Antoniu Karina-Maria, Bădulescu Andrei-Marcel,
Mărășescu Ștefan Gabriel

Software Architecture Documentation

Traffic Sign Recognition Project

Introduction

The Traffic Sign Recognition Project aims to demonstrate the functionality of a Machine Learning model that detects and classifies traffic signs that is tightly integrated with a Windows UI Desktop application. This project's main scope is to illustrate the capabilities of a modern Computer Vision algorithm paired with a reliable camera system in the field of ADAS / Active Traffic Safety systems, as early detection of traffic signs can drastically improve the decision flow of Assisted Driving systems.

The application is structured to put emphasis on the input / output data and provide the user basic functionalities to interact with the trained ML model. These functions include uploading footage for testing or further training, sending given input for processing through the pipeline, viewing the results of the aforementioned process and viewing a list of all traffic signs that the model should recognise. There is a slim sidebar for selecting between interaction with the model or viewing all of the recognised traffic signs, which will be listed on multiple rows and columns.

Given the structure of this project, there are three main parts: the frontend application, written in C# and XAML using .NET and WinUI3; the processing pipeline, written in Python and the Machine Learning training algorithm, written also in Python.

Some examples of intended users for this application are Automotive testers, who would test the video output quality and reliability of their ADAS systems, and Insurance Providers, who would want to see traffic signs in the area of an accident in order to establish which driver is at fault.

System Overview

Given the nature of the project, we decided to use Python for the ML training algorithm and pipeline development, because there are numerous community-provided libraries that can help speed up the process of building a reliable and efficient back-end. Some of the libraries we use are pandas (reading and processing CSV data), moviepy (editing and cutting video files), torch, numpy, keras.

The network's architecture is divided into a "backbone" and a "head". The backbone consists of six sequential blocks primarily composed of Convolutional Layers and MaxPool Layers. The convolutional layers function as sophisticated feature extractors; each layer's kernel, or filter, is a small, learnable matrix trained to recognize a specific low-level pattern, such as a distinct edge, corner, or texture. As the kernel scans the input, it produces a feature map that indicates the locations and strength of these detected patterns. Following these layers, MaxPooling is applied to progressively downsample the spatial dimensions of the representation and to select only the maximum feature activation from a local vicinity. After this deep feature extraction, the final block (head) is responsible for interpreting these features and generating the final prediction. This transition begins as the 3D feature map is

flattened into a 1D vector. This vector is then processed by fully-connected layers, which learn complex combinations of the extracted features. To enable the learning of non-linear relationships, a LeakyReLU activation is applied. A Dropout layer is also included as a regularization technique, active only during training, to mitigate overfitting by randomly deactivating neurons. Finally, a Sigmoid function is applied to the output tensor, scaling every value to a standardized range between 0 and 1.

There are two available pipelines available to the user: the training and the inference pipeline. The first aims to help improve reliability and/or expand the array of recognised traffic signs by providing a way to further train the ML model. Although the application is shipped with a default trained model, the user may opt to provide additional data (given that the data is processed beforehand with the toolset that we provide) to increase the functional level. The second is used to provide input and just obtain results from the model, i.e. one may want to see each traffic sign that is successfully recognised by the model in a video or a picture. This pipeline is known as the testing or inference pipeline.

The front-end application is written in C# and XAML and it is in essence a next-generation UWP app, as it uses WinUI3 (a UI framework) for displaying every element, ranging from buttons to sidebars and application segments. We chose to implement the application using these technologies because we wanted to learn to develop applications for Windows natively and it also serves as an advantage, given that most of the expected users will use a Windows machine to run our application. The implementation follows the MVVM (ModelView-View Model) guidelines, as this is the widely accepted design pattern for this technology stack and all documentation and guides regarding WinUI3 is designed with MVVM in mind.

Detailed Component Design

The project is divided into three distinct parts: the pipeline, the Machine Learning algorithm and the front-end application. This Software Architecture Documents explains in depth each component in the following sections.

Pipelines

The Pipeline component is responsible for handling all the data processing required by the Traffic Light Recognition application. It serves as the bridge between the raw video input, the Machine Learning model, and the final video output that the user sees. The pipeline is divided into two separate workflows:

1. Training Pipeline – prepares and processes training data for the ML mod

The training pipeline begins with video segmentation, downscaling, and frame extraction, all implemented in a single script (video_cutter). This script reads a CSV file that indicates timestamps of subclips where relevant traffic signs appear, cuts the long dashcam video into short clips, reduces resolution (1080p → 720p) and frame rate (60fps → 10fps) to simplify the input for the ML model and reduce computational load during training, and then extracts each clip into individual frames saved as images in a dedicated folder.

Next, a separate script handles dataset preparation and annotation integration. This script adds bounding-box annotations to the extracted frames and organizes them in the format required by the ML training script, producing a fully labeled dataset ready for model training.

All pipeline outputs are stored in well-organized folders containing images and corresponding labels. These folders serve as the input for the ML model training script. Communication with the rest of the system is entirely file-based, with no live network interaction required.

2. Inference Pipeline

The User Video Processing Pipeline handles videos uploaded by users, detects traffic signs using the trained ML model, and produces annotated videos for frontend display. The pipeline begins with video preprocessing, implemented in a python script that downsizes the video to 720p and reduces the frame rate to match the ML model's expected input, then extracts all the individual frames from the video and saves them in a dedicated folder for inference.

Next, the ML model inference step is handled by a separate script, which loads the trained ML model and runs inference on each extracted frame to predict bounding boxes and class labels for detected traffic signs. The output of this step is structured prediction data that will be used to visually annotate the frames. Using the predictions from the ML model, this script draws rectangles and labels on each frame with libraries such as OpenCV or MoviePy, producing visually annotated frames.

