ENSEIRB-MATMECA
Bordeaux INP

Rapport Projet :

Détection en temps réel des panneaux de signalisation par intelligence artificielle embarquée

Étudiant : Nejma Rachdi & Malek Bejaoui

Année universitaire : 2025–2026

Filière : Systèmes Embarqués

Encadrant : Kadionik Patrice

Rapport de Projet avancé S9

Contents

1 Introduction

1.1 Contexte général

Ces dernières années, l'intelligence artificielle embarquée, également appelée *Edge AI*, connaît un fort développement dans de nombreux domaines tels que les systèmes autonomes, la robotique, la vision par ordinateur et les systèmes d'aide à la conduite. Contrairement aux approches basées sur le cloud, l'IA embarquée impose des contraintes importantes en termes de puissance de calcul, de mémoire, de consommation énergétique et de latence, tout en devant garantir un fonctionnement en temps réel.

La détection d'objets constitue l'un des cas d'usage majeurs de la vision par ordinateur. Elle est essentielle pour de nombreuses applications critiques, notamment dans le contexte des véhicules autonomes et des systèmes de perception embarqués. Le déploiement de modèles de détection performants sur des plateformes à ressources limitées représente ainsi un enjeu technique majeur.

Dans ce contexte, les architectures de type YOLO (*You Only Look Once*) se distinguent par leur capacité à effectuer une détection rapide et efficace en une seule passe, ce qui les rend particulièrement adaptées aux systèmes embarqués.

1.2 Motivation du projet

La motivation principale de ce projet est d'étudier la faisabilité du déploiement d'un modèle de détection d'objets basé sur l'intelligence artificielle sur des plateformes embarquées hétérogènes, telles que le Raspberry Pi, tout en conservant des performances compatibles avec des applications en temps réel.

Les cartes Raspberry Pi, largement utilisées dans le prototypage et l'embarqué, offrent un compromis intéressant entre coût, consommation énergétique et capacités de calcul. Toutefois, leurs ressources restent limitées par rapport à un ordinateur classique, ce qui impose une optimisation rigoureuse du modèle et de son format de déploiement.

Ce projet s'inscrit ainsi dans une démarche d'ingénierie visant à comprendre l'impact du matériel et du format du modèle sur les performances d'inférence, et à identifier les solutions permettant d'améliorer le débit de traitement sur des systèmes embarqués.

1.3 Objectifs du projet

L'objectif principal de ce projet est de déployer et d'évaluer un modèle de détection d'objets sur différentes plateformes embarquées et de comparer leurs performances.

Plus précisément, les objectifs sont les suivants :

- Déployer un modèle de détection d'objets basé sur plusieurs plateformes matérielles, notamment un PC(un processeur Intel Xeon E3-1241 v3 @ 3.50GHz 8 Go RAM.), un Raspberry Pi 4 et un Raspberry Pi 5 (8Go RAM).
- Comparer les performances obtenues sur ces plateformes en termes de débit d'inférence (FPS) et de fluidité d'exécution.
- Exporter le modèle entraîné sous différents formats, tels que *Pytorch*, *Float32*, *ONNX*, *NCNN* et *OpenVino INT8* afin d'évaluer leur impact sur les performances.
- Analyser l'influence du format du modèle et du matériel sur les résultats obtenus.
- Atteindre un débit d'inférence supérieur à 10 images par seconde (FPS), seuil considéré comme acceptable pour une application de détection quasi temps réel sur système embarqué.

2 Présentation du système et environnement de développement

2.1 Présentation du système

Le système conçu dans le cadre de ce projet vise à réaliser une détection automatique de panneaux de signalisation en temps réel sur des plateformes aux ressources limitées. Pour garantir une faible latence et une exécution locale sans dépendance au cloud, le système repose sur une chaîne de traitement séquentielle optimisée.

Comme illustré par la Figure 1, le fonctionnement du système suit quatre étapes principales :

- **Caméra :** Capture physique de l'environnement via le capteur.
- **Acquisition vidéo :** Traitement du flux d'entrée pour préparer les images à l'inférence.
- **Modèle YOLO :** Inférence du modèle entraîné pour identifier et localiser les objets présents dans les trames.
- **Détection et affichage :** Superposition des boîtes englobantes et des classes détectées sur le flux vidéo final pour l'utilisateur.

La chaîne complète du système est la suivante :

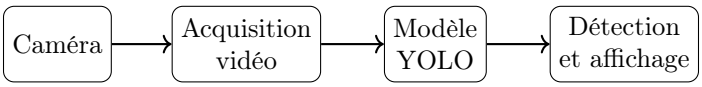

Figure 1: Chaîne de traitement du système de détection d'objets

2.2 Architecture du modèle de détection

Le système repose sur l'utilisation d'un **modèle de détection d'objets pré-entraîné**. Dans ce projet, le modèle **YOLO**, fourni par la bibliothèque **Ultralytics**, a été choisi comme modèle de détection.

Le choix de YOLO est motivé par sa capacité à réaliser la détection d'objets en une seule passe (*You Only Look Once*), ce qui permet de réduire la latence et d'atteindre des performances compatibles avec des applications temps réel sur des plateformes embarquées à ressources limitées.

Licence du framework YOLO

Les modèles YOLOv8 fournis par Ultralytics sont distribués sous un système de double licence.

- **Licence AGPL-3.0 (Open Source)** : utilisée dans le cadre de notre projet à l'ENSEIRB-MATMECA. Elle est gratuite pour un usage académique et de recherche, mais impose le partage du code source si le projet est distribué.
- **Licence Entreprise (Commerciale)** : licence payante permettant une utilisation dans des produits propriétaires sans obligation de partage du code source.

