.lite_model_required == False:
            softmaxHead = Dropout(0.5)(softmaxHead)
        softmaxHead = Dense(NUM_CLASSES, activation="softmax", name="class_label")(
            softmaxHead
        )
        self.model = Model(
```

A.6: Custom model module