

Rapport d'étude

**« Entraînement d'un réseau de neurone YOLO
pour la détection de petits objets 3D
dans des images »**

Exploitation sur carte Raspberry Pi 4

Version 2.0

Jean-Luc CHARLES
Consultant IA/Data processing

version 2.0 du 17 décembre 2024

HISTORIQUE DES MODIFICATIONS

Édition	Révision	Date	Modification	Visa
1	0	2024-12-02	Version initiale	
2	0	2024-12-17	Version caméra grand angle sur Thymio + 200 images	

Table des matières

1 Contexte de l'étude.....	5
1.1 Objectifs de l'étude.....	5
Étude préliminaire.....	5
Étude consolidée.....	6
2 Préparation du jeu de données.....	7
2.1 Création des images.....	7
2.2 Annotation des images sur le site Roboflow.....	8
3 Entraînement du réseau de neurones YOLO.....	10
3.1 Choix des versions du réseau de neurones.....	11
3.2 Création de l'Environnement Virtuel Python (EVP).....	11
3.3 Choix des méta-paramètres d'entraînement.....	12
4 Entraînement des réseaux YOLO, résultats.....	13
4.1 Environnement de calcul.....	13
4.2 Résultats des entraînements.....	15
Temps d'inférence.....	15
Précision des réseaux entraînés.....	16
4.3 Conclusion.....	17
5 Exploitation sur RPi4 des réseaux YOLO entraînés.....	18
5.1 Préparation de la carte SD pour la RPi4.....	18
5.2 Configuration de la carte RPi4.....	18
Création de l'environnement virtuel Python vision.....	18
Installation des modules Python dans l'EVP ucia activé.....	19
5.3 Exploitation du réseau YOLO sur RPi4.....	19
Programme de détection des objets.....	19
Fichiers de poids du réseau entraîné au format NCNN.....	21
Programme de détection des objets et des couleurs.....	21
6 Conclusions.....	23
7 Glossaire.....	24
8 Annexes.....	25
8.1 Prise d'images avec la caméra de RPi4.....	25
take_image.py.....	25
8.2 Programmes Python d'entraînement.....	26
train_YOLOv8.py.....	26
train_YOLOv11.py.....	27
eval_YOLOv8n.py.....	28
eval_YOLOv11.py.....	29
process_results.py.....	30
8.3 Fichiers résultats.....	31
results_yolov8n_V2.txt :.....	31
results_yolov8s_V2.txt :.....	32
results_yolo11n_V2.txt.....	33
results_yolo11s_V2.txt.....	34
9 Références.....	35

Index des figures

Figure 1: Dispositif de prise d'images de l'étude V1.0.....	5
Figure 2: Dispositif de prise d'image version V2.0.....	6
Figure 3: Création des images d'entraînement de l'étude V2.0.....	7
Figure 4: Exemples d'images d'entraînement.....	7
Figure 5: L'interface web du site Roboflow pour annoter les images.....	8
Figure 6: Le jeu de données créé sur Roboflow.....	9
Figure 7: Le réseau de neurones YOLO sur le site web Ultralytics	10
Figure 8: Différentes versions du réseau YOLO11.....	11
Figure 9: Arborescence du projet.....	13
Figure 10: Exemple d'image de validation du réseau YOLO entraîné.....	14
Figure 11: Statistiques d'entraînement du réseau YOLO.....	15
Figure 12: Flash de la carte micro SD pour la PRi4.....	18
Figure 13: Terminal de lancement du programme inf_camera-1.py.....	20
Figure 14: Fenêtre graphique d'affichage des objets détectés.....	20
Figure 15: Message du module ultralytics pour la première utilisation du format ncnn.....	21
Figure 16: Terminal de lancement du programme inf_camera-2.py.....	22
Figure 17: Fenêtre graphique d'affichage des objets détectés.....	22

Index des tableaux

Tableau 1: Tableau des plages de valeurs des méta-paramètres d'entraînement.....	12
Tableau 2: Tableau des paramètres d'entraînement.....	12
Tableau 3: Temps moyen d'inférence des réseaux yolov8 et yolo11 (PC entraînement).....	15
Tableau 4: Évaluation des entraînements du réseau yolov8n.....	31
Tableau 5: Évaluation des entraînement du réseau yolov8s.....	32
Tableau 6: Évaluation des entraînement du réseau yolov11n.....	33
Tableau 7: Évaluation des entraînement du réseau yolo11s.....	34

1 Contexte de l'étude

Depuis janvier 2023 l'association « la ligue de l'enseignement » coordonne le projet UCIA (Usages et Consciences Des Intelligences Artificielles). Dans le cadre de ce projet, un kit pédagogique doit être créé incluant notamment un robot IA Open Source et Open Hardware dont l'utilisation doit permettre d'encourager un regard critique sur l'Intelligence Artificielle.

Le document [UCIA_Cahier des charges 11-2024.pdf](#) précise le fonctionnement attendu du robot support des fonctionnalités IA. Trois niveaux sont décrits dans le cahier des charges UCIA : dans le cadre de cette étude Il s'agit de consolider le développement des composants logiciels du **jalon Niveau 0 pour le mode chasseur trésor** (page 8 à 11).

1.1 Objectifs de l'étude

Compte tenu des besoins fonctionnels exprimés dans le cahier des charges UCIA pour le « jalon Niveau 0 pour le mode chasseur trésor » mentionné ci-dessus, les objectifs de l'étude sont :

1. Consolider la procédure d'entraînement d'un réseau de la famille YOLO, avec une banque de données d'environ 200 images prises avec la caméra Raspberry « grand angle ».
2. Exploiter sur la carte RPi4 munie d'une caméra RPi2, la meilleure version du réseau de neurones de la famille YOLO entraîné à détecter trois types d'objets (sphère, cube, étoile) situés en face du robot portant la caméra, dans un rayon de 5 à 45 cm.
3. Développer les outils et procédures permettant l'exploitation à distance du réseau YOLO, avec un ordinateur connecté en WiFi avec la carte RPi4.

1 Étude préliminaire

La version préliminaire de l'étude, utilise la caméra standard « Raspberry Camera V2 » posée sur un pied, avec laquelle 62 images ont été prises pour l'entraînement des réseaux de neurones. La figure 1 illustre le dispositif de prise d'images.



Figure 1: Dispositif de prise d'images de l'étude V1.0.

Le document [UCIA-IA-DetectionOBJ_V1.1.pdf](#) décrit la méthodologie et les résultats de l'étude préliminaire.

2 Étude consolidée

Pour l'étude consolidée, les images des objets 3D sont réalisées avec le même environnement que celui qui a été utilisé à l'exploitation :

- caméra Raspberry « Grand angle »,
- carte RPi4 disposée sur le robot Thymio grâce au support plexiglas spécialement conçu dans le cadre du projet UCIA.

La figure 2 illustre le dispositif utilisé pour l'étude V2.0.

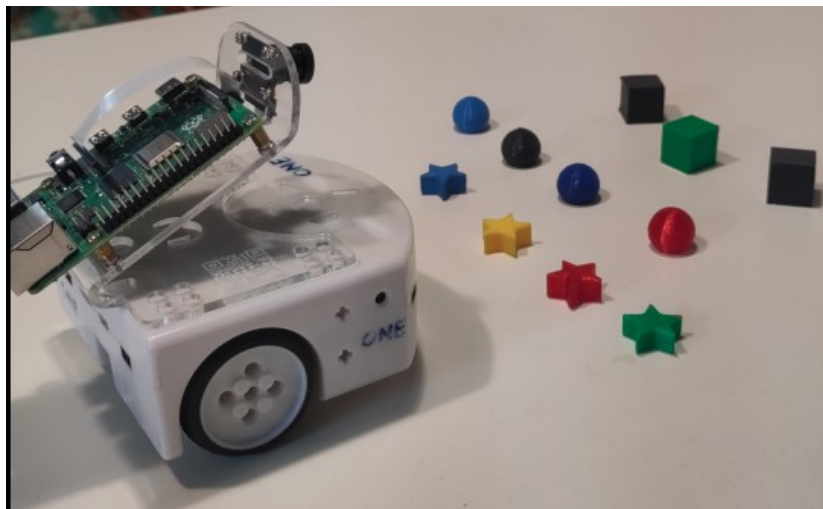


Figure 2: Dispositif de prise d'image version V2.0.

Le présent document [UCIA-IA-DetectionOBJ_V2.0.pdf](#) décrit la méthodologie et les résultats de l'étude consolidée V2.0.