Dans ce projet, l'option open source a été retenue car elle est la plus adaptée au contexte académique et expérimental de nos travaux.

La Figure 2 illustre l'architecture générale du modèle YOLO.

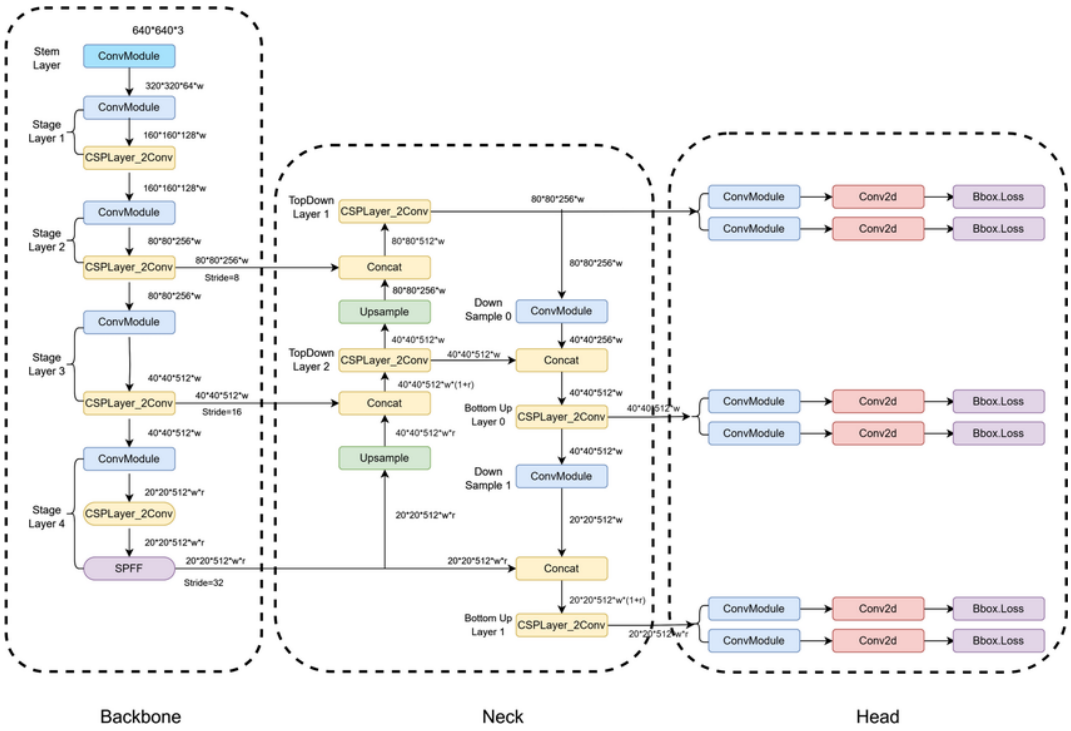

Figure 2: Architecture générale du modèle YOLO

L'architecture de YOLO se compose de trois blocs principaux :

- **Backbone** : extraction des caractéristiques visuelles à partir de l'image d'entrée.
- **Neck** : fusion des caractéristiques à différentes échelles afin d'améliorer la détection d'objets de tailles variées.
- **Head** : prédiction des boîtes englobantes, des classes et des scores de confiance.

Cette architecture permet une exécution locale sans dépendance au cloud, garantissant une faible latence.

2.3 Choix du modèle de détection

Une comparaison des différentes variantes du modèle YOLOv8 est présentée à la Figure 3. Cette comparaison met en évidence le compromis entre précision, complexité de calcul et vitesse d'inférence.

Parmi les différentes variantes du modèle YOLOv8, le modèle **YOLOv8n** a été retenu pour ce projet. Le tableau de comparaison montre que YOLOv8n possède le plus faible nombre de paramètres ainsi que le plus faible coût de calcul (FLOPs), ce qui le rend particulièrement adapté aux plateformes embarquées à ressources limitées.

Bien que les modèles de plus grande taille offrent une meilleure précision, leur complexité de calcul élevée les rend moins compatibles avec un déploiement temps réel sur Raspberry Pi. Le choix de YOLOv8n représente ainsi un compromis pertinent entre précision et performances, en cohérence avec l'objectif du projet d'atteindre un débit d'inférence supérieur à 10 images par seconde.

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Figure 3: Comparaison des différentes variantes du modèle YOLOv8 en termes de précision, de complexité et de coût de calcul (source : Ultralytics)

2.4 Environnement matériel et logiciel

Trois plateformes matérielles ont été utilisées afin de comparer les performances du modèle de détection.

- Plateformes matérielles utilisées
  - PC: utilisé comme plateforme de référence pour l'entraînement et l'évaluation des performances.
  - Raspberry Pi 4 8 Go ram : plateforme embarquée largement utilisée, servant de référence en environnement contraint.
  - Raspberry Pi 5 8 Go ram : version plus récente offrant de meilleures performances de calcul
- Système d'exploitation : Raspberry Pi OS 64 bits (Debian GNU/Linux 12 – Bookworm)
- Langage : Python 3.11.2
- Caméra : Caméra CSI (capteur IMX219)
- Environnement d'exécution : environnement virtuel Python

Les détails relatifs à l'environnement matériel et logiciel, ainsi qu'à la procédure d'installation et de configuration, sont disponibles dans le fichier `README` associé au projet, Le manuel d'installation détaillé, incluant les commandes, la configuration de la caméra et la résolution des problèmes de compatibilité.

3 Dataset et préparation des données

3.1 Présentation du dataset

Le projet s'appuie sur un dataset de panneaux de signalisation routière issu de la plateforme Roboflow. Le dataset contient des images annotées destinées à l'entraînement et à l'évaluation d'un modèle de détection d'objets. Les annotations sont fournies au format compatible avec les modèles YOLO.

