Mini Project Report

Specialty: Artificiel Intelligence

Traffic sign recognition

Directed by :

- Bekkari Abderrahmane.
- Assassi Salah Eddine.
- Nouar El Mouataz Billah.
- Azri Mohamed.
- Meftah Abderrahim.

University year: 2024-2025

1. Data collection

1.1 Introduction

1.1.1 General Summary

Data collection is the process of systematically gathering information for analysis and decision-making. It involves identifying the purpose of the data, selecting appropriate methods, and defining the scope of collection. Methods can be primary (e.g., surveys, interviews) or secondary (e.g., existing records, databases). Data quality is ensured through validation and by minimizing errors during collection. Ethical considerations, such as consent and privacy, must be adhered to. Properly collected data supports accurate analysis and meaningful conclusions, providing a foundation for informed decisions.

1.2 Data collection sources

1.2.1 Mapillary

Mapillary

- Mapillary is a platform for crowdsourced street-level imagery, where users can upload photos and create a visual map of streets and locations worldwide. It provides a rich dataset that can be used for a variety of applications, including geographic mapping, urban planning, autonomous vehicle training, and more.
- Mapillary collects images from both users and partners, allowing contributors to add photos from any location. The platform also offers tools for analyzing and using these images in various ways, such as detecting objects or identifying street signs, and it's often used by developers and researchers for computer vision and machine learning projects.

- **Purpus of use it**

Here's a more concise version in item form regarding the purpose of using Mapillary in a traffic sign dataset:

1. **Crowdsourced and Diverse Data**
   - Images from various regions, offering a wide range of traffic signs.
   - Real-world driving conditions for more authentic data.
2. **Large-scale Dataset Generation**
   - Massive volume of street-level images.
   - Easier creation of large traffic sign datasets for model training.

3.Improved Traffic Sign Recognition

- Helps train machine learning models for traffic sign detection and classification.
- Allows testing of models in realistic conditions.

3. **Geospatial Context**
   - Geo-tagged images for precise location data.
   - Surrounding environment context aids model understanding.
4. **Public and Open Data**
   - Free access to images and data, ideal for research or small-scale projects.
5. **Data Augmentation**
   - Enhances existing traffic sign datasets by adding more diverse examples.

• Example Use Cases

- **Autonomous Vehicle Training**: Improve models to recognize stop signs, speed limits, etc.
- **Geospatial Analysis**: Study traffic sign distribution across regions.
- **Traffic Safety Research**: Analyze the effectiveness and distribution of traffic signs.

• Size of data

The size of the data from Mapillary can vary depending on the specific region, the number of images, and the type of content being extracted. However, here are some general points about the data size:

1. Image Size

- **Resolution**: Mapillary images can vary in resolution, but typical street-level imagery is around 1024x1024 pixels or higher.
- **File Size**: Individual images typically range from **500 KB to 2 MB** depending on resolution and compression settings.

2. Dataset Size

- **Volume of Images**: Mapillary has millions of images contributed by users around the world. As of recent reports, Mapillary hosts over **500 million street-level images**.
- **Traffic Sign Dataset**: The size of a traffic sign-specific dataset will depend on how many images are labeled with traffic signs. A complete dataset with millions of labeled traffic sign images can range from **tens of gigabytes (GB) to hundreds of gigabytes** or more.

3. Geo-tagged Data

- **Metadata**: In addition to images, Mapillary provides metadata, such as geolocation information and image timestamps. This metadata adds **a few kilobytes to a megabyte** of data per image, depending on the level of detail.

4. Data Access

- **Mapillary API**: If you're using the Mapillary API to collect data, you can download images in batches, and the total data size will depend on the number of images and the resolution you request. API responses for a batch of images typically range from **hundreds of megabytes to several gigabytes** per download.

Example:

- If you were to download 10,000 images at an average size of 1 MB each, the total dataset size would be around **10 GB**.
- A large-scale dataset containing images from multiple regions could easily reach **hundreds of GB or more**.

The size of the data will depend on how much of Mapillary's dataset you choose to use and the type of images (e.g., full-resolution or smaller versions) you request.

1.2.2 Roboflow

Roboflow is a platform designed to help with the creation, training, and deployment of machine learning models, particularly for computer vision tasks such as image classification, object detection, and segmentation. It offers a variety of tools and features to simplify the entire workflow, from data annotation to model deployment.

- **Key Features of Roboflow**

1. **Data Annotation and Labeling**:
   - **Automated Labeling**: Roboflow can help automate the annotation process by providing tools to quickly label images with bounding boxes, segmentation masks, or key points.
   - **Collaboration**: It allows teams to collaborate in annotating datasets and reviewing annotations to ensure accuracy.
2. **Dataset Management**:
   - **Dataset Import and Export**: You can import datasets from various sources (e.g., Mapillary, COCO, VOC) or upload your own images. Roboflow supports common formats like JSON, XML, and CSV.
   - **Preprocessing**: It provides easy tools to resize, crop, flip, and augment your images to improve the diversity of your training dataset and enhance model generalization.

3.Model Training:

- **Model Selection**: Roboflow allows you to choose pre-built model architectures for object detection, classification, and segmentation tasks (e.g., YOLO, EfficientDet, and others).
- **Training on the Cloud**: It supports cloud-based training, meaning you don't need powerful hardware locally to train models. Roboflow manages the training process and helps you monitor the model's performance.

3. **Data Augmentation**:
   - Roboflow offers a set of data augmentation techniques like rotations, flips, brightness adjustments, and more to expand your dataset and improve model robustness.
4. **Model Deployment**:
   - **API Access**: After training a model, Roboflow can help deploy the model via API for real-time predictions in your applications.
   - **Edge Deployment**: It also supports deployment to edge devices (e.g., Raspberry Pi, Nvidia Jetson) for real-time applications like robotics and IoT devices.
5. **Performance Monitoring**:
   - **Evaluation Metrics**: Roboflow provides tools to evaluate model performance using metrics like precision, recall, and mAP (mean average precision).
   - **Model Versioning**: You can track different versions of your models and monitor how changes to data or model parameters impact performance.
6. **Integration with Other Tools**:
   - **TensorFlow, PyTorch, Keras, and ONNX**: Roboflow supports integration with popular deep learning frameworks for seamless model training.
   - **Exporting Models**: After training, you can export models to formats compatible with TensorFlow, PyTorch, or ONNX for further use in production or other frameworks.

1.3 Conclution

Both **Mapillary** and **Roboflow** are powerful tools that can complement each other in building, training, and deploying traffic sign detection models.

- **Mapillary** provides a rich, crowdsourced dataset of street-level images, offering diversity in traffic signs, real-world conditions, and geospatial context. It enables the creation of large-scale, geographically diverse datasets for training models, which is essential for developing robust traffic sign recognition systems.
- **Roboflow**, on the other hand, offers a comprehensive platform for managing, annotating, and augmenting datasets. It streamlines the process of training and deploying computer vision models for tasks like traffic sign detection. With tools for data annotation, augmentation, and cloud-based model training, Roboflow ensures that you can efficiently build, refine, and deploy high-quality models.

Together, **Mapillary's** vast dataset of real-world images and **Roboflow's** advanced tools for model creation and deployment form a strong combination for developing traffic sign detection systems. Mapillary provides the diverse and scalable data needed, while Roboflow simplifies the pipeline from annotation to model deployment, enabling efficient development of robust, real-world machine learning solutions.

2. Realization

2.1 Introduction

2.1.1 General Summary

Python is a high-level, interpreted programming language known for its simplicity and readability, making it ideal for beginners and professionals alike. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python has an extensive standard library and a large ecosystem of third-party modules for tasks like web development, data analysis, machine learning, and automation. Its dynamic typing and ease of use make it flexible, while its scalability supports complex applications. Python emphasizes code readability with its indentation-based syntax and is widely used in academia, industry, and research for its versatility and efficiency.

2.1.2 Purpose of using

1. Ease of Use: Python's simple and readable syntax allows developers to focus on solving ML problems rather than dealing with complex programming constructs.

2. Extensive Libraries: Libraries like NumPy, Pandas, Scikit-learn, TensorFlow, and PyTorch provide robust tools for data processing, model building, and evaluation.