2 Préparation du jeu de données

2.1 Création des images

Le programme Python `take_image.py` (code source en annexe et dans les livrables RPi4) développé pour l'étude permet de numéroter automatiquement au format `objets3D-nnn.jpg` les images des objets prises par la caméra. On regroupe 12 objets dans chaque image (4 étoiles, 4 balles et 4 cubes) pour obtenir un nombre d'objets suffisant : avec 12 objets par images, les 200 images de l'étude V2.0 donnent environ 2400 objets, ce qui constitue une bonne base d'entraînement :



Figure 3: Création des images d'entraînement de l'étude V2.0.

Les exemples de la figure 4 illustrent le type d'images réalisées avec la caméra « grand angle » :

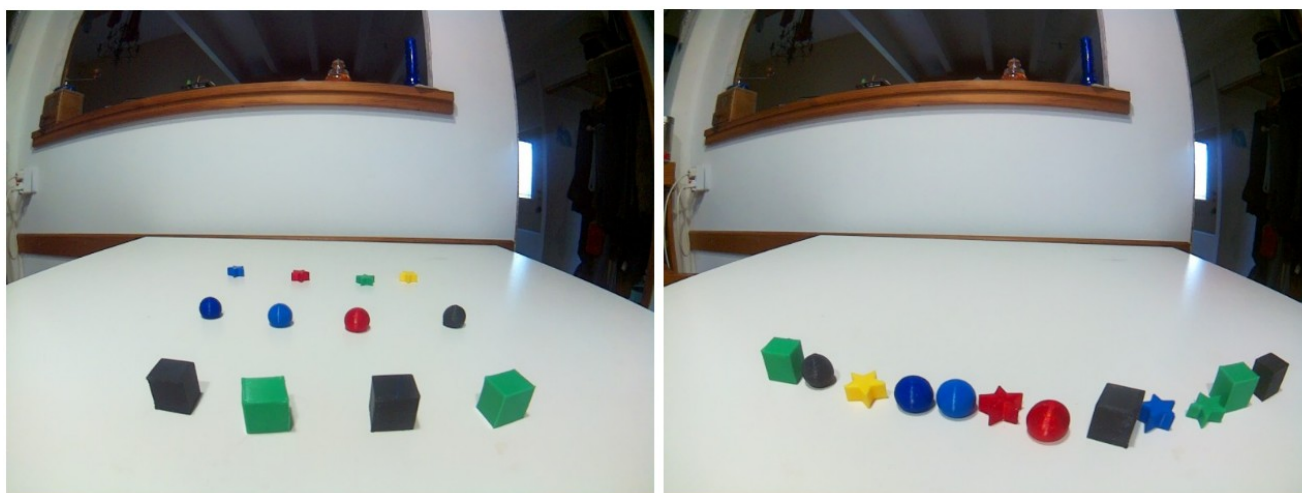


Figure 4: Exemples d'images d'entraînement

Au total nous avons réalisé 200 images différentes avec les 12 objets disponibles. Les images sont ensuite téléchargées sur le site Roboflow pour réaliser l'annotation manuelle.

2.2 Annotation des images sur le site Roboflow

Le site Roboflow propose une interface facile à utiliser dans un navigateur web pour réaliser la tâche d'annotation des images (voir figure 5).

Les images chargées sur le site sont ensuite annotées à la main une par une. Pour chacun des 12 objets contenus dans chaque image, il faut :

1. Délimiter précisément avec la souris la boîte englobante de l'objet (*bounding box*).
2. Labelliser l'objet délimité en utilisant une des classes : *star*, *cube*, *ball*.

Le travail d'annotation est une étape très importante pour la qualité de l'entraînement du réseau de neurones. Une fois tous les objets d'une image entourés et labellisés, on peut passer à l'image suivante.

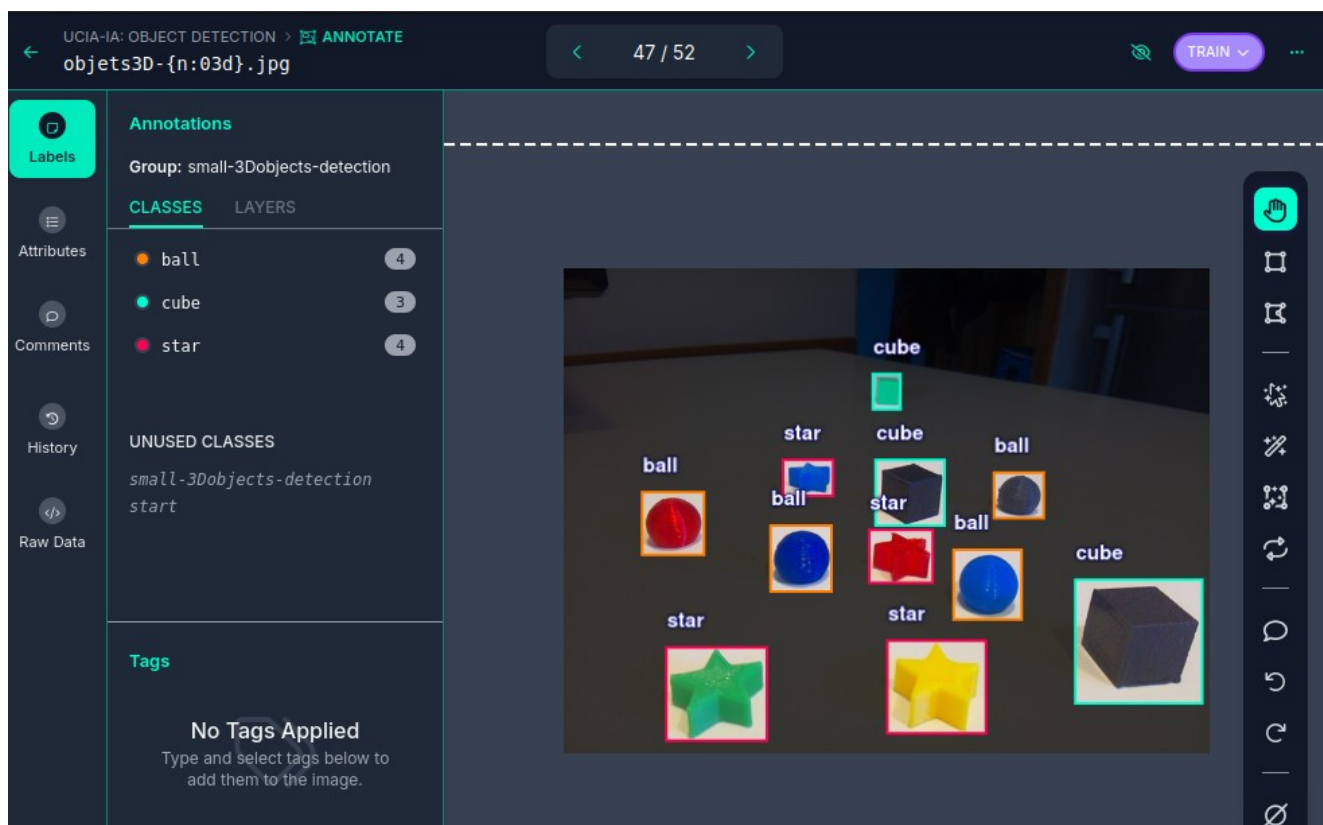


Figure 5: L'interface web du site Roboflow pour annoter les images.

Quand toutes images sont annotées, on peut créer sur le site Roboflow des jeux de données (*datasets*), en répartissant les 200 images annotées en plusieurs jeux :

- **Train set** : le jeu d'images pour l'entraînement du réseau de neurones.
- **Validation set** : le jeu d'images pour évaluer les performances du réseau de neurones à différentes étapes de l'entraînement. Ces images ne sont jamais apprises par le réseau de neurones !
- **Test set** : un jeu d'images qu'on peut former pour mesurer les performances du réseau entraîné, indépendamment des jeux d'entraînement et de validation. Dans le cadre de l'étude on choisit de privilégier les jeux d'entraînement et de validation : des images de test supplémentaires seront créées lors de la mesure sur carte RPi4 des performances du réseau entraîné.

Avec les 200 images créées, on choisit de mettre :

- 163 images pour l'entraînement, soit un total d'environ 2000 objets cube, étoile et balle,
 - 38 images (~456 objets) pour les validations utilisées pendant l'entraînement du réseau.
- Les images du jeu de données sont converties en images 640 x 640 pixels en ton de gris.

La figure 6 montre le jeu de données créés sur le site Roboflow.