3.2 Décompression et organisation des données

Afin d'accélérer la phase d'entraînement du modèle, l'environnement Google Colab a été utilisé pour bénéficier de l'accélération matérielle par GPU. Le dataset est stocké sur Google Drive puis décompressé temporairement dans le répertoire `/tmp` de l'environnement Colab.

Cette approche permet de limiter l'utilisation de l'espace de stockage du Drive et d'éviter la saturation de celui-ci. Le répertoire `/tmp` étant temporaire, les données décompressées sont automatiquement supprimées après la fin de la session, ce qui constitue une bonne pratique pour la gestion de l'espace disque.

3.3 Visualisation et validation des données

Afin de valider visuellement le contenu du dataset, plusieurs images sont sélectionnées de manière aléatoire à partir du jeu de données d'entraînement et affichées sous forme de grille. Cette visualisation permet de vérifier la diversité des scènes, la qualité des images et la cohérence globale du dataset avant le lancement de l'entraînement.

Figure 4: Exemples d'images sélectionnées aléatoirement à partir du dataset

3.4 Étude d'une image du dataset

Une image issue du dataset est analysée afin de caractériser le format des données manipulées par le modèle. L'image étudiée possède une résolution de 640 × 640 pixels et est représentée sous la forme d'un tenseur 640 × 640 × 3, correspondant aux trois canaux de couleur RGB (Red, Green, Blue).

Ce format est compatible avec l'entrée du modèle YOLO utilisé dans ce projet. La normalisation et le redimensionnement sont gérés automatiquement par la bibliothèque Ultralytics lors de l'entraînement et de l'inférence.

Figure 5: Exemple d'une image du dataset

4 Entraînement du modèle de détection

4.1 Choix de l'environnement d'entraînement

L'entraînement du modèle est réalisé sur **Google Colab** afin de bénéficier de l'accélération matérielle par GPU. Ce choix permet de réduire considérablement le temps d'entraînement par rapport à une exécution sur plateforme embarquée, tout en générant un modèle directement exploitable sur Raspberry Pi.

4.2 Organisation du dataset

Le dataset utilisé pour l'entraînement est structuré selon le format standard attendu par les modèles YOLO. Il est divisé en trois sous-ensembles distincts : entraînement, validation et test.

```
dataset/
|-- train/
|   |-- images/--> 3530
|   `-- labels/
|-- valid/
|   |-- images/--> 801
|   `-- labels/
|-- test/
|   |-- images/--> 638
|   `-- labels/
|-- data.yaml
|-- README.dataset
`-- README.roboflow
```

Le fichier `data.yaml` définit les chemins vers les différents ensembles ainsi que les classes d'objets utilisées lors de l'entraînement.

4.3 Configuration de l'entraînement

L'entraînement du modèle repose sur une approche de **fine-tuning** à partir du modèle pré-entraîné YOLOv8n. Cette méthode consiste à réutiliser des poids déjà entraînés sur un large jeu de données générique, puis à les adapter au dataset spécifique de panneaux de signalisation utilisé dans ce projet.

Le dataset est décrit par le fichier `data.yaml`, qui définit les chemins vers les ensembles d'entraînement, de validation et de test. L'apprentissage est réalisé sur **30 époques**, ce qui permet d'obtenir un compromis entre qualité de détection et temps d'entraînement.

La taille du batch ainsi que l'optimiseur sont automatiquement sélectionnés par la bibliothèque Ultralytics en fonction des ressources GPU disponibles sur Google Colab. Le modèle issu du fine-tuning est sauvegardé sous le nom `first_model` afin d'être utilisé pour les étapes d'export et de déploiement.

```
1
2 model = YOLO("yolov8n.pt")
3
4 Result_Final_model = model.train(
5     data="/tmp/S9/data.yaml",
6     epochs=30,
7     batch=-1,
8     optimizer="auto",
9     name="first_model"
10 )
```

Les paramètres susceptibles d'être ajustés lors de la phase d'entraînement sont détaillés en 8.

5 Validation du modèle

5.1 Procédure de validation

À l'issue de l'entraînement, le modèle est évalué sur le jeu de validation afin de mesurer ses performances et sa capacité de généralisation. Cette phase de validation est automatiquement réalisée par la bibliothèque Ultralytics.

Les résultats de validation sont générés sous forme de courbes de performance, de prédictions sur des images du jeu de validation et de matrices de confusion.

5.2 Courbes de performance

Les performances du modèle sont analysées à l'aide de différentes courbes générées lors de la phase de validation. Ces courbes permettent d'évaluer le compromis entre précision et rappel ainsi que le comportement global du modèle en fonction du seuil de confiance.

5.2.1 Analyse de la courbe Précision–Rappel

La Figure 6 présente la courbe Précision–Rappel obtenue sur le jeu de validation.