3. Versatility: Python supports multiple domains such as data analysis, visualization, and model deployment, making it an end-to-end solution for machine learning projects.

4. Strong Community Support: A large community of developers and researchers ensures that resources, tutorials, and solutions are widely available.

5. Integration Capabilities: Python easily integrates with other languages and tools, such as C++ and Hadoop, enabling seamless workflows.

6. Platform Independence: Python's compatibility across different operating systems allows developers to build and deploy ML solutions anywhere.

7. Visualization Tools: Libraries like Matplotlib and Seaborn aid in data visualization, crucial for interpreting results and understanding models.

2.2 Libraries used for programming

2.2.1 OS

The os module in Python provides a way to interact with the operating system. It offers functions to work with files, directories, and processes, making it a versatile tool for system-level programming. With os, you can manipulate paths, handle environment variables, manage files and directories, and interact with the system in a platform-independent manner. It is part of Python's standard library and is available across all platforms where Python is installed, helping to ensure portability between different operating systems.

- **Purpose of using**

1. Managing File Paths: It allows you to navigate file directories, handle relative and absolute file paths, and ensure your code works across different operating systems (e.g., Windows, Linux).

2. Loading and Saving Models: You can use it to load and save machine learning models or datasets stored on disk. Functions like `os.path.join()` ensure that paths are handled correctly regardless of the operating system.

3. Directory Management: It is useful for creating directories, checking if they exist, or removing files or directories during the data preprocessing and model training phases.

4. Environment Variables: You can set and retrieve environment variables, such as API keys or configurations, that might be needed for your machine learning code to interact with external services.

5. Process Management: For running or managing subprocesses (e.g., parallel training of models or running external scripts) in a machine learning pipeline.

2.2.2 Json

The `json` module in Python is used for parsing JSON (JavaScript Object Notation), a lightweight data interchange format. It allows you to convert Python objects such as dictionaries, lists, and strings into JSON format and vice versa. This is useful for handling data exchanged between a server and a client, as well as reading and writing data in JSON format from and to files or APIs.

- ### Purpose of using

1. Data Serialization and Deserialization: Convert machine learning models, hyperparameters, or results into JSON format for easy storage and retrieval.

2. Interfacing with APIs: Fetch data or send results to/from APIs in JSON format, which is commonly used in web services.

3. Config Files: Store and manage configuration settings (like model parameters or file paths) in JSON files, allowing easy adjustments without changing code.

4. Data Preprocessing: Read and manipulate datasets stored in JSON format, especially when dealing with nested or hierarchical data structures.

5. Logging and Reporting: Save training logs, metrics, and model performance data in JSON format for structured analysis and sharing.

2.2.3 Torch

The torch library is a fundamental component of the PyTorch framework, widely used for deep learning tasks. It provides support for tensor computation and building neural networks. Key modules such as torch.nn, imported as nn, offer classes and functions for defining and training neural network layers. The torch.optim module, imported as optim, provides optimization algorithms like SGD and Adam to update the model parameters. The torch.utils.data module, with classes like DataLoader and Dataset, facilitates efficient data handling by loading datasets in batches and applying transformations. Additionally, torchvision, which includes datasets and transforms, simplifies the process of working with standard image datasets and applying data preprocessing techniques. These tools together enable the creation, training, and evaluation of deep learning models in a flexible and efficient manner.

- **Purpose of using**

1. Tensor Computation: PyTorch allows efficient computation with tensors (multi-dimensional arrays), essential for building neural networks and performing operations like matrix multiplication, element-wise operations, etc.

2. Neural Network Construction: With `torch.nn`, you can easily define, customize, and train neural networks using pre-built layers, activation functions, and loss functions.

3. Optimization: `torch.optim` provides optimization algorithms (e.g., Adam, SGD) that help adjust the parameters of the neural network during training to minimize the loss function.

4. Data Handling: The `torch.utils.data` module facilitates efficient data loading and batching through `Dataset` and `DataLoader`, which is crucial when working with large datasets.

5. Computer Vision: `torchvision` offers tools for image-related tasks, providing pre-built datasets and transformations for image preprocessing and augmentation.

2.2.4 matplotlib

The matplotlib.pyplot module in Python, often imported as import matplotlib.pyplot as plt, is part of the Matplotlib library used for creating static, interactive, and animated visualizations. It provides a MATLAB-like interface, making it easy to generate various types of plots and charts, such as line plots, bar charts, histograms, and scatter plots. With functions like plt.plot(), plt.scatter(), and plt.show(), it allows for effective data visualization. Additionally, the module offers customization options for plot appearance, including titles, labels, and colors, and supports integration with libraries like NumPy for plotting data directly from arrays. Moreover, matplotlib.pyplot can save visualizations in several formats, such as PNG, PDF, and SVG. It is a powerful tool for presenting data insights in both static and interactive environments.

- **Purpose of using**

1. Data Visualization: It enables the creation of various types of plots (e.g., line plots, bar charts, histograms, scatter plots) to visually represent data.

2. Exploratory Data Analysis: By visualizing data distributions, relationships, and trends, it aids in the exploration and analysis of datasets.

3. Communication of Results: Visualizations created with `matplotlib.pyplot` help in presenting complex data in an understandable way, making it easier to communicate findings and insights.

2.2.5 YOLO

The YOLO (You Only Look Once) framework is a state-of-the-art system for real-time object detection. It is widely used in applications requiring high-speed and accurate object detection. YOLO uses a single neural network to predict bounding boxes and class probabilities directly from an image in one evaluation, making it extremely efficient. Its architecture emphasizes speed without compromising detection accuracy, making it suitable for various real-world scenarios. Key components include anchor boxes, grid cells, and confidence scores, which work together to predict and localize objects in images.

YOLOv8, the latest iteration of YOLO, introduces multiple model variants to balance speed, computational cost, and accuracy, denoted as **n (nano)**, **s (small)**, **m (medium)**, **l (large)**, and **x (extra-large)**. These variants differ in the complexity of the model architecture, which directly impacts their performance and hardware requirements.

- **YOLOv8n**: The smallest and fastest model, optimized for deployment on devices with limited computational resources such as mobile phones and edge devices.

- **YOLOv8s**: A slightly larger model, suitable for applications requiring better accuracy without significant performance trade-offs.

- **YOLOv8m**: The medium-sized variant, striking a balance between accuracy and inference speed, making it an excellent choice for general-purpose applications.

- **YOLOv8l**: A larger model with enhanced accuracy, ideal for scenarios where precision is prioritized over speed.

- **YOLOv8x**: The most complex and powerful model, delivering the highest accuracy at the cost of slower inference times, suitable for high-end hardware.

Purpose of using YOLOv8

1. **Real-Time Object Detection**: YOLO achieves impressive speeds by processing images in a single forward pass, enabling real-time detection.
2. **Model Scalability**: The YOLOv8 model variants allow developers to choose the best fit for their hardware and application needs, from lightweight to high-precision detection.
3. **End-to-End Training**: The framework allows simultaneous optimization of classification and localization, streamlining the model training process.
4. **Versatility**: YOLO is highly adaptable and supports detection across multiple classes, making it suitable for diverse use cases such as traffic monitoring, surveillance, and autonomous vehicles.
5. **Integration with PyTorch**: YOLO implementations in PyTorch (e.g., YOLOv8) provide flexibility for customization, training, and deployment, supported by advanced libraries like torch and torch vision.

This modular approach enables developers to select the optimal model size and complexity based on their specific use case, ensuring the right balance between speed and accuracy.

2.3 Conclusion

The imported libraries provide essential tools for machine learning workflows. **os** and **json** assist with file management and data handling, while **torch** and **torch.nn** form the core for building and training neural networks. **torch.optim** optimizes models, and **DataLoader** helps efficiently load data. **torchvision** supports computer vision tasks, **tqdm** tracks progress, and **matplotlib.pyplot** visualizes data and model performance. Together, they enable data processing, model creation, and visualization.