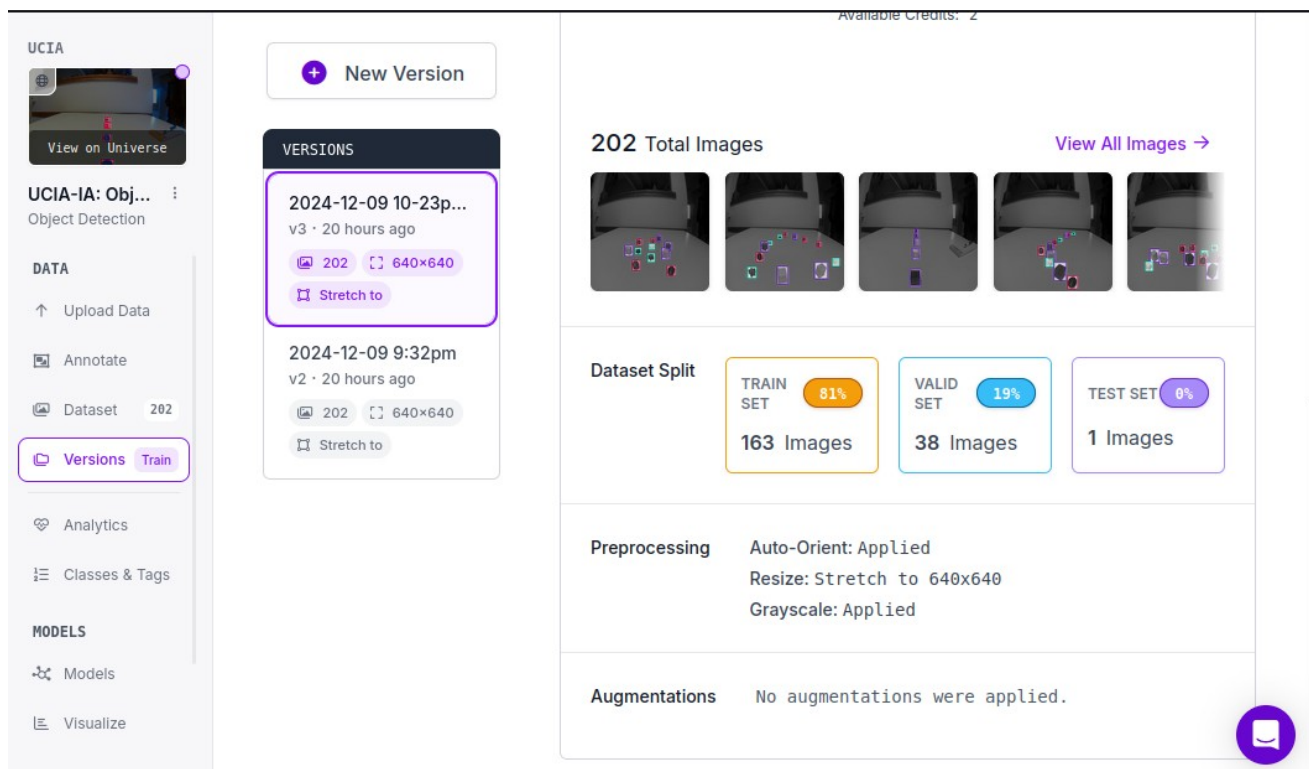


Figure 6: Le jeu de données créé sur Roboflow.

Une fois annoté le jeu de données sur le site Roboflow, il est téléchargé sur le PC d'entraînement aux formats **yolov8** et **yolo11** pour réaliser l'entraînement des réseaux de neurones correspondants.

3 Entraînement du réseau de neurones YOLO

Le réseau de neurones YOLO (*You Only Look Once*) est un modèle populaire de détection d'objets et de segmentation d'images. Il a été développé par Joseph Redmon et Ali Farhadi à l'Université de Washington. Lancé en 2015, YOLO a rapidement gagné en popularité grâce à sa rapidité et à sa précision. La version YOLO8 est couramment adoptée comme version optimale de YOLO. La version actuelle est YOLO11.

Les différentes versions du réseau YOLO sont gérées sur le site WEB Ultralytics :

<https://docs.ultralytics.com/fr>

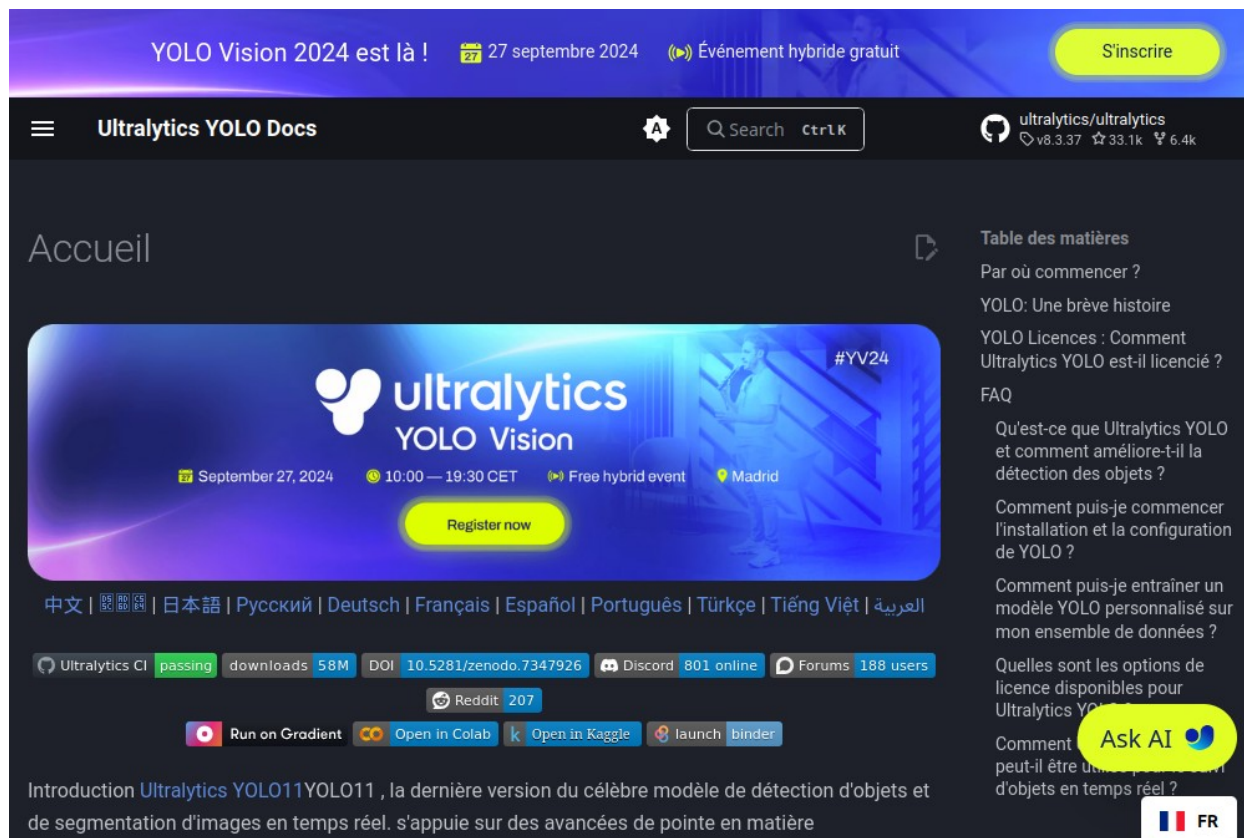


Figure 7: Le réseau de neurones YOLO sur le site web Ultralytics .

Extraits de la page d'accueil du site Ultralytics :

Qu'est-ce que Ultralytics YOLO et comment améliore-t-il la détection des objets ?

Ultralytics YOLO est la dernière avancée de la célèbre série YOLO (You Only Look Once) pour la détection d'objets et la segmentation d'images en temps réel. Elle s'appuie sur les versions précédentes en introduisant de nouvelles fonctionnalités et des améliorations pour accroître les performances, la flexibilité et l'efficacité. YOLO prend en charge diverses tâches d'IA visionnaire telles que la détection, la segmentation, l'estimation de la pose, le suivi et la classification. Son architecture de pointe garantit une vitesse et une précision supérieures, ce qui la rend adaptée à diverses applications, y compris les appareils périphériques et les API en nuage.

3.1 Choix des versions du réseau de neurones

Les versions 8 et 11 du réseau YOLO sont proposées en 4 architectures de complexité croissante :

- YOLO...n → nano pour les tâches petites et légères.
- YOLO...s → small mise à niveau de nano, meilleure précision.
- YOLO...m → medium pour une utilisation à usage général.
- YOLO...l → large meilleure précision au prix d'un calcul plus lourd.
- YOLO...x → Extra-large pour une précision et des performances maximales.