Finally, a video reconstruction script combines the annotated frames back into a video. The video can optionally be upscaled to 1080p for display purposes, and is saved as a final .mp4 file ready for frontend consumption. The output video is stored in a predefined directory where the frontend can access it, with all communication handled via the file system.

Machine Learning algorithm

To detect and classify traffic signs from images we will use a custom, from scratch implementation of the YOLO algorithm. The algorithm is organized into five important modules, each performing a distinct function.

The primary module is the Model Architecture itself which defines a complex structure made from 7 different sequential blocks of layers. The class constructor sets the most important parameters for the YOLO algorithm: a 7x7 grid with 2 bounding boxes per cell to classify 45 specific traffic sign classes resulting in the model's final prediction shape of (7, 7, 55). The forward method defines the data's propagation path, sequentially passing an input tensor (at first having the 448x448x3 dimension) through the network, transforming it from raw pixel data into abstract feature space and finally reshaping the output into the calculated expected grid.

The second module is the Data Pipeline whose primary function is to translate each raw input image into the tensor format the neural network expects. At first, the image is

loaded and converted into a PyTorch tensor and separated into a 448 x 448 grid. The corresponding label file is read, which may contain annotations for one or more traffic signs. Each label is translated into a vector. It first instantiates an empty, zero-filled target tensor with the shape of the model's output. Then, for each individual traffic sign in the label file, it identifies the responsible 7x7 grid cell based on the sign's center coordinates. It then computes the coordinates relative to that cell's top-left corner and populates the target tensor at that specific location with the complete vector: the relative coordinates, the object's dimensions, a confidence value of 1, and the correct class vector. The module returns two distinct tensors.

Third, the Loss Function Module is the system's evaluator, measuring the discrepancy between the model's predictions and the ground truth. It is a composite loss function, highly important to the Yolo algorithm. A necessary preceding step involves the Intersection over Union (IoU) utility, which is used to identify which of the 2 bounding boxes is "responsible" for the ground truth object. The total error is computed as a weighted sum of three distinct components: a Localization Loss (for box coordinate error, heavily weighted), a Confidence Loss (a penalty punishing both false positives and missed detections) and a Classification Loss (penalizing the misidentification of any of the 45 traffic sign classes).

The Training Script is the fourth module, the main part of the program, connecting all of the preceding modules. It contains the primary training loop: for each batch of data executes the 3 core steps of deep learning: a forward pass to get model predictions, a loss calculation to measure the error and a backward pass to compute and apply the gradients, thereby updating the model's weights.

The final part of the algorithm contains the Non-Maximum Suppression function to process the finishing output. This code deciphers the grid-relative output, filters all low-confidence detections, and then algorithmically refines the remaining candidates. It iteratively selects the box with the highest confidence, retains it as a final decision, and then discards all other boxes that have a significant overlap (based on the IoU threshold). This process is repeated until only a clean unambiguous list of final traffic signs detections remains.

Front-end application

The front-end will have a main class that will describe the application itself, and every subsequent view (meaning the list of supported traffic signs and the ML model interaction screen) will also be described as a standalone class. The application has functions implemented that support communication with the two pipelines, more exactly functions that will be responsible for the following operations:

- sending the data provided by the user to one of the two pipelines
- gathering the results provided by the model through the pipelines
- launching the toolchain for processing input data (videos, photos)

The application will interact with the scripts by saving the data provided by the user in a predefined working directory and executing the scripts while providing as parameter the same path. The list of supported traffic signs will be distributed with the app as a JSON object accompanied by images for each sign.

The compatible traffic signs will be displayed inside a scrollable list, with multiple elements of fixed size on multiple rows and columns that should adjust naturally to the dimensions of the window.

The main screen, where the user can interact with the ML model, will in the first phase present a central pair of elements, the first being a button through which content can be uploaded and the second being a dropdown list where the user can select between training and testing / inference. If the user chooses to test the model, the results will be displayed in the center of the window as a preview, with the option to scroll through photos or seek the video. However, if the user chooses to further train the model, the application will ask for confirmation if the data input is already processed using the pre-processing toolchain (that is included in the release alongside the application). After that, the application sends the data to the model for refinement and displays a notification when the process ends.

Requirements and deployment

The application can be run on machines that have Windows 11 or 10 version 1809 installed. The model and the pipelines require Python version 3.14.0, as well as additional libraries that are mentioned in the requirements.txt file. There is no need for any .NET framework installation, as there is no dependency on it.

In the project's repository, there is a release that after following the deployment steps, offers the application as a click-to-run compatible executable, meaning there is no processing or preparation that needs to be done by either the front-end or the back-end before execution starts.

Conclusion

This project aims to demonstrate what students can build using knowledge acquired in college, as well as from online sources / courses and modern tools and frameworks. The chosen theme reflects a product that would easily fit in the market of the current day, given the advance of ADAS systems and need for autonomous traffic safety systems that use state-of-the-art technologies and features.

We expect to work approximately 30 hours per person towards implementing all of the features, as well as all the pipelines and tools that the projects proposes to include. In order to complete the feature set and develop a reliable system, every team member had to learn intermediate Git functions, how to search and use various Python libraries, how to develop a C# front-end that interfaces with Python scripts and toolchains and general good practice rules regarding product life cycle and development.

Given that at this stage the development percentage is around 15%, there may appear slight differences between expected usage of libraries / frameworks, as well as interface functionalities and layout and actual features implemented and available to the user.

The data used to train the model is partially obtained from online sources that will be mentioned in the final documentation of the project, as well as from in-house footage recorded in traffic in Bucharest, Romania.