3. Implementation

3.1 Introduction

3.1.1 General Summary

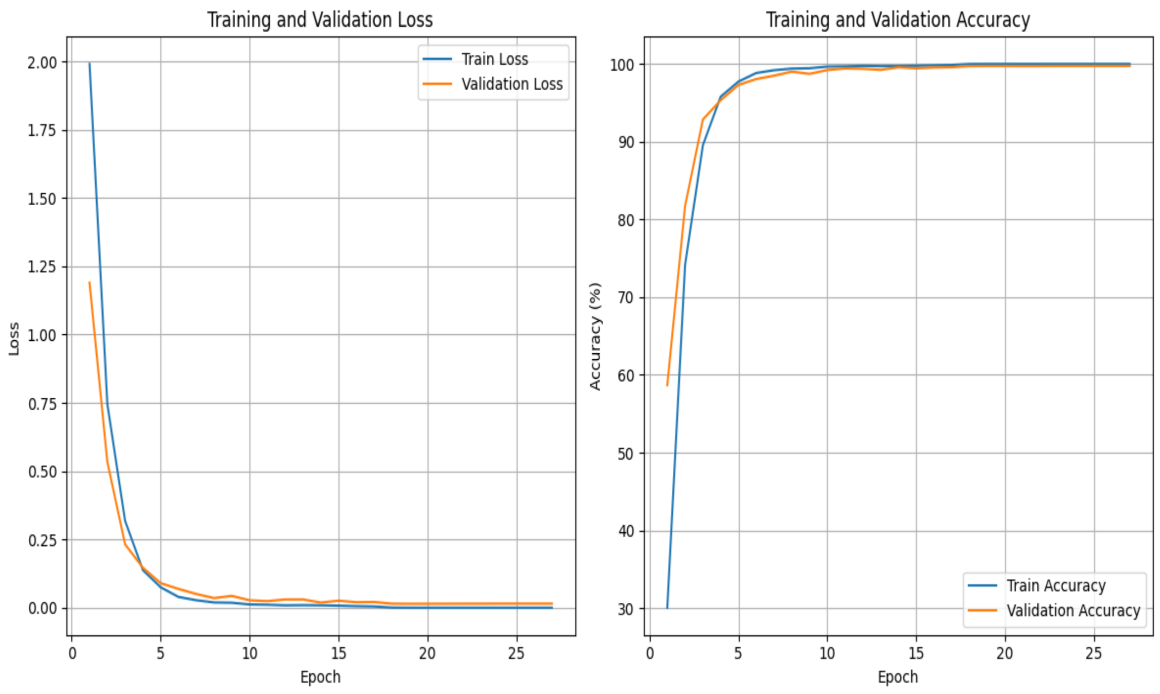
In machine learning, the **train** and **test** sets are fundamental to evaluating a model's performance. The **training set** is used to teach the model, allowing it to learn patterns, relationships, and features from the data by adjusting its parameters to minimize errors. After the model is trained, it is evaluated on the **test set**, which contains data the model has not seen before. This step helps assess how well the model can generalize to new, unseen data, providing an unbiased evaluation of its performance. The goal is to avoid overfitting, where the model memorizes the training data but struggles with new data, and underfitting, where the model fails to learn meaningful patterns from the training set. By using separate training and test sets, the model's ability to generalize is effectively tested.

3.2 The resulting model

3.2.1 Train

We used 28 epochs for training with a dataset of 240,000 images, divided into 5% test data and 95% training data, the training images were augmented using the following transformations:

- Horizontal flipping (left-right),
- Curving (left-right),
- Scaling (+5%),
- Rotation (5-20 degrees),
- Addition of noise (5%),
- RGB-to-GBR color shift.



As shown in Schema 1, the training and validation loss decreased steadily, while accuracy increased, stabilizing around 99% after 5 epochs. This indicates effective learning without signs of significant overfitting. For a detailed breakdown of results at each epoch, refer to this like : https://drive.google.com/file/d/1N81G77cGTXKoV6ciceiG2FeU-FHuDzDr/view?usp=drive_link

3.2.2 Test

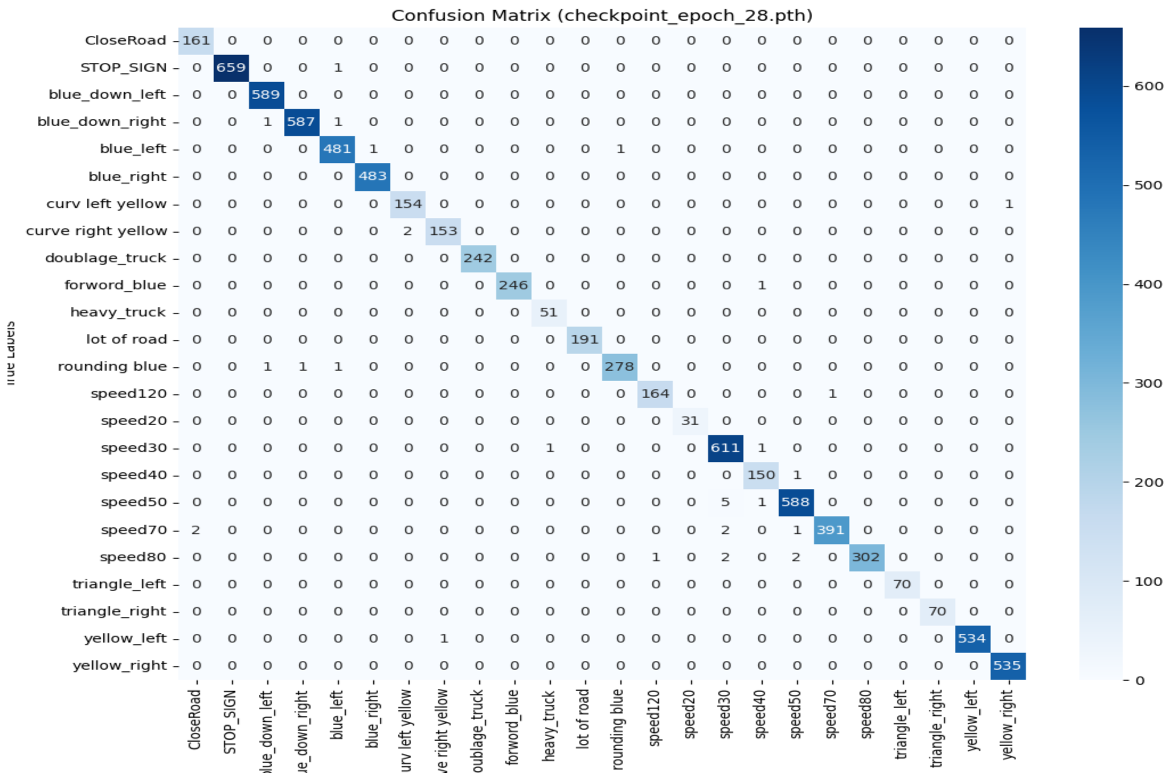
3.2.2.1 Testing With Normal Data

We tested the model using a dataset of 7,500 images, augmented with transformations similar to the training set

- Horizontal flipping (left-right),
- Curving (left-right),
- Scaling (+5%),
- Rotation (5-20 degrees),
- Addition of noise (5%),
- RGB-to-GBR color shift.