La figure 8¹ détaille quelques caractéristiques des 5 architectures du réseau YOLO11(Source : <https://learnopencv.com/yolo11/>) :

Model	size (pixels)	mAP ^{val} 50-95	Speed CPU ONNX (ms)	Speed T4 TensorRT10 (ms)	params (M)	FLOPs (B)
YOLO11n	640	39.5	56.1 ± 0.8	1.5 ± 0.0	2.6	6.5
YOLO11s	640	47.0	90.0 ± 1.2	2.5 ± 0.0	9.4	21.5
YOLO11m	640	51.5	183.2 ± 2.0	4.7 ± 0.1	20.1	68.0
YOLO11l	640	53.4	238.6 ± 1.4	6.2 ± 0.1	25.3	86.9
YOLO11x	640	54.7	462.8 ± 6.7	11.3 ± 0.2	56.9	194.9

Figure 8: Différentes versions du réseau YOLO11.

Vu la simplicité des objets à détecter, on choisit de se limiter aux architecture **n** et **s** susceptibles de fournir un bon score en un temps raisonnable sur une RPi4. Les architectures plus complexes **m**, **l** et **x** risquent de pénaliser le temps d'inférence. Au final les réseaux candidats retenus pour l'étude sont : **yolov8n**, **yolov8s**, **yolo11n** et **yolo11s**.

3.2 Création de l'Environnement Virtuel Python (EVP)

L'état de l'art pour entraîner un réseau de neurones avec les modules Python consiste à créer un **Environnement Virtuel Python** (EVP), au sein duquel les modules Python sont chargés et les programmes d'entraînement sont développés et exploités.

L'EVP **.vision** est créé avec le module Python **venv** sur le PC d'entraînement :

```
python3 -m venv .vision
```

L'EVP **.vision** est ensuite activé, puis les modules Python nécessaires à l'entraînement des réseaux sont ajoutés dans l'EVP :

```
source .vision activate
pip install ultralytics, onnx, onnxruntime, pathlib
```

Le module Python **ultralytics** utilise le mécanisme de dépendance pour charger à son tour tous les autres modules Python nécessaires au *machine learning* (**tensorflow**, **torch**, **numpy**, **scipy**, **matplotlib**...).

¹ Source : <https://learnopencv.com/yolo11/>

3.3 Choix des méta-paramètres d'entraînement

L'entraînement supervisé d'un réseau de neurones est un processus complexe grandement facilité par l'utilisation de modules Python comme **torch** ou **tensorflow**, installés automatiquement par le module **ultralytics**. De nombreux paramètres influent sur l'entraînement, sa vitesse, la qualité du réseau entraîné obtenu....

La page [modes/train/#resuming-interrupted-trainings](https://ultralytics.com/docs/train/#resuming-interrupted-trainings) du site Ultralytics liste ces paramètres.

Pour les besoins de l'étude V2.0, nous retenons les méta-paramètres et les plages de valeurs présentés sur le tableau 1 :

Tableau 1: Tableau des plages de valeurs des méta-paramètres d'entraînement.

Paramètre	Description	Plage de valeurs
epochs	Nombre de répétitions du processus complet d'entraînement pour converger vers le meilleur état de réseau entraîné	20, 40, 60, 80, 100
batch	Nombre d'images fournies dans un lot d'images d'entraînement	2, 4, 8, 16, 32

Le tableau 2 donne les valeurs des autres paramètres utilisés pour les entraînements :

Tableau 2: Tableau des paramètres d'entraînement.

Paramètre	Description	Valeur
imgz	Taille des image (en pixels)	640
patience	Nombre d'époques à attendre sans amélioration des mesures de validation avant d'arrêter l'entraînement. Permet d'éviter le sur-entraînement en arrêtant le processus lorsque les performances atteignent un plateau.	100
pretrained	Détermine s'il faut utiliser un modèle pré-entraîné.	True
seed	Définit la graine aléatoire pour l'entraînement pour garantir la reproductibilité des résultats d'une exécution à l'autre avec les mêmes configurations.	1234
workers	Nombre de threads de travail pour le chargement des données.	0

Nota : des valeurs du paramètre **workers** autre que 0 conduisent à des anomalies de l'entraînement sur le PC d'entraînement.

4 Entraînement des réseaux YOLO, résultats.

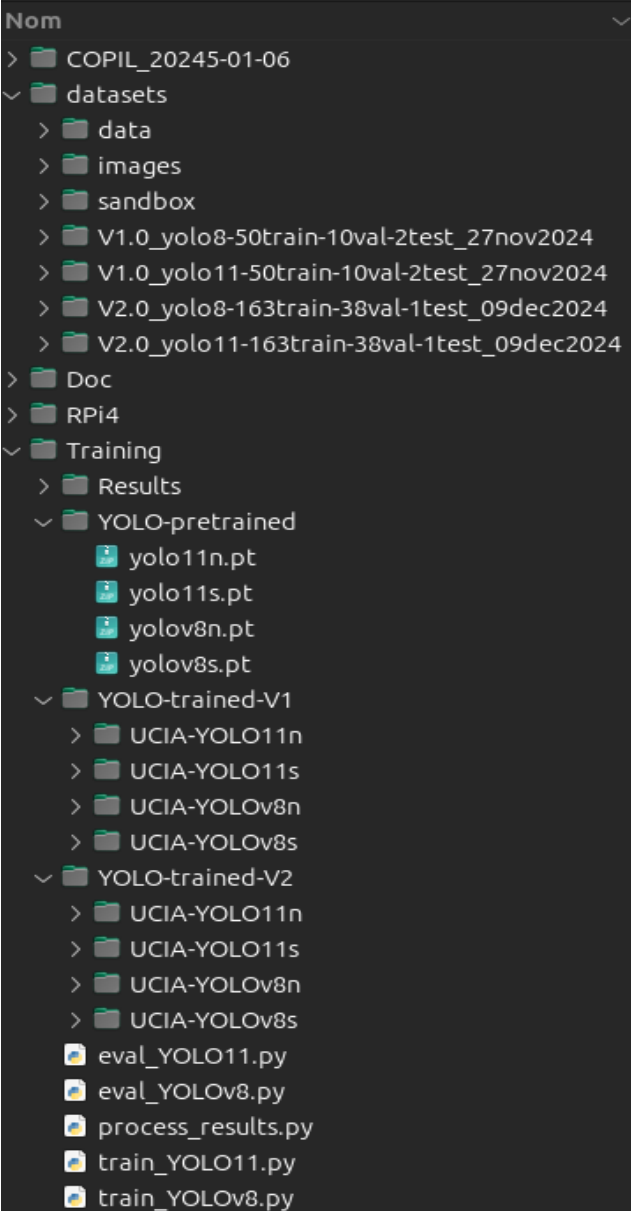
4.1 Environnement de calcul

La combinaison des cas d'entraînement est la suivante :

- 4 réseaux : **yolov8n**, **yolov8s**, **yolov11n** et **yolo11s**,
- 5 valeurs du paramètre **batch** (2, 4, 8, 16 et 32),
- 5 valeur du paramètre **epochs** (20, 40, 60, 80, 100)

ce qui donne $4 \times 5 = 20$ entraînements, répétés $20 + 40 + 60 + 80 + 100 = 300$ fois, soit au total 6000 calculs d'entraînement. Sur un PC portable « core i7 » un entraînement sur 163 images ton de gris prend environ ~10 secondes (fonction de la valeur des méta-paramètres), ce qui donnerait plus de 13 h de calculs pour traiter tous les cas retenus. Nous avons utilisé un PC Ubuntu avec une carte graphique « Nvidia Quadro TRX8000 » pour réduire les temps de calcul à un peu moins de 2 h.