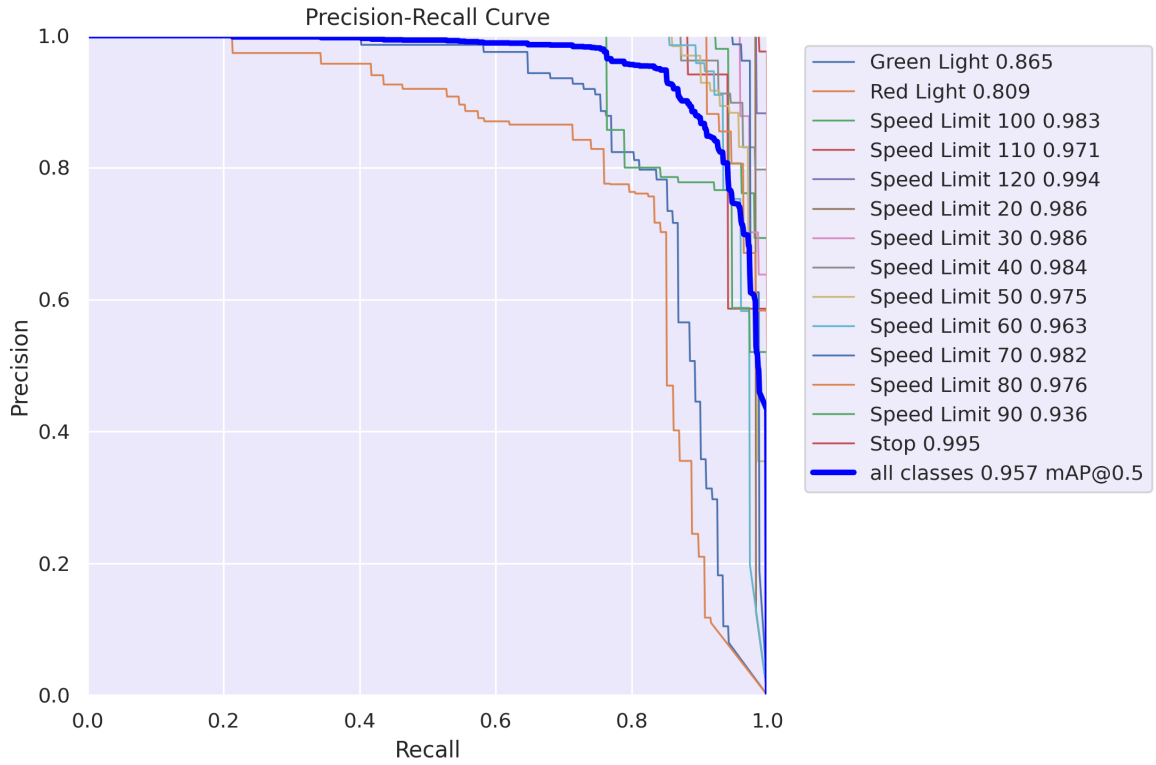

Figure 6: Courbe Précision–Rappel obtenue sur le jeu de validation

Cette courbe met en évidence le compromis entre la précision (proportion de détections correctes parmi les détections effectuées) et le rappel (proportion d'objets effectivement détectés parmi les objets présents).

On observe que, pour la majorité des classes, la précision reste élevée sur une large plage de valeurs de rappel, ce qui traduit une bonne capacité du modèle à détecter les objets tout en limitant les faux positifs. En particulier, certaines classes telles que Stop ou les panneaux de limitation de vitesse présentent des performances très élevées, avec des valeurs d'AP proches de 1, indiquant une séparation efficace entre les objets d'intérêt et l'arrière-plan.

La courbe globale, représentée par la moyenne sur l'ensemble des classes, atteint une valeur de mAP@0.5 de 0.957, ce qui reflète un excellent niveau de performance globale du modèle. La chute de la précision observée pour des valeurs de rappel proches de 1 est un comportement attendu, correspondant à l'augmentation des faux positifs lorsque le seuil de confiance est fortement abaissé.

Ces résultats montrent que le modèle offre un bon équilibre entre précision et rappel, et qu'il est particulièrement adapté à une utilisation en détection d'objets en environnement réel

5.2.2 Matrice de confusion

La Figure suivante présente la matrice de confusion normalisée obtenue sur le jeu de validation. Cette matrice permet d'analyser plus finement les performances du modèle en comparant, pour chaque classe réelle, la proportion de prédictions correctes et les confusions éventuelles avec les autres classes.