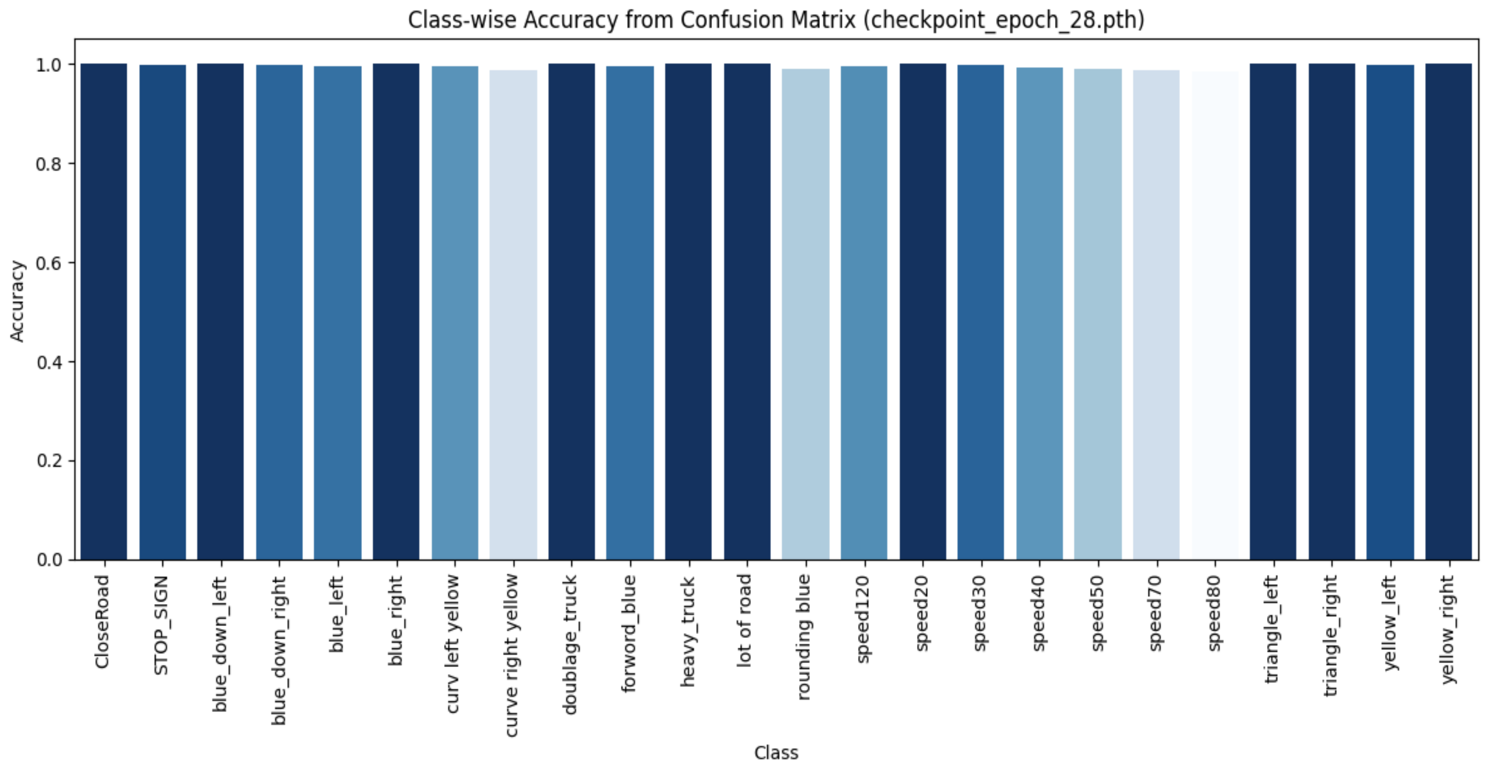
For more information about the results of each epoch click on the link : https://drive.google.com/file/d/1WwQ_oxf0fNkquVYQL-BkflAUZhABMfwc/view?usp=drive_link

- **confusion matrix**

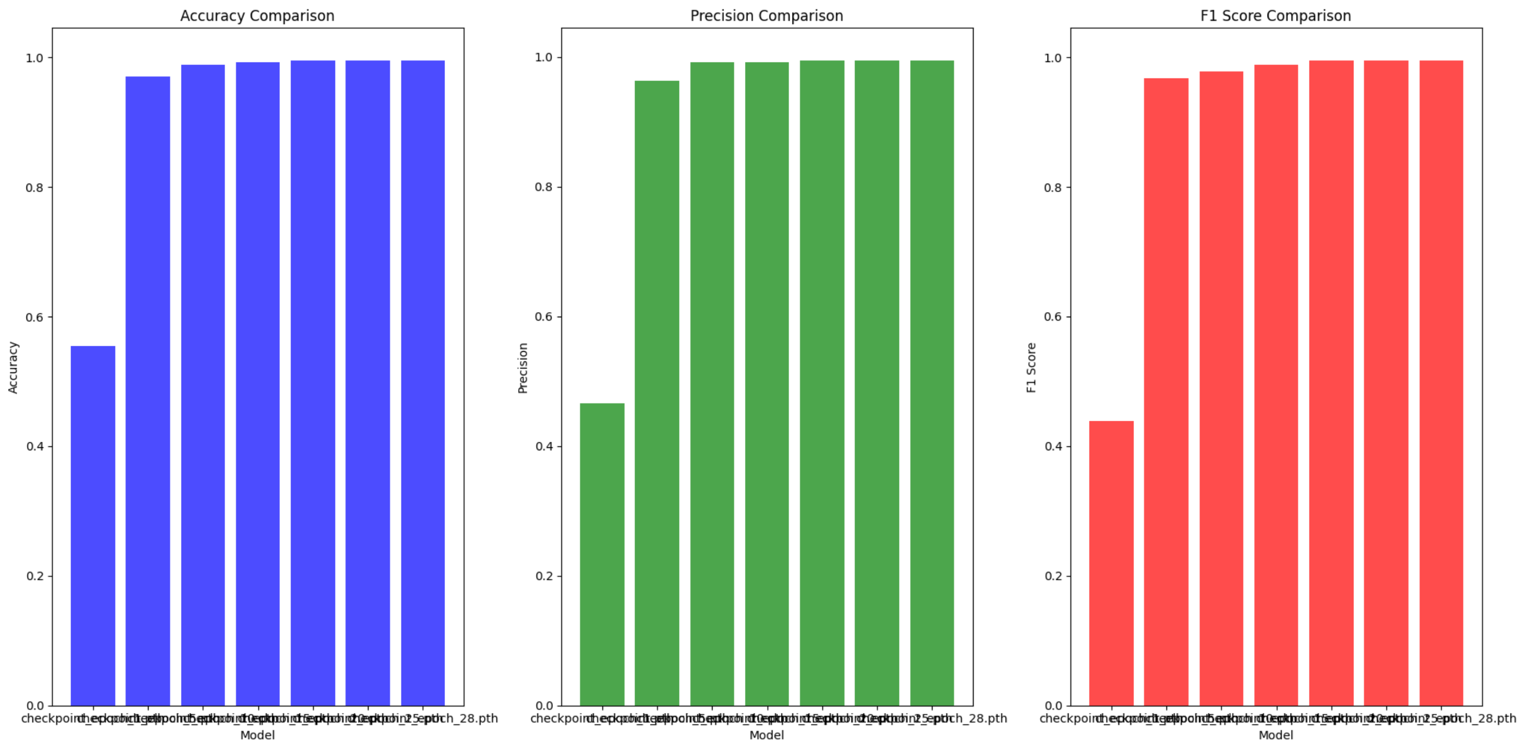


	CloseRoad	STOP_SIGN	lue_down_left	ue_down_right	blue_left	blue_right	urv left yellow	ve right yellow	oublage_truck	forword_blue	heavy_truck	lot of road	rounding blue	speed120	speed20	speed30	speed40	speed50	speed70	speed80	triangle_left	triangle_right	yellow_left	yellow_right
CloseRoad	161	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
STOP_SIGN	0	659	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
blue_down_left	0	0	589	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
blue_down_right	0	0	1	587	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
blue_left	0	0	0	0	481	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
blue_right	0	0	0	0	0	483	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
curv left yellow	0	0	0	0	0	0	154	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
curve right yellow	0	0	0	0	0	0	2	153	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
doublage_truck	0	0	0	0	0	0	0	0	242	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
forword_blue	0	0	0	0	0	0	0	0	0	246	0	0	0	0	0	0	1	0	0	0	0	0	0	0
heavy_truck	0	0	0	0	0	0	0	0	0	0	51	0	0	0	0	0	0	0	0	0	0	0	0	0
lot of road	0	0	0	0	0	0	0	0	0	0	0	191	0	0	0	0	0	0	0	0	0	0	0	0
rounding blue	0	0	1	1	1	0	0	0	0	0	0	0	278	0	0	0	0	0	0	0	0	0	0	0
speed120	0	0	0	0	0	0	0	0	0	0	0	0	0	164	0	0	0	0	1	0	0	0	0	0
speed20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	31	0	0	0	0	0	0	0	0	0
speed30	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	611	1	0	0	0	0	0	0	0
speed40	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	150	1	0	0	0	0	0	0
speed50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	1	588	0	0	0	0	0	0
speed70	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	1	391	0	0	0	0	0
speed80	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	2	0	2	0	302	0	0	0	0
triangle_left	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	70	0	0	0
triangle_right	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	70	0	0
yellow_left	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	534	0
yellow_right	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	535

- **confusion matrix bar chart**



- **Model comparaison**



3.2.2.2 Testing With Highly Enhanced Data

Augmentation Techniques:

- Flip: Left-right.
- Curve: Left-right.
- Scaling: 20%.
- Rotation: 0–180 degrees.
- Noise: 80%.
- Grayscale: Yes.
- Color Shift: From RGB to GBR.