L'arborescence du développement sur le PC de calcul est représentée sur la figure 9 :



```

Nom
> COPI_20245-01-06
> datasets
  > data
  > images
  > sandbox
  > V1.0_yolo8-50train-10val-2test_27nov2024
  > V1.0_yolo11-50train-10val-2test_27nov2024
  > V2.0_yolo8-163train-38val-1test_09dec2024
  > V2.0_yolo11-163train-38val-1test_09dec2024
> Doc
> RPi4
> Training
  > Results
  > YOLO-pretrained
    yolo11n.pt
    yolo11s.pt
    yolov8n.pt
    yolov8s.pt
  > YOLO-trained-V1
    > UCIA-YOLO11n
    > UCIA-YOLO11s
    > UCIA-YOLOv8n
    > UCIA-YOLOv8s
  > YOLO-trained-V2
    > UCIA-YOLO11n
    > UCIA-YOLO11s
    > UCIA-YOLOv8n
    > UCIA-YOLOv8s
  eval_YOLO11.py
  eval_YOLOv8.py
  process_results.py
  train_YOLO11.py
  train_YOLOv8.py

```

- Dossier **datasets** :
contient les jeux d'images annotées, téléchargés depuis le site Roboflow, comme expliqué chapitre 2.2 page 8.
- Dossier **Training** :
contient les fichiers Python pour l'entraînement et le test des réseaux entraînés, ainsi que les principaux sous-dossiers :
 - **YOLO_pretrained** :
contient les fichiers binaires au format **pytorch** des poids des réseaux YOLO pré-entraînés à la détection d'objets avec le jeu d'images MS-COCO.
 - **YOLO-trained_V1** :
contient l'arborescence des poids des réseaux **yolov8vn**, **yolov8vn** et **yolo11s**, **yolo11s** entraînés pour l'étude préliminaire V1.
 - **YOLO-trained_V2** :
contient l'arborescence des poids des réseaux **yolov8vn**, **yolov8vn** et **yolo11s**, **yolo11s** entraînés avec les valeurs des méta-paramètres du tableau 1, pour l'étude V2.

Figure 9: Arborescence du projet.

Le nommage du dossier des résultats pour chacune des combinaisons d'entraînement est :

Training/YOLO-trained-V2/UCIA-YOLOvvv/batch-BB_epo-EEE

vvv : version du réseau YOLO parmi (**v8n**, **v8s**, **11n**, **11s**)

BB : valeur du méta-paramètre **batch**, parmi (**02**, **04**, **08**, **16**, **32**)

EEE : valeur du méta-paramètre **epochs**, parmi (**020**, **040**, **060**, **080**, **100**)

Les entraînements sont réalisés avec les fichiers Python **train_YOLOv8.py** et **train_YOLO11.py** complétés pour l'étude V2 (source donné en annexe et dans les livrables de l'étude).

À la fin de chaque entraînement, les fichiers des poids du réseau entraîné sont écrits dans le dossier **Training/YOLO-trained-V2/UCIA-YOLOvvv/batch-BB_epo-EEE/weights/** :

- **best.pt** : poids du réseau au format binaire du module **pytorch**,
- **best.onnx** : poids du réseau au format ONNX² optimisé pour RPi4,
- **best_ncnn_model/** : dossier des poids du réseau au format NCNN³ optimisé RPi4.

Les tailles des fichiers binaires des poids des réseaux YOLO sont donnés sur le tableau 3 :

Tableau 3: Taille des fichiers binaires des poids des réseaux YOLO utilisés.

	yolov8n	yolov8s	yolo11n	yolo11s
.pt	~6 Mo	~22 Mo	~6 Mo	~19 Mo
.onnx	~12 Mo	~43 Mo	~11 Mo	~37 Mo
.ncnn	~12 Mo	~43 Mo	~11 Mo	~37 Mo

Chacun des dossiers **UCIA-YOLOvvv/batch-BB_epo-EEE** contient également :

- des extraits des images d'entraînement,
- les images de validation à la fin de chaque entraînement :

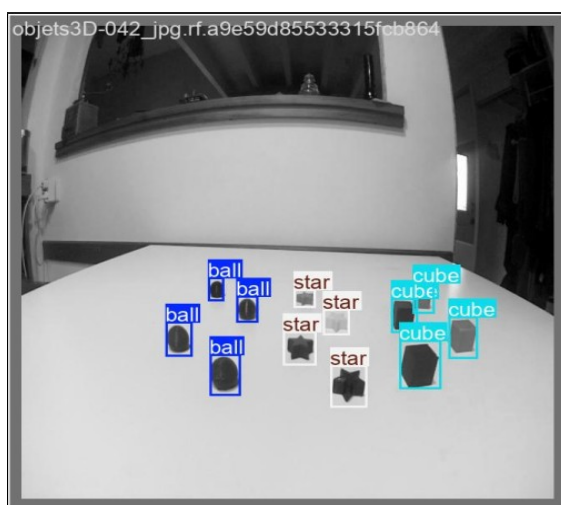


Figure 10: Exemple d'image de validation.

- le tracé de la matrice de confusion,

2 Format ONNX : <https://docs.ultralytics.com/fr/integrations/onnx/>

3 Format NCNN : <https://docs.ultralytics.com/fr/integrations/ncnn/>

- les indicateurs de performance du réseau entraîné : par exemple le fichier **results.png** (figure 11).

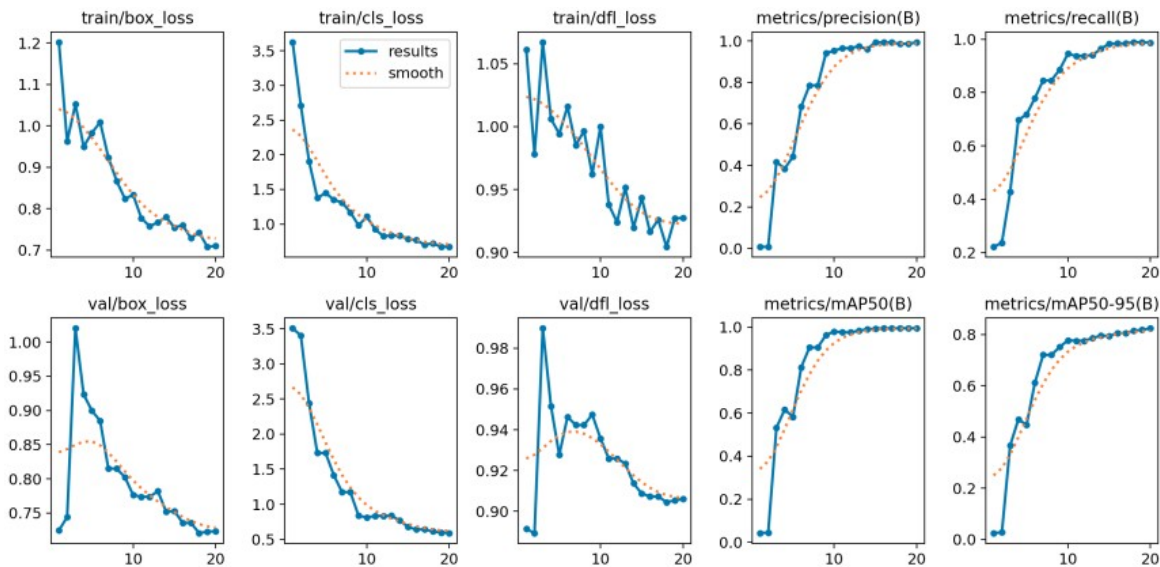


Figure 11: Statistiques d'entraînement du réseau YOLO.

4.2 Résultats des entraînements, comparaison

Parmi toutes les combinaisons d'entraînement des 4 réseaux YOLO et des 5 x 5 valeurs des méta-paramètres **batch** et **epochs** ($4 \times 5 \times 5 = 100$ combinaisons) on cherche à identifier celles qui optimisent le rapport « performances / rapidité de calcul ».

Le module **ultralytics** fournit des fonctions pour évaluer les indicateurs de performance des réseaux de neurones entraînés à la détection d'objets. Les programmes **eval_YOLOv8.py** et **eval_YOLO11.py** (déjà développés pour l'étude V1) calculent ces indicateurs pour les 100 entraînements des réseaux YOLO.

Les indicateurs de performances calculés présents dans ces fichiers sont⁴ :

- prec** (*precision*) : précision des objets détectés, indiquant le nombre de détections correctes.
- recall** (*recall*) : capacité du réseau à identifier toutes les instances d'objets dans les images.
- mAP50** : précision moyenne pour un seuil d'intersection sur union (IoU⁵) de 0,5. Mesure la précision d'un réseau détecteur d'objet pour des détections "faciles".
- mAP50-95** : précision moyenne pour différents seuils IoU allant de 0,50 à 0,95 par pas de 0.5. Mesure la précision d'un réseau détecteur d'objet à différents niveaux de difficulté de détection.
- fitness** : $0.1 * \text{mAP50} + 0.9 * \text{mAP50-95}$

Le détail des résultats est donné pour les 4 modèles de réseau YOLO dans les fichiers **results_yolovvv_V2.txt**, avec **vvv** parmi **v8n**, **v8s**, **11n** et **11s** (cf annexe, pages 33 à 35).