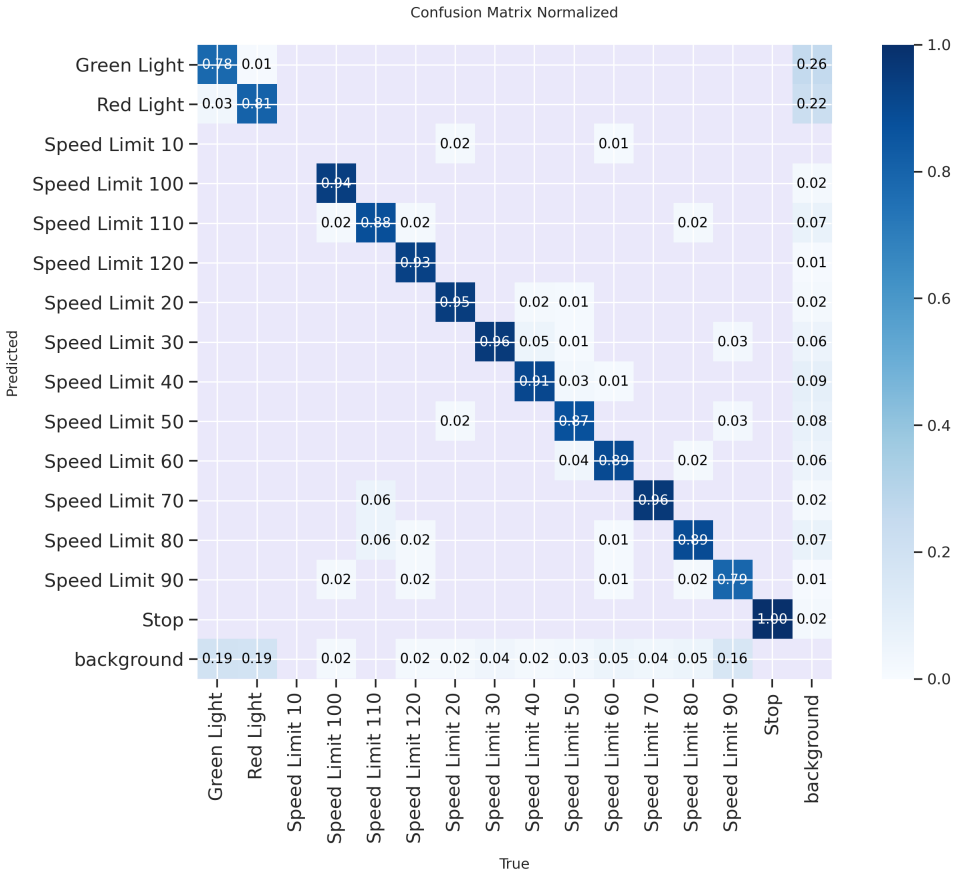

Figure 7: Matrice de confusion normalisée obtenue sur le jeu de validation

On observe une forte concentration des valeurs sur la diagonale principale, ce qui indique un taux élevé de classifications correctes pour la majorité des classes. En particulier, les panneaux de limitation de vitesse ainsi que la classe Stop présentent des taux de reconnaissance très élevés, souvent supérieurs à 0,9, confirmant la bonne capacité du modèle à discriminer ces catégories visuellement proches.

Les principales confusions apparaissent entre certaines classes de limitations de vitesse voisines (par exemple 30, 40, 50 ou 70 km/h), ce qui est cohérent avec la similarité visuelle de ces panneaux. Ces erreurs restent toutefois limitées et représentent une faible proportion des prédictions totales.

Concernant les classes Green Light et Red Light, la matrice met en évidence quelques confusions résiduelles, principalement dues à des variations d'éclairage, d'angle de prise de vue ou de conditions environnementales. Néanmoins, les taux de classification corrects demeurent élevés, traduisant une bonne robustesse du modèle face à ces variations.

Enfin, la présence de valeurs non nulles associées à la classe background indique que certaines détections correspondent à des faux positifs ou à des objets partiellement visibles. Ce phénomène est attendu dans un contexte de détection d'objets et reste maîtrisé au vu des performances globales observées précédemment.

Dans l'ensemble, cette matrice de confusion confirme les excellentes performances du modèle, en cohérence avec l'analyse de la courbe Précision–Rappel, et met en évidence un comportement fiable pour une application de détection d'objets en conditions réelles.

5.3 Analyse visuelle des prédictions

Des exemples de prédictions sont générés sur des images du jeu de validation afin de vérifier visuellement la qualité de la détection. Ces résultats permettent de comparer les annotations réelles aux prédictions du modèle.

Cette analyse visuelle constitue une validation qualitative du bon fonctionnement du modèle avant son export et son déploiement.

Figure 8: Exemples de prédictions du modèle sur le jeu de validation

6 Intégration et tests sur plateformes embarquées

Dans cette étape, notre mission était de faire sortir le modèle de l'ordinateur de bureau pour le faire tourner sur nos cartes Raspberry Pi. C'est un vrai défi technique, car un modèle d'IA "brut" est souvent trop lourd pour ces petits processeurs. Nous avons donc dû transformer notre travail pour l'adapter aux capacités de nos cibles.

6.1 Export du modèle et formats utilisés

Une fois l'entraînement terminé, nous avons obtenu un fichier PyTorch (`first_model.pt`). C'est notre fichier de base, mais il n'est pas du tout optimisé pour le matériel embarqué. Pour que la détection soit fluide et rapide, nous avons converti ce fichier dans différents formats.

L'idée pour nous était de tester plusieurs "recettes" pour trouver celle qui offre le meilleur équilibre entre vitesse et précision :

- **PyTorch (.pt) :** C'est le modèle original. Nous l'avons gardé comme référence pour comparer les résultats, mais nous avons vite remarqué qu'il faisait beaucoup trop ramer la Raspberry Pi.
- **TensorFlow Lite Float32 :** C'est un format très populaire pour les appareils mobiles. Nous l'avons choisi car il est plus simple à lire pour la carte tout en gardant exactement la même qualité de détection que le modèle de départ.
- **ONNX :** Nous avons utilisé ce format comme un "traducteur" universel. Il nous a permis de rendre notre modèle indépendant du logiciel d'origine et de préparer plus facilement les autres conversions.
- **NCNN :** C'est sans doute le format qui nous intéressait le plus. Il est spécialement conçu pour les processeurs de type ARM (comme ceux de nos Raspberry Pi). Son rôle est d'accélérer les calculs au maximum pour que la vidéo ne saccade pas lors de la détection.
- **OpenVINO INT8 :** Ici, nous avons testé une méthode un peu plus radicale appelée "quantification". Au lieu d'utiliser des chiffres très précis (32 bits), nous avons simplifié les données du modèle en 8 bits. Cela rend le modèle beaucoup plus léger et rapide à calculer. Notre but était de voir si ce gain de vitesse valait la petite perte de précision que cela peut engendrer.

Pour illustrer l'impact de ces choix, la figure suivante (issue de la documentation officielle d'Ultralytics) présente une étude du temps d'inférence en fonction du format exporté pour différents modèles.