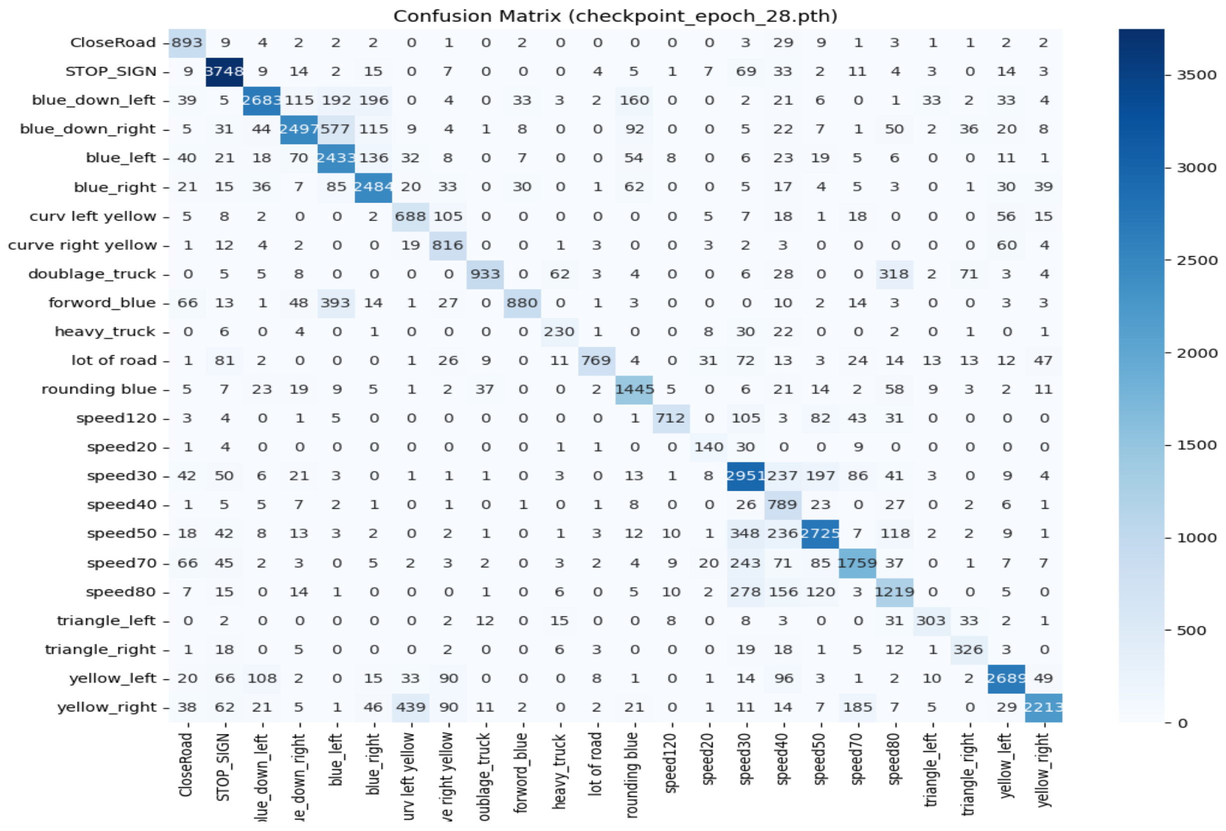
Test Set:

- Highly augmented with **46,000 images**.

For more information about the results of each epoch click on the link :https://drive.google.com/file/d/1AOw2dHeu5JkfEGyh1J9z1AOYI_BhpeaD/view?usp=drive_link

For more information about the hyperparamater click on the link :https://drive.google.com/file/d/1VU_kQvDozNoBtMTUvBT0cgFLtAlb8CIk/view?usp=drive_link