4 Voir article « Analyse approfondie des mesures de performance » <https://docs.ultralytics.com/fr/guides/yolo-performance-metrics/>

5 *Intersection over Union* (IoU) : L'intersection sur l'union donne la mesure du chevauchement entre les boîtes englobantes prédite et vraie. Elle joue un rôle fondamental dans l'évaluation de la précision de la localisation des objets. Voir https://en.wikipedia.org/wiki/Jaccard_index.

1 Temps d'inférence

Le tableau 4 synthétise les temps moyen d'inférence d'entraînement des 4 réseaux testés sur le PC de calcul : on peut noter la différence significatives des temps d'inférences pour l'architecture **s**, deux fois plus grande que pour l'architecture **n**.

Tableau 4: Temps moyen d'inférence des réseaux **yolov8** et **yolo11** (PC de calcul)

Réseau	Temps moyen d'inférence [ms]
yolov8n	2
yolo11n	2
yolov8s	4
yolo11s	4

On constate par ailleurs qu'il n'y a pas de différence significative entre les temps moyens d'inférence **yolov8n** et **yolo11n** d'une part , et **yolov8s** et **yolo11s** d'autre part.

2

3 Précision des réseaux entraînés

On s'attache ici à regarder les colonnes des métriques **recall**, **mAP50-95** et **fitness** dans les 4 fichiers résultats **results_yolovvv_V2.txt**, avec **vvv** parmi **v8n**, **v8s**, **11n** et **11s**, qui sont les plus significatives pour quantifier les performances des réseaux entraînés à la détection d'objets.

Les tendances observées sont les suivantes :

- Les entraînements avec **epoch=20** donnent souvent lieu à des valeurs faibles pour les trois métriques.
- Les entraînements avec **epoch=80** et **epoch=100** donnent souvent des valeurs élevées pour les trois métriques.

Pour synthétiser les résultats nous avons complété le programme Python **process_results.py** pour l'étude V2 (code source en annexe page 32 et dans les livrables) : on lit les 4 fichiers avec la fonction **read_csv** du module **pandas**, puis on trie les **dataframes** obtenus par ordre décroissant des colonnes **recall** et **mAP50-95**, puis par ordre décroissant de la colonne **fitness**.

On affiche à chaque fois les 4 premières lignes qui montrent les meilleures combinaisons d'entraînement.

En triant avec les colonnes **recall** et **mAP50-95**, on obtient :

```
File <Training/Results/results_yolov8n-V2.txt>
#meta-params recall mAP50-95 fitness
batch-02_epo-060 1.0 0.821 0.838
batch-02_epo-080 1.0 0.821 0.838
batch-04_epo-100 1.0 0.821 0.838
batch-04_epo-080 1.0 0.819 0.836

File <Training/Results/results_yolo11n-V2.txt>
#meta-params recall mAP50-95 fitness
batch-08_epo-100 1.0 0.814 0.833
batch-04_epo-080 1.0 0.813 0.831
batch-16_epo-100 1.0 0.813 0.831
batch-04_epo-060 1.0 0.809 0.828

File <Training/Results/results_yolov8s-V2.txt>
#meta-params recall mAP50-95 fitness
batch-02_epo-080 1.0 0.837 0.853
batch-08_epo-060 1.0 0.837 0.852
batch-02_epo-060 1.0 0.835 0.851
batch-02_epo-100 1.0 0.834 0.850

File <Training/Results/results_yolo11s-V2.txt>
#meta-params recall mAP50-95 fitness
batch-04_epo-080 1.0 0.839 0.855
batch-32_epo-060 1.0 0.838 0.854
batch-08_epo-080 1.0 0.837 0.852
batch-02_epo-100 1.0 0.836 0.852
```

En triant avec la colonne **fitness**, on obtient :

```
File <Training/Results/results_yolov8n-V2.txt>
#meta-params recall mAP50-95 fitness
batch-02_epo-100 0.998 0.828 0.844
batch-08_epo-080 0.998 0.821 0.839
batch-08_epo-100 0.998 0.822 0.839
batch-02_epo-080 1.000 0.821 0.838

File <Training/Results/results_yolo11n-V2.txt>
#meta-params recall mAP50-95 fitness
batch-02_epo-100 0.998 0.815 0.833
batch-08_epo-100 1.000 0.814 0.833
batch-02_epo-080 0.998 0.814 0.832
batch-04_epo-100 0.998 0.814 0.832

File <Training/Results/results_yolov8s-V2.txt>
#meta-params recall mAP50-95 fitness
batch-02_epo-080 1.0 0.837 0.853
batch-08_epo-060 1.0 0.837 0.852
batch-02_epo-060 1.0 0.835 0.851
batch-04_epo-060 1.0 0.834 0.850

File <Training/Results/results_yolo11s-V2.txt>
#meta-params recall mAP50-95 fitness
batch-04_epo-080 1.0 0.839 0.855
batch-32_epo-060 1.0 0.838 0.854
batch-04_epo-040 1.0 0.836 0.852
batch-08_epo-060 1.0 0.836 0.852
```

Par comparaison avec les résultats de l'étude préliminaire v1, où les réseaux de neurones étaient entraînés avec les images de la caméra Raspberry Standard, on peut faire les constatations suivantes :

- les valeurs de l'indicateur **recall** (classification des objets) sont pratiquement les mêmes que pour l'étude préliminaire V1 : les réseaux entraînés avec les 163 images de la caméra « Grand Angle » classifient aussi bien les objets que ceux de l'étude V1 (entraînés avec les 52 images de la caméra « Standard »), mais pas mieux.
- les valeurs de l'indicateur **mAP50-95** (positionnement des boîtes englobantes) sont un peu moins bonne pour l'étude V2 que pour l'étude V1, ce qui paraît surprenant ! Une explication possible pourrait être que la caméra Grand Angle de l'étude V2 produit des déformations des objets très proches et un écrasement des objets loins, ce qui peut perturber le positionnement des boîtes englobantes des objets...

Les configurations d'entraînement qui donnent les meilleurs résultats avec les images de validation sont :

- **yolov8n** : **batch-04_epo-100**
- **yolo11n** : **batch-02_epo-100** ou **batch-08_epo-100**
- **yolov8s** : **batch-02_epo-080**
- **yolo11s** : **batch-04_epo-080**

4.3 Conclusion

Au vu des résultats obtenus précédemment, nous pouvons tester sur RPi4 les réseaux **yolov8n** et **yolo11n** (temps d'inférence les plus faibles) dans les configurations les plus performantes :

- **yolov8n** : **batch-04_epo-100**
- **yolo11n** : **batch-02_epo-100** ou **batch-08_epo-100**

On retrouve des tendances connues :

- **batch_size** petit \rightsquigarrow favorise en entraînement de qualité,
- **epochs** élevé \rightsquigarrow compense le petit nombre d'images fournies à chaque entraînement.

Il faut garder présent à l'esprit que ces résultats sont obtenus avec le jeu d'images de validation, et qu'en situation d'exploitation sur RPi4 montée sur le robot mobile, il peut s'avérer que ce soit d'autres combinaisons d'entraînement qui donnent des performances optimales. D'où la nécessité de faire l'évaluation sur la carte RPi4 des réseaux entraînés sur PC de calcul.

5 Exploitation sur RPi4 des réseaux YOLO entraînés

5.1 Préparation de la carte SD pour la RPi4

Il est important d'utiliser une carte SD rapide pour installer le système d'exploitation « Raspberry Pi OS (64bits) » de la RPi4 afin d'optimiser ses performances. Nous avons utilisé une carte micro SD 64 GB de la marque TEAM, de classe A1.

Le système d'exploitation est installé sur la carte micro-SD avec le logiciel « Raspberry Pi imager⁶ » qui propose une interface graphique efficace pour flasher des cartes micro-SD. La version du système d'exploitation installée est datée du 2024-11-19 (cf figure 12).