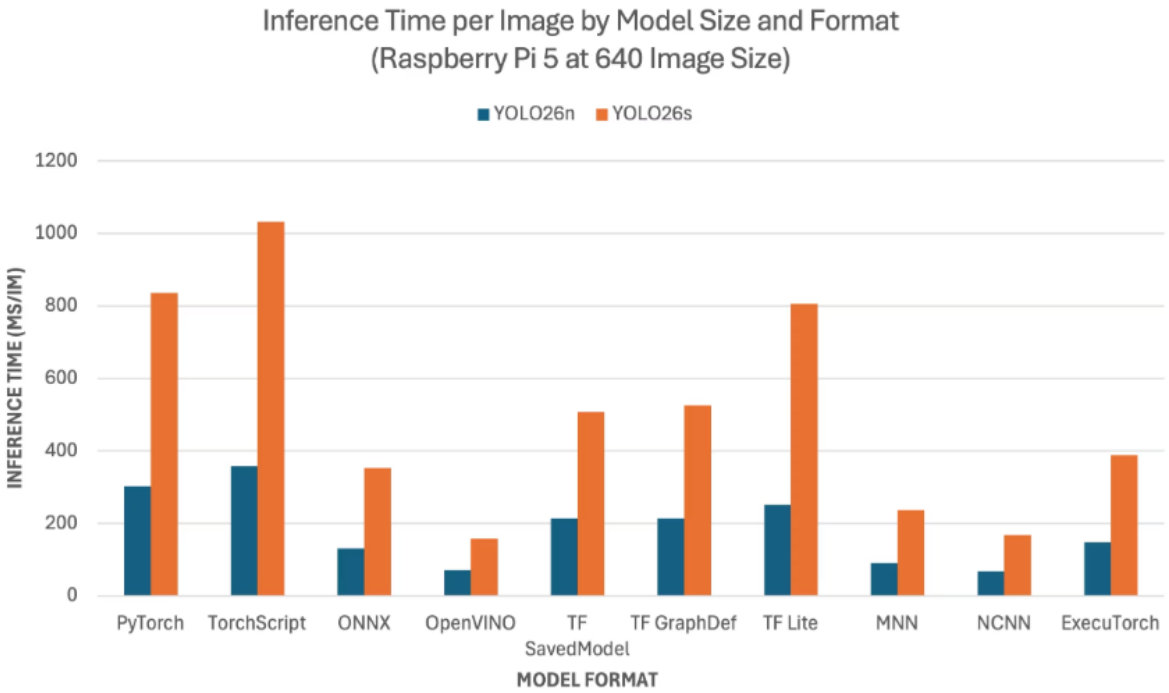

Figure 9: Étude du temps d'inférence en fonction du format exporté pour différents modèles

L'objectif de ces conversions est de comparer les performances d'inférence (temps d'exécution et nombre d'images par seconde) ainsi que la compatibilité du modèle avec différents moteurs d'inférence embarqués.

6.2 Évaluation des performances en temps réel

La Figure 10 présente une comparaison des performances en temps réel du modèle, exprimées en images par seconde (FPS), pour différents moteurs d'inférence et plateformes matérielles.

Les résultats montrent que l'exécution du modèle sous PyTorch et ONNX offre des performances plus limitées sur les plateformes embarquées, en particulier sur le Raspberry Pi 4. En revanche, l'utilisation de moteurs d'inférence optimisés permet une amélioration significative du débit de traitement.

Le moteur *NCNN* se distingue comme la solution la plus performante sur l'ensemble des plateformes testées, atteignant les valeurs de FPS les plus élevées sur Raspberry Pi 4, Raspberry Pi 5 et PC. Le moteur *OpenVINO INT8* offre également de bonnes performances, notamment sur Raspberry Pi 5 et sur PC, grâce aux optimisations liées à la quantification.

L'augmentation des performances observée entre Raspberry Pi 4 et Raspberry Pi 5 pour tous les moteurs d'inférence confirme l'impact direct des capacités matérielles sur l'exécution temps réel. Enfin, la plateforme PC permet d'atteindre les performances maximales et sert de référence pour l'évaluation globale du modèle.

Ces résultats soulignent l'importance du choix du moteur d'inférence et des optimisations appliquées afin de garantir une exécution temps réel efficace sur des systèmes embarqués à ressources limitées.

L'objectif du projet, fixé à une exécution en temps réel supérieure à 10 FPS, est ainsi atteint et dépassé avec le format NCNN, notamment sur Raspberry Pi 5 et sur PC. Ces résultats montrent que NCNN est le format le plus adapté pour le déploiement embarqué du modèle, grâce à ses optimisations spécifiques pour les architectures ARM, permettant un gain significatif en performances par rapport aux formats TensorFlow Lite, PyTorch, ONNX et OpenVINO INT8.

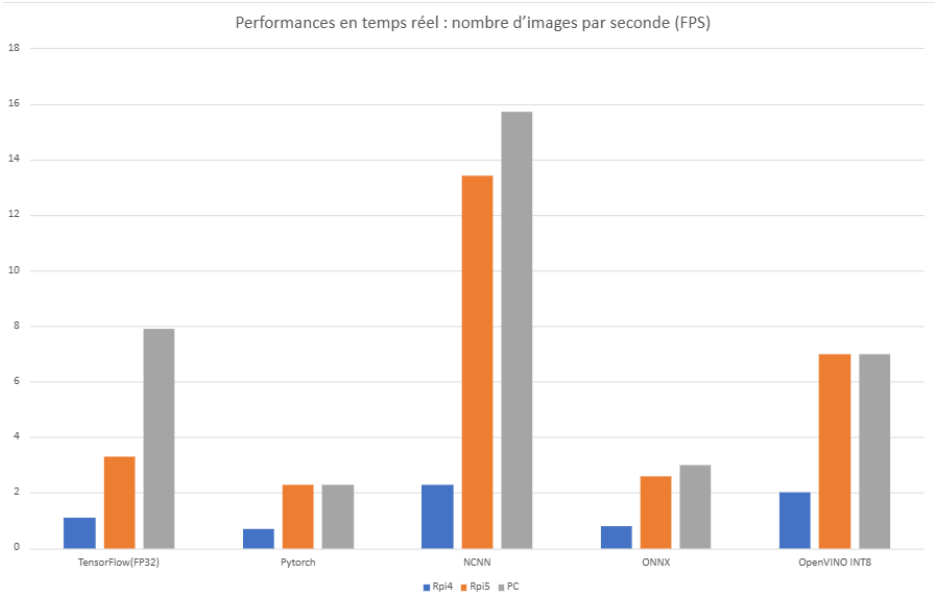

Figure 10: Performances en temps réel : nombre d'images par seconde (FPS)

7 Limites et perspectives

7.1 Limites du projet

Malgré les excellentes performances globales de notre modèle, il est important de souligner certaines limites critiques pour une utilisation dans un scénario réel de conduite autonome. L'analyse de la matrice de confusion non normalisée nous permet de mettre en évidence des points de vigilance majeurs.