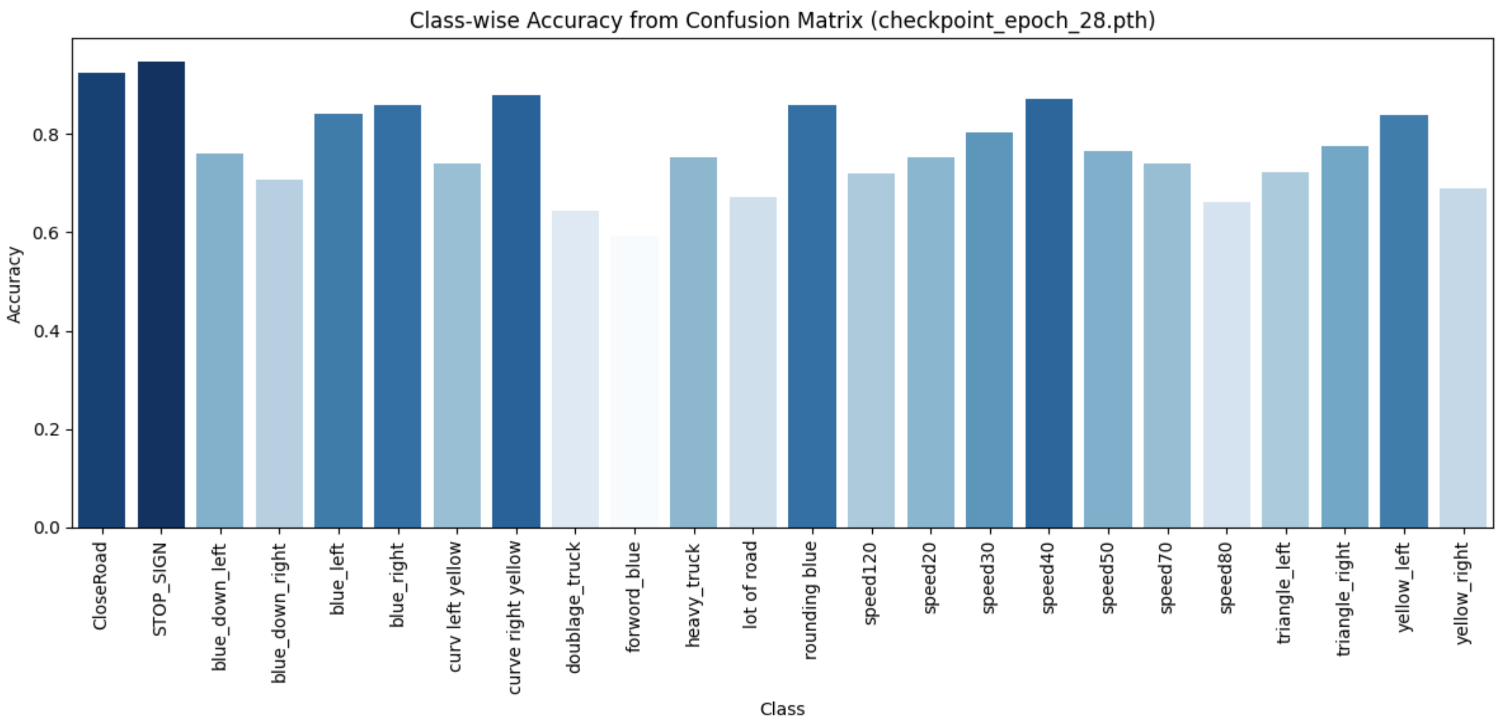
- **confusion matrix**



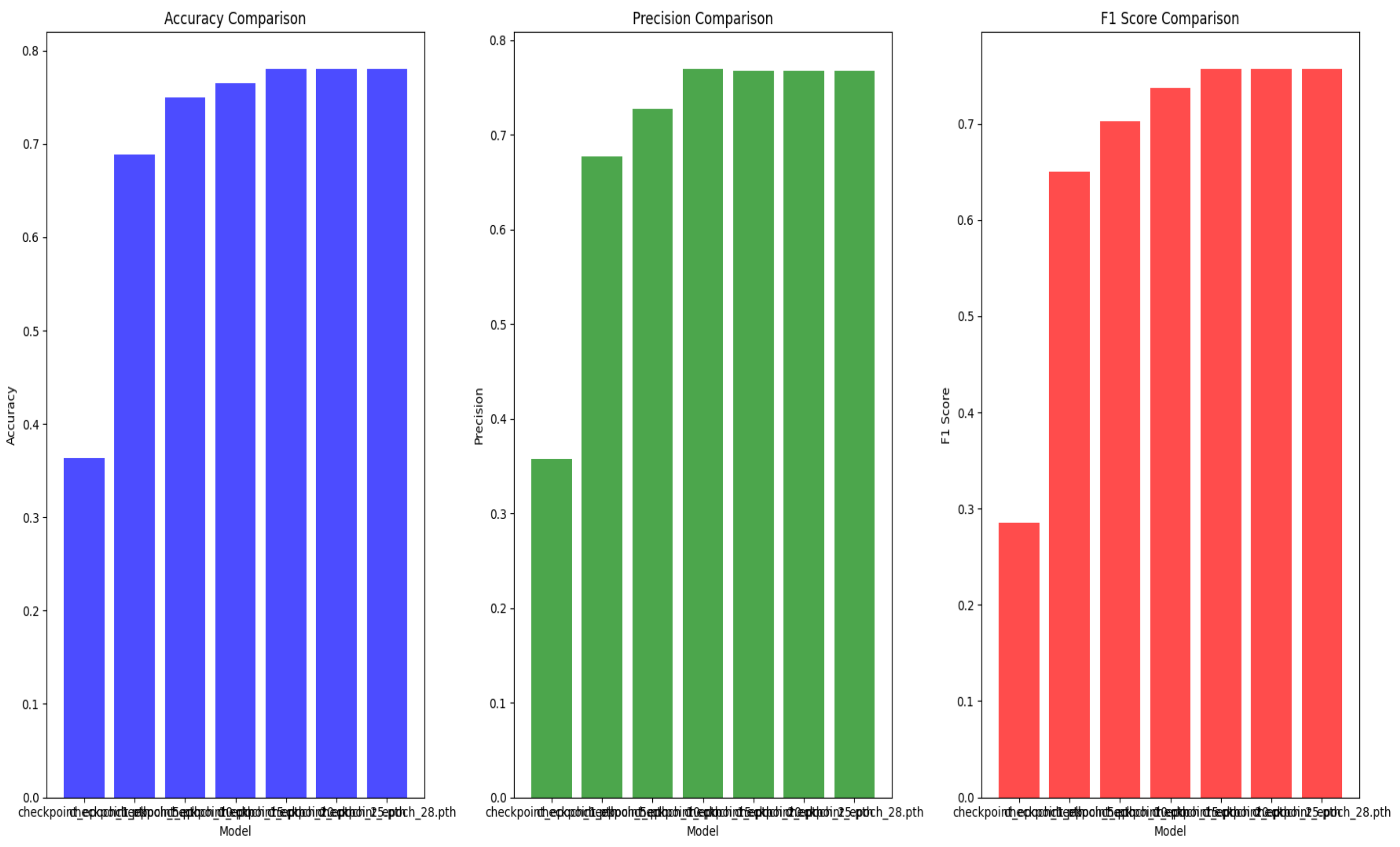
Confusion Matrix (checkpoint_epoch_28.pth)

	CloseRoad	STOP_SIGN	lue_down_left	e_down_right	blue_left	blue_right	urv left yellow	ve right yellow	oublage_truck	forword_blue	heavy_truck	lot of road	rounding blue	speed120	speed20	speed30	speed40	speed50	speed70	speed80	triangle_left	triangle_right	yellow_left	yellow_right
CloseRoad	893	9	4	2	2	2	0	1	0	2	0	0	0	0	0	3	29	9	1	3	1	1	2	2
STOP_SIGN	9	3748	9	14	2	15	0	7	0	0	0	4	5	1	7	69	33	2	11	4	3	0	14	3
blue_down_left	39	5	2683	115	192	196	0	4	0	33	3	2	160	0	0	2	21	6	0	1	33	2	33	4
blue_down_right	5	31	44	2497	577	115	9	4	1	8	0	0	92	0	0	5	22	7	1	50	2	36	20	8
blue_left	40	21	18	70	2433	136	32	8	0	7	0	0	54	8	0	6	23	19	5	6	0	0	11	1
blue_right	21	15	36	7	85	2484	20	33	0	30	0	1	62	0	0	5	17	4	5	3	0	1	30	39
curv left yellow	5	8	2	0	0	2	688	105	0	0	0	0	0	0	5	7	18	1	18	0	0	0	56	15
curve right yellow	1	12	4	2	0	0	19	816	0	0	1	3	0	0	3	2	3	0	0	0	0	0	60	4
doublage_truck	0	5	5	8	0	0	0	0	933	0	62	3	4	0	0	6	28	0	0	318	2	71	3	4
forword_blue	66	13	1	48	393	14	1	27	0	880	0	1	3	0	0	0	10	2	14	3	0	0	3	3
heavy_truck	0	6	0	4	0	1	0	0	0	0	230	1	0	0	8	30	22	0	0	2	0	1	0	1
lot of road	1	81	2	0	0	0	1	26	9	0	11	769	4	0	31	72	13	3	24	14	13	13	12	47
rounding blue	5	7	23	19	9	5	1	2	37	0	0	2	1445	5	0	6	21	14	2	58	9	3	2	11
speed120	3	4	0	1	5	0	0	0	0	0	0	0	1	712	0	105	3	82	43	31	0	0	0	0
speed20	1	4	0	0	0	0	0	0	0	0	1	1	0	0	140	30	0	0	9	0	0	0	0	0
speed30	42	50	6	21	3	0	1	1	1	0	3	0	13	1	8	2951	237	197	86	41	3	0	9	4
speed40	1	5	5	7	2	1	0	1	0	1	0	1	8	0	0	26	789	23	0	27	0	2	6	1
speed50	18	42	8	13	3	2	0	2	1	0	1	3	12	10	1	348	236	2725	7	118	2	2	9	1
speed70	66	45	2	3	0	5	2	3	2	0	3	2	4	9	20	243	71	85	1759	37	0	1	7	7
speed80	7	15	0	14	1	0	0	0	1	0	6	0	5	10	2	278	156	120	3	1219	0	0	5	0
triangle_left	0	2	0	0	0	0	0	2	12	0	15	0	0	8	0	8	3	0	0	31	303	33	2	1
triangle_right	1	18	0	5	0	0	0	2	0	0	6	3	0	0	0	19	18	1	5	12	1	326	3	0
yellow_left	20	66	108	2	0	15	33	90	0	0	0	8	1	0	1	14	96	3	1	2	10	2	2689	49
yellow_right	38	62	21	5	1	46	439	90	11	2	0	2	21	0	1	11	14	7	185	7	5	0	29	2213

- **confusion matrix bar chart**



- **Model comparaison**



4. Integration of Detection and Classify Model

4.1 Motivation

To enhance the system's performance and modularity, we implemented a dual-model pipeline:

1. **YOLO Model for Detection**: This model is exclusively responsible for detecting and localizing traffic signs in an image.
2. **Classification Model**: The classification task is delegated to the pre-trained model, ensuring high accuracy in identifying traffic sign categories.

This division optimizes the workflow by leveraging YOLO’s real-time detection capabilities and the classification model’s specialized performance.

4.2 Detection Model (YOLO)

4.2.1 Train The YOLOv8m Model

Dataset :

For training the YOLOv8m model, we used the dataset available at **Roboflow - Traffic Signs and Traffic Lights Dataset v2**.