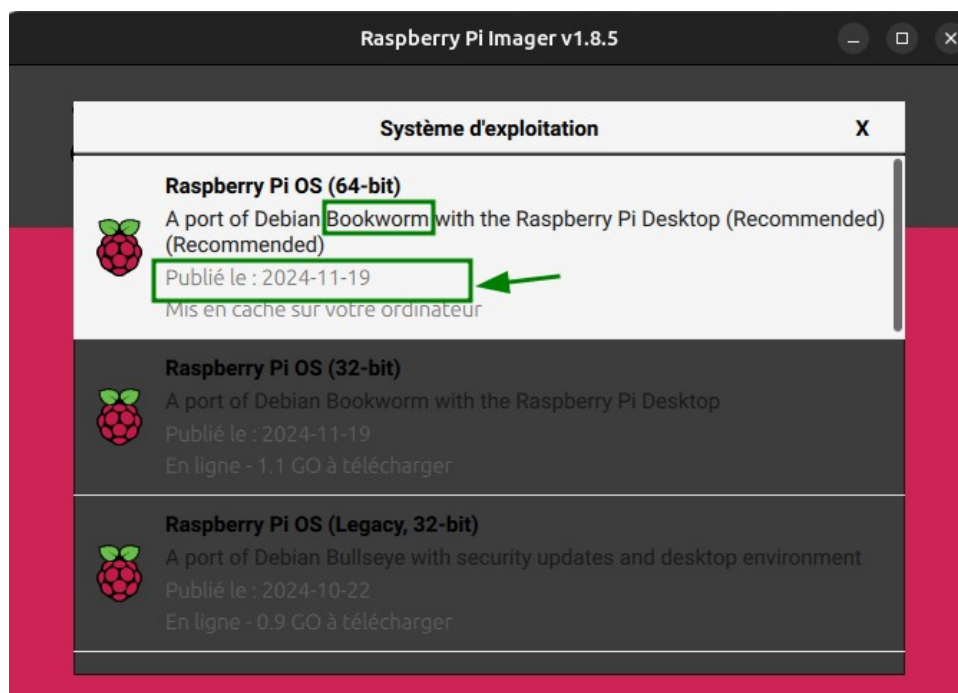


Figure 12: Flash de la carte micro SD pour la RPi4.

5.2 Configuration de la carte RPi4

Au premier démarrage de la RPi4 l'utilisateur **ucia** est créé, avec le mot de passe **poppy!station**, le système d'exploitation est mis à jour, puis le système procède à un redémarrage.

1 Création de l'Environnement Virtuel Python (EVP) **vision**

Après le deuxième démarrage, l'environnement d'exploitation des réseaux entraînés est configuré :

```
ucia@raspberrypi:~ $ mkdir UCIA
ucia@raspberrypi:~ $ cd UCIA
ucia@raspberrypi:~/UCIA $ python -m venv --system-site-packages vision
ucia@raspberrypi:~/UCIA $ source vision/bin/activate
(vision) ucia@raspberrypi:~/UCIA $
```

Le fichier `~/bashrc` est modifié pour activer automatiquement l'EVP **vision** au lancement d'un terminal.

⁶ <https://www.raspberrypi.com/software/>

2 Installation des modules Python dans l'EVP vision activé

```
(vision) ucia@raspberrypi:~/UCIA $ pip install ultralytics
...
(vision) ucia@raspberrypi:~/UCIA $ pip install onnx
...
(vision) ucia@raspberrypi:~/UCIA $ pip install onnxruntime
...
```

L'installation du module **ultralytics** est assez longue car il installe par le biais des dépendances un grand nombre de modules dont certains sont très volumineux (**pytorch**, **tensorflow**...). Au final, le dossier **~/UCIA/vision** prend environ 1.2 GiO.

3 Point d'accès WiFi

Pour la visualisation distante depuis un PC portable des images des réseaux de neurones exploités sur la carte RPi4, il est nécessaire d'utiliser un réseau WiFi.

Afin d'éviter toute difficulté dans l'identification des adresses IP des différents nœuds utilisés sur le WiFi, le plus simple est de configurer la carte RPi4 pour qu'elle émette son propre WiFi, avec une adresse IP déterminée à l'avance, fixe et connue de tous les nœuds du réseau.

Le WiFi généré en utilisant le menu « Réseau » du bureau de la RPi4 s'est avéré inutilisable : avec un débit inférieur à quelques ko/s, il ne permet pas de transmettre suffisamment de données aux autres nœuds connecté au WiFi.

Nous avons donc utilisé un travail déjà mené pour les robots Poppy-Torso et Poppy-Ergo, qui permet de configurer un *hot-spot* (point d'accès WiFi) performant. Le site GitHub du projet <https://github.com/poppy-project/rpi3-hotspot> fournit les détails de l'installation du *hot-spot* sur la carte Rpi.

Les principaux paramètres du *hot-spot* configuré pour la carte micro-SD de la RPi4 sont :

- SSID : **RPi4-UCIA**
- clef : **poppy!station**
- adresse IP de la carte RPi4 : 10.99.99.1
- adresse réseau du WiFi : 10.99.99.0, masque réseau : 255.255.255.0
- Lancement automatique du WiFi au démarrage de la carte RPi4 : oui.

La figure 13 montre la détection du WiFi UCIA sur un PC portable sous Windows11.

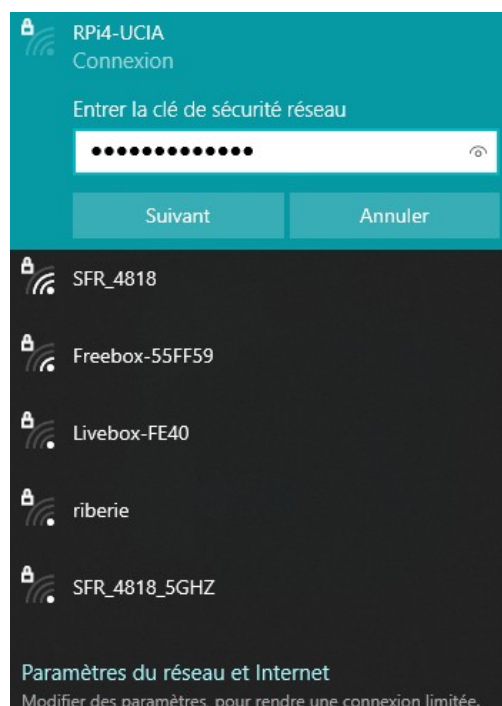


Figure 13: Configuration de l'accès au WiFi RPi4-UCIA

5.3 Exploitation du réseau YOLO sur RPi4

1 Programme de détection des objets

Le programme `inf_camera-1.py`⁷ permet de choisir un réseau `yolo` entraîné et de faire des inférences sur les images du flux vidéo de la caméra de la RPi4. Il affiche en temps réel :

- dans le terminal de lancement : les objets détectés et les temps de *pre-processing*, *inference* et *post-processing* nécessaires au traitement de chaque image par le réseau entraîné,
- dans une fenêtre graphique : le tracé des boîtes englobantes ainsi que du nom de la classe de l'objet et la confiance de détection (cf figure 15).

Le lancement du programme se fait dans le terminal :

```
(vision) ucia@raspberrypi:~ $ cd UCIA/UCIA_ObjectDetection/
(vision) ucia@raspberrypi:~/UCIA/UCIA_ObjectDetection $ python detect_camera-1.py
```

Les figures 14 et 15 illustrent les fenêtres correspondantes.

On peut quitter le programme :

- soit en tapant la touche Q dans la fenêtre graphique,
- soit en tapant la séquence de touches « Ctrl + C » dans le terminal.

```
ucia@raspberrypi: ~/UCIA/UCIA_ObjectDetection
Fichier  Édition  Onglets  Aide
(vision) ucia@raspberrypi:~ $ cd UCIA/UCIA_ObjectDetection/
(vision) ucia@raspberrypi:~/UCIA/UCIA_ObjectDetection $ python inference_camera.py
[0:09:33.974085994] [2081] INFO Camera camera_manager.cpp:325 libcamera v0.3.2+99-1230f78d
[0:09:34.007502367] [2087] WARN RPiSdn sdn.cpp:40 Using legacy SDN tuning - please consider moving SDN inside rpi.denoise
[0:09:34.009854689] [2087] WARN RPi vc4.cpp:393 Mismatch between Unicam and CamHelper for embedded data usage!
[0:09:34.010646574] [2087] INFO RPi vc4.cpp:447 Registered camera /base/soc/i2c0mux/i2c@1/imx219@10 to Unicam device /dev/
media1 and ISP device /dev/media0
[0:09:34.010735722] [2087] INFO RPi pipeline_base.cpp:1120 Using configuration file '/usr/share/libcamera/pipeline/rpi/vc4
/rpi_apps.yaml'
[0:09:34.019330217] [2081] INFO Camera camera.cpp:1197 configuring streams: (0) 800x600-RGB888 (1) 1640x1232-SBGGR10_CSI2P
[0:09:34.022923070] [2087] INFO RPi vc4.cpp:622 Sensor: /base/soc/i2c0mux/i2c@1/imx219@10 - Selected sensor format: 1640x1
232-SBGGR10_1X10 - Selected unicam format: 1640x1232-pBAA
Loading YOLO-trained/UCIA-YOLOv8n/batch-08_epo-080/weights/best_ncnn_model for NCNN inference...

0: 640x640 1 ball, 3 cubes, 2 stars, 494.0ms
Speed: 136.4ms preprocess, 494.0ms inference, 156.0ms postprocess per image at shape (1, 3, 640, 640)

0: 640x640 1 ball, 3 cubes, 2 stars, 704.3ms
Speed: 37.6ms preprocess, 704.3ms inference, 6.6ms postprocess per image at shape (1, 3, 640, 640)

0: 640x640 1 ball, 2 cubes, 2 stars, 523.3ms
Speed: 28.0ms preprocess, 523.3ms inference, 4.0ms postprocess per image at shape (1, 3, 640, 640)

0: 640x640 1 ball, 2 cubes, 2 stars, 442.6ms
```

Figure 14: Terminal de lancement du programme `inf_camera-1.py`.