L'un des risques principaux réside dans le fait que le système n'est pas totalement sécurisé. Comme nous pouvons le voir sur la matrice, le modèle fait parfois des erreurs de confusion entre différentes classes.

Par exemple :

- Il arrive que certains panneaux de limitation de vitesse soient confondus entre eux (comme les limitations à 50, 60 or 70 km/h), ce qui pourrait entraîner des décisions dangereuses pour un véhicule automatisé.
- Nous observons également quelques fausses détections où des éléments du décor (le "background") sont interprétés comme des panneaux, ou inversement, des panneaux réels qui ne sont pas détectés.

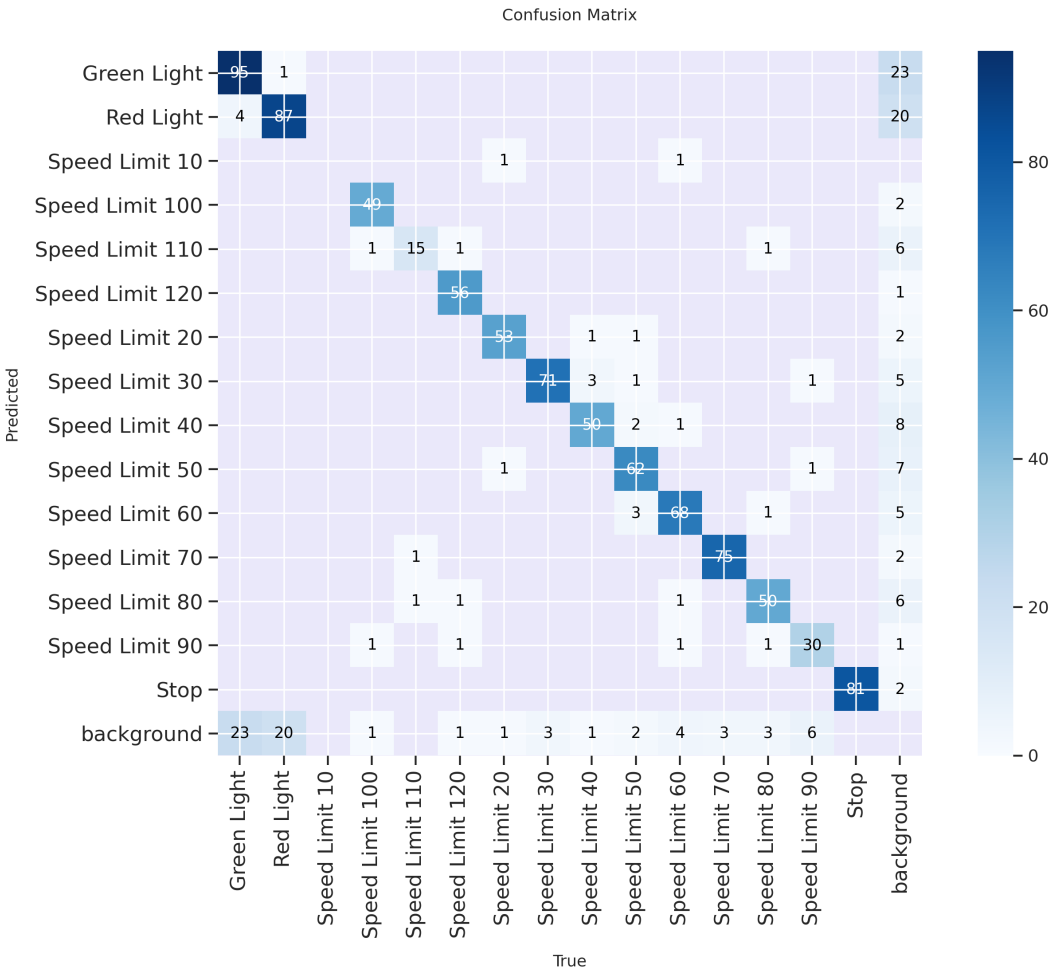

Figure 11: Matrice de confusion sur le jeu de validation

En conclusion, s'appuyer uniquement sur un tel système reste risqué. Il s'agit d'un système non déterministe basé sur des probabilités : une simple variation de lumière ou un angle de vue différent peut modifier le résultat. Pour une application critique, ce modèle devrait être couplé à d'autres capteurs ou à des algorithmes de vérification supplémentaires pour garantir une sécurité maximale.

7.2 Perspectives d'amélioration

Pour aller plus loin dans ce projet et corriger les limites que nous avons identifiées, plusieurs pistes d'amélioration peuvent être envisagées :

- **Augmentation de la base de données :** Pour réduire les erreurs de confusion entre les panneaux, nous pourrions utiliser des techniques d'augmentation de données (Data Augmentation). En ajoutant des images avec des conditions météo difficiles, des flous de mouvement ou des éclairages variés, le modèle deviendrait beaucoup plus robuste et fiable face aux imprévus de la route.
- **Évolution de la plateforme matérielle :** Bien que la Raspberry Pi 5 soit performante, l'utilisation de cartes spécialisées comme la **NVIDIA Jetson** serait un vrai plus. Ces cartes possèdent des processeurs

graphiques (GPU) pensés pour l'IA, ce qui permettrait de faire tourner des modèles plus lourds et plus précis sans sacrifier la vitesse.