[https://universe.roboflow.com/sonia-yv7rn/traffic-signs-and-traffic-lights-ukvmt/dataset/2]

Dataset Content:

- Includes annotated images of traffic signs and traffic lights, with bounding boxes for each object.
- **Classes**: Covers all necessary traffic sign types and traffic lights relevant to the project. (47 classes)
- **Training Set Size**: 4883 images used for training.
- **Validation Set Size**: 1051 images for validation.
- **Test Set Size**: 1003 images for testing.

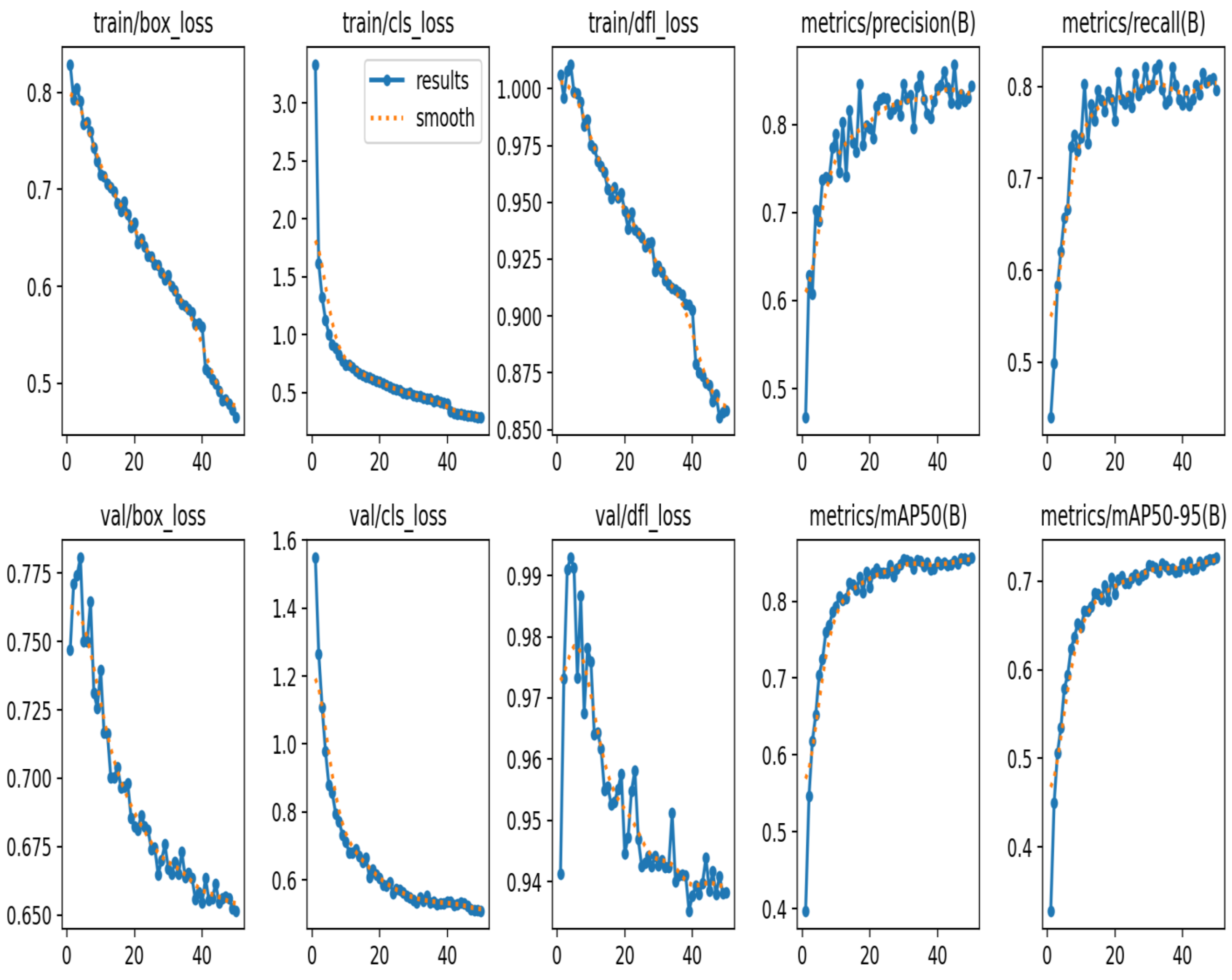
Model Configuration:

We utilized YOLOv8m (medium-sized model) for detection with the following parameters:

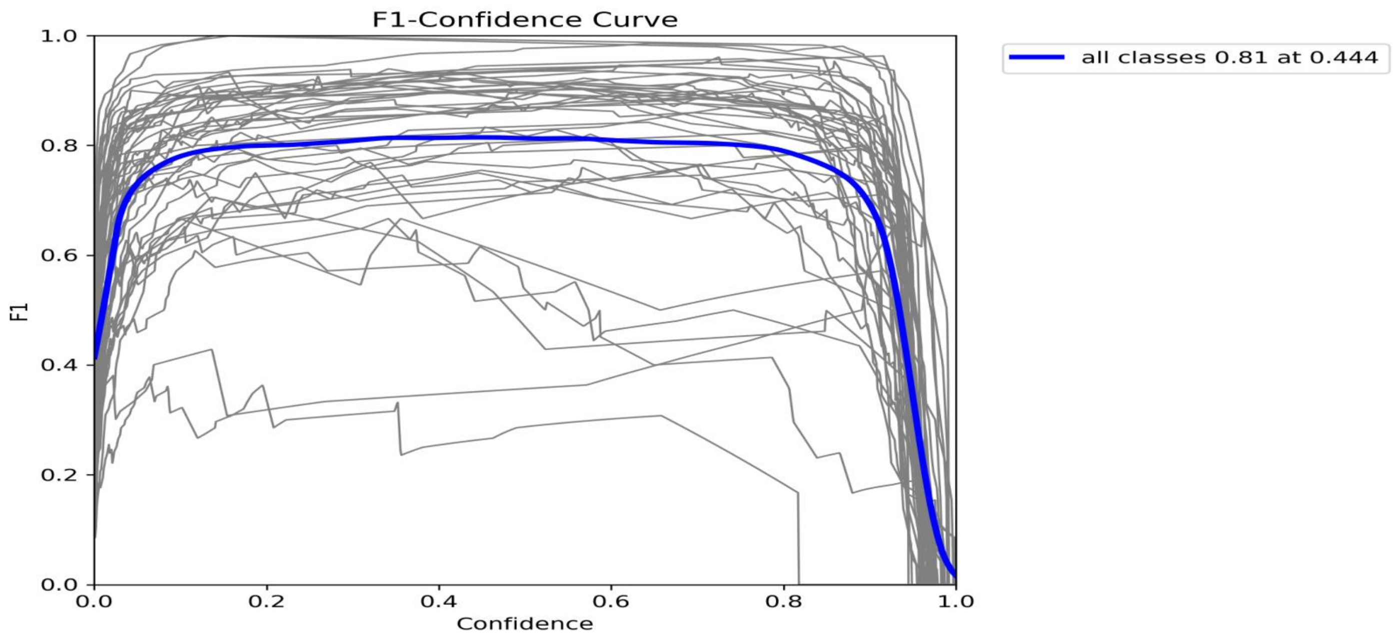
- **Epochs**: 50
- **Input Size**: 640x640 (default)
- **Batch Size**: 16 (default)
- **Optimizer**: AdamW (default)
- **Learning Rate**: Adaptive learning rate starting at 0.01 (default)