- **Utilisation d'accélérateurs matériels :** Sur nos Raspberry Pi actuelles, nous pourrions ajouter des modules comme la **Coral USB Accelerator** (Google). Ce type de composant permet de décharger le processeur principal et d'améliorer considérablement le nombre d'images par seconde (FPS), rendant la détection encore plus fluide pour du temps réel.

8 Conclusion

Pour conclure, ce projet nous a permis de traverser toutes les étapes de la création d'une application d'intelligence artificielle embarquée, de l'entraînement du modèle jusqu'à son déploiement réel sur le terrain.

Nous avons réussi à démontrer qu'il est tout à fait possible de faire de la détection de panneaux de signalisation en temps réel sur des plateformes à ressources limitées comme la Raspberry Pi 5. Grâce à l'utilisation du modèle YOLOv8n et à l'optimisation au format NCNN, nous avons atteint, et même dépassé, notre objectif initial de 10 images par seconde (FPS).

Cependant, ce travail nous a aussi montré que l'IA reste un domaine complexe où la sécurité est primordiale. L'analyse de notre matrice de confusion nous rappelle qu'un système basé sur des probabilités comporte toujours des risques d'erreur. C'est pourquoi, pour une utilisation réelle dans un véhicule autonome, notre système devrait être vu comme une base solide qu'il faudrait compléter avec d'autres capteurs et une base de données encore plus riche.

En résumé, ce projet a été pour nous une expérience très formatrice qui nous a permis de mieux comprendre le compromis permanent entre la puissance de calcul, la vitesse d'exécution et la précision du modèle dans le monde de l'Edge AI.

Annexe A : Manuel d'installation et de configuration

8.1 Installation des bibliothèques python

Depuis l'environnement virtuel `yolo_env`, lancer l'installation :

```
(yolo_env) rpi:~/PATH $ pip install --upgrade pip
(yolo_env) rpi:~/PATH $ pip install -r requirements.txt
```

⏳ Cette étape peut prendre environ **10 minutes**, selon la connexion et le Raspberry Pi utilisé.

8.2 Vérification de l'installation

Vérifier que YOLO est correctement installé : ⚠️ Lors du premier chargement, ne pas interrompre l'exécution (Ctrl+C), certaines initialisations peuvent prendre quelques instants.

```
(yolo_env) rpi:~/PATH $ python3
>>> from ultralytics import YOLO
>>>
```

Si aucune erreur n'apparaît, l'installation de YOLO est terminée avec succès.

L'installation est désormais terminée. Toutefois, cette étape peut introduire un problème de compatibilité. La section suivante explique l'erreur rencontrée et la méthode pour la résoudre.

8.3 Test caméra après installation (`rpi-cam-hello`)

Relancer un test de la caméra :

```
(yolo_env) rpi:~/PATH $  rpi-cam-hello
```

⚠️ Si une erreur apparaît à cette étape, cela indique généralement un conflit de dépendances Python (ex. NumPy), traité dans la section suivante.

Explication du problème

- La bibliothèque **rpicam** (caméra Raspberry Pi) dépend de **NumPy 1.24.2**
- Le package **Ultralytics** installe automatiquement une version plus récente de NumPy (**NumPy 2.1.3**)
- Cette mise à jour **casse la compatibilité avec rpicam**
- Résultat :

Figure 12: Extrait du manuel d'installation

La documentation complète, incluant le manuel d'installation, est disponible sur le dépôt GitHub : Installation Manual.

Annexe B : Les paramètres d'entraînement

Argument	Type	Default	Description
`project`	`str`	`None`	Name of the project directory where training outputs are saved. Allows for organized storage of different experiments.
`optimizer`	`str`	`'auto'`	Choice of optimizer for training. Options include `SGD`, `Adam`, `AdamW`, `NAdam`, `RAdam`, `RMSProp` etc., or `auto` for automatic selection based on model configuration. Affects convergence speed and stability.
`name`	`str`	`None`	Name of the training run. Used for creating a subdirectory within the project folder, where training logs and outputs are stored.
`model`	`str`	`None`	Specifies the model file for training. Accepts a path to either a `.pt` pretrained model or a `.yaml` configuration file. Essential for defining the model structure or initializing weights.
`data`	`str`	`None`	Path to the dataset configuration file (e.g., `coco8.yaml`). This file contains dataset-specific parameters, including paths to training and validation data, class names, and number of classes.
`classes`	`list[int]`	`None`	Specifies a list of class IDs to train on. Useful for filtering out and focusing only on certain classes during training.
`device`	`int` or `str` or `list`	`None`	Specifies the computational device(s) for training: a single GPU (`device=0`), multiple GPUs (`device=[0,1]`), CPU (`device=cpu`), MPS for Apple silicon (`device=mps`), or auto-selection of most idle GPU (`device=-1`) or multiple idle GPUs (`device=[-1,-1]`)
`freeze`	`int` or `list`	`None`	Freezes the first N layers of the model or specified layers by index, reducing the number of trainable parameters. Useful for fine-tuning or transfer learning.
`batch`	`int` or `float`	`16`	Batch size, with three modes: set as an integer (e.g., `batch=16`), auto mode for 60% GPU memory utilization (`batch=-1`), or auto mode with specified utilization fraction (`batch=0.70`).

Figure 13: Extrait des principaux paramètres d'entraînement du modèle YOLO