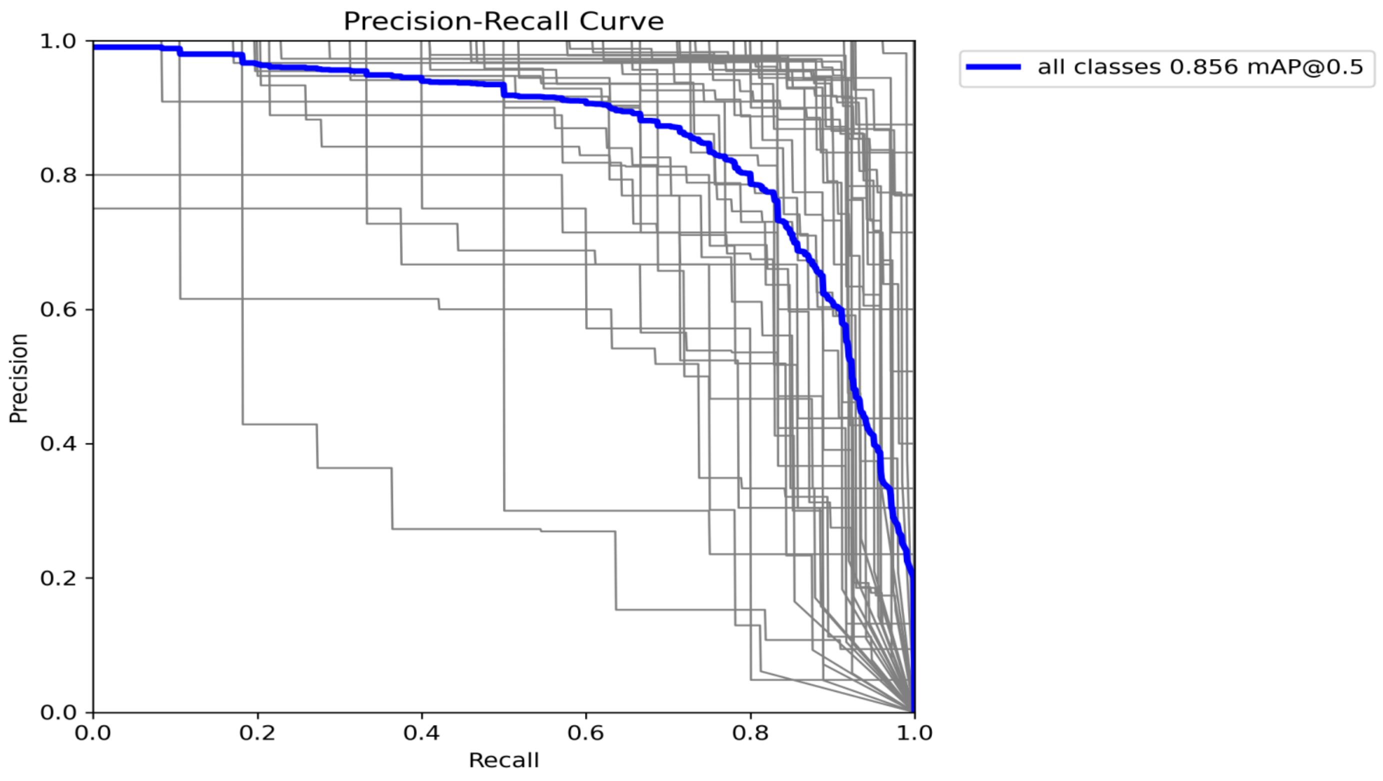
4.2.2 Results: The detection model achieved the following metrics:

- **train/box_loss:** 0.46475
- **train/cls_loss:** 0.2835
- **train/dfl_loss:** 0.85841
- **metrics/precision(B):** 0.84355
- **metrics/recall(B)**: 0.79643
- **metrics/mAP50(B):** 0.85669
- **metrics/mAP50-95(B):** 0.72631
- **val/box_loss**: 0.65161
- **val/cls_loss:** 0.50905
- **val/dfl_loss:** 0.93826



- **F1-Score :** 81%
- **Mean Average Precision (mAP)**: 85.6% at IoU 0.5.
- **Precision**: 100%
- **Recall**: 94%





For more information about the results and the plots click on the link :YOLO Train Results

[https://drive.google.com/drive/folders/1cO-4NRAV2foiTNfx4CwSoPJ9DmbEZAZX]

4.3 Pipeline Integration

The output from the YOLO model (bounding box coordinates) is passed to the classification model for label prediction. The integration works as follows:

1. YOLO detects objects in real-time and outputs bounding boxes and confidence scores.
2. Detected regions are cropped and resized, then sent to the classification model.
3. The classification model predicts the traffic sign class for each bounding box.

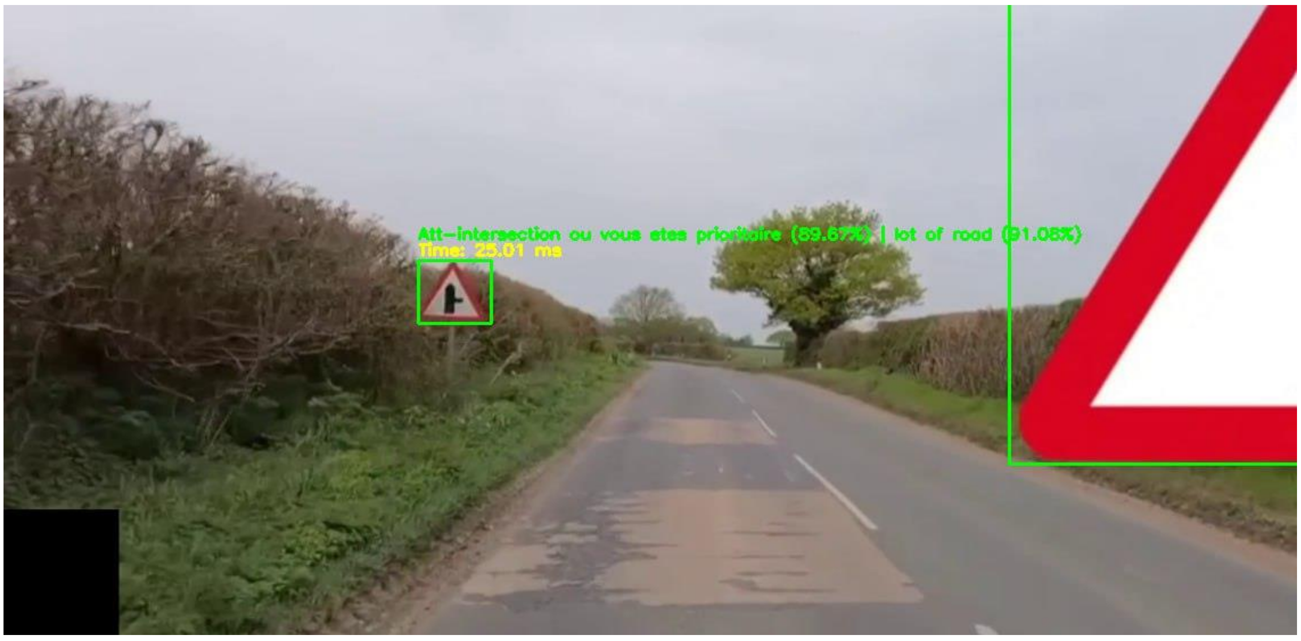
Include a flowchart showing the detection-classification pipeline.

4.4 Real-World Testing and Performance

We tested the combined system on real-world scenarios, including complex conditions such as occlusion, lighting variations, and overlapping signs. The combined system demonstrated:

- **Detection Speed**: 3 FPS [around 250 - 450 ms for YOLO and 20 - 35 ms for classify model]
- **Classification Accuracy**: (70% - 99%)

Example:



Att-intersection ou vous etes prioritaire (89.67%) | lot of road (91.08%)
Time: 25.01 ms

- **Detection Speed**: 400ms (YOLOv8m) and 25ms   (classify model)
- **Classification Accuracy**: 89% (YOLOv8m), 91% (classify Model)

5. General Conclusion

Our project offers machine learning that allows the user to recognize traffic signs by capturing or uploading them. For this reason, in the first chapter we gave an overview of the sources we used to fetch the data. The second chapter is devoted to the preparatory aspect of creating the model by using the appropriate language with its libraries.

In the third chapter, we put the output of the project which represents the confuction matrix, the confuction matrix chart, and the comparison of the model.

This project was the subject of an interesting experiment that gave us the opportunity to improve our knowledge and skills in developing and designing machine learning.

Therefore, we consider that we have reached our main goal, which is to build a model that meets the user's needs by helping to identify traffic signs and we plan to improve the application through the actual experience of many users.