⁷ Inspiré de <https://docs.ultralytics.com/fr/guides/raspberry-pi/#inference-with-camera>

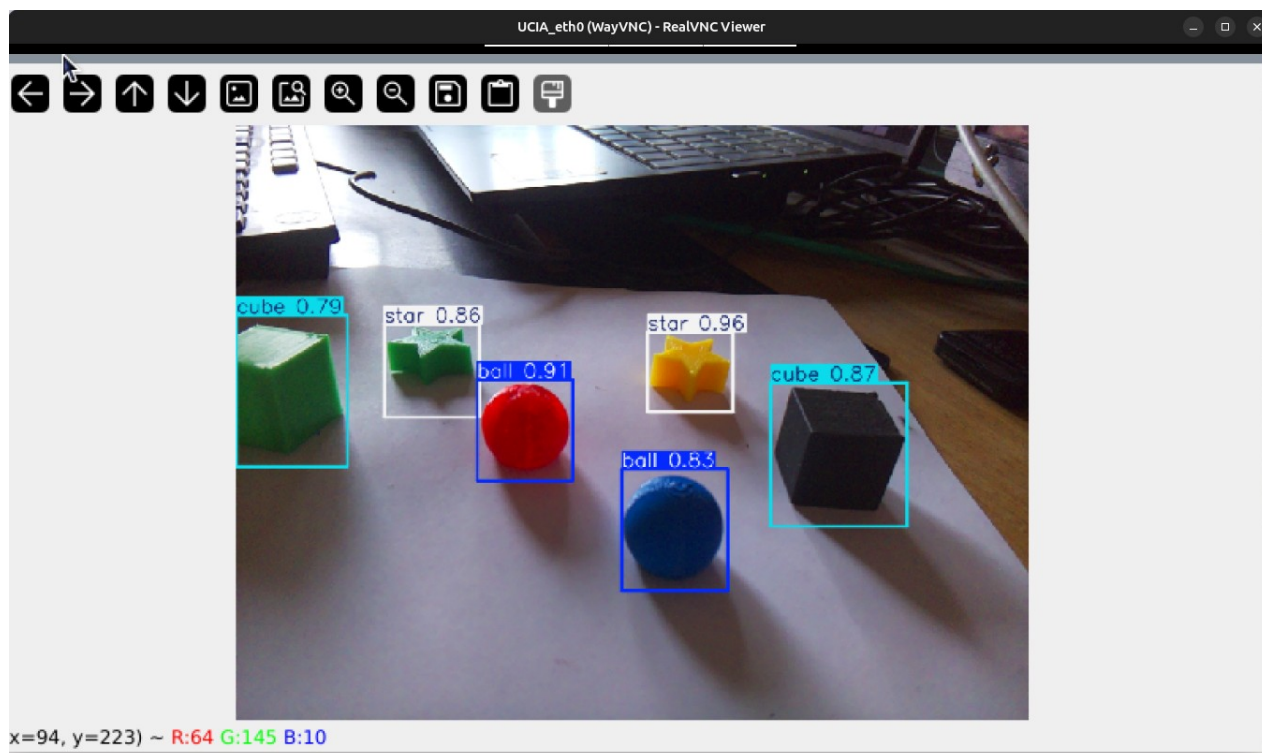


Figure 15: Fenêtre graphique d'affichage des objets détectés.

2 Fichiers de poids du réseau entraîné au format NCNN

Lorsqu'on utilise pour la première fois un fichier au format **NCNN** sur la RPi4, le module **ultralytics** affiche le message de la figure 16:

```
(vision) ucia@raspberrypi:~/UCIA/UCIA_ObjectDetection $ python eval.py
model loaded in 0.8 ms
image pre-processing: 55.0 ms
Loading YOLO-trained/UCIA-YOLOv8n/batch-08_pat-100_epo-080/weights/best_ncnn_model for NCNN inference...
requirements: Ultralytics requirement ['git+https://github.com/Tencent/ncnn.git'] not found, attempting AutoUpdate...
Running command git clone --filter=blob:none --quiet https://github.com/Tencent/ncnn.git /tmp/pip-req-build-km5uft8k
Running command git submodule update --init --recursive -q
```

Figure 16: Message du module ultralytics pour la première utilisation du format ncnn.

En clair, le message est :

```
requirements: Ultralytics requirement
['git+https://github.com/Tencent/ncnn.git'] not found, attempting AutoUpdate...
Running command git clone --filter=blob:none --quiet
https://github.com/Tencent/ncnn.git /tmp/pip-req-build-km5uft8k
Running command git submodule update --init --recursive -q
```

Le chargement et la compilation du module prennent un bon quart d'heure sur RPi4...

3 Programme de détection des objets et des couleurs

On rappelle ici que les images d'entraînement du réseau de neurones sont converties en ton de gris : l'information de couleur est absente, seule la forme est apprise. En suivant un algorithme naïf de traitement des pixels contenus dans la boîte englobante d'un objet, on peut déduire la couleur de l'objet contenu dans la boîte.

Le programme Python **inf_camera-2.py**, spécifiquement développé pour cette étude, permet de choisir un réseau **yolo** entraîné, de faire des inférences sur les images de la caméra de la RPi4. Il affiche en temps réel :

- dans le terminal :
 - le nom de l'objet, la confiance de détection, la couleur déduite des pixels contenus dans la boîte englobante de l'objet,
 - les coordonnées (x1, y2, x2, y1) des coins bas-gauche et haut-droit de la boîte englobante.
- dans une fenêtre graphique : le tracé des boîtes englobantes ainsi que du nom de la classe de l'objet et la confiance de détection (cf figure 18).

Le lancement du programme se fait dans le terminal :

```
(vision) ucia@raspberrypi:~ $ cd UCIA/UCIA_ObjectDetection/
(vision) ucia@raspberrypi:~/UCIA/UCIA_ObjectDetection $ python detect_camera-2.py
```

```

ucia@raspberrypi: ~/UCIA/UCIA_ObjectDetection
Fichier  Édition  Onglets  Aide

cube tensor(0.9130) yellow (105, 599, 251, 429) (141, 176, 95)
balle tensor(0.8989) yellow (427, 426, 504, 342) (190, 103, 77)
balle tensor(0.8901) blue (323, 360, 390, 291) (67, 54, 171)
cube tensor(0.8065) blue (483, 599, 675, 458) (117, 108, 126)

0: 640x640 2 balls, 3 cubes, 1 star, 452.6ms
Speed: 21.4ms preprocess, 452.6ms inference, 5.4ms postprocess per image at shape (1, 3, 640, 640)
cube tensor(0.9439) black (123, 394, 232, 281) (75, 69, 85)
étoile tensor(0.9344) yellow (290, 478, 390, 386) (161, 177, 129)
balle tensor(0.9065) yellow (427, 426, 503, 342) (189, 101, 75)
cube tensor(0.9030) yellow (105, 599, 251, 430) (141, 176, 94)
balle tensor(0.8938) blue (323, 360, 391, 291) (68, 55, 171)
cube tensor(0.8097) blue (483, 599, 675, 458) (117, 108, 126)

0: 640x640 2 balls, 3 cubes, 1 star, 449.9ms
Speed: 24.1ms preprocess, 449.9ms inference, 4.4ms postprocess per image at shape (1, 3, 640, 640)
cube tensor(0.9459) black (122, 393, 232, 281) (75, 69, 84)
étoile tensor(0.9351) yellow (290, 478, 389, 386) (161, 178, 129)
cube tensor(0.9156) yellow (105, 599, 251, 429) (141, 176, 95)
balle tensor(0.9003) yellow (427, 426, 503, 342) (189, 102, 75)
balle tensor(0.8904) blue (323, 361, 390, 291) (68, 55, 171)
cube tensor(0.8566) blue (482, 599, 675, 459) (117, 108, 126)

```

Figure 17: Terminal de lancement du programme `inf_camera-2.py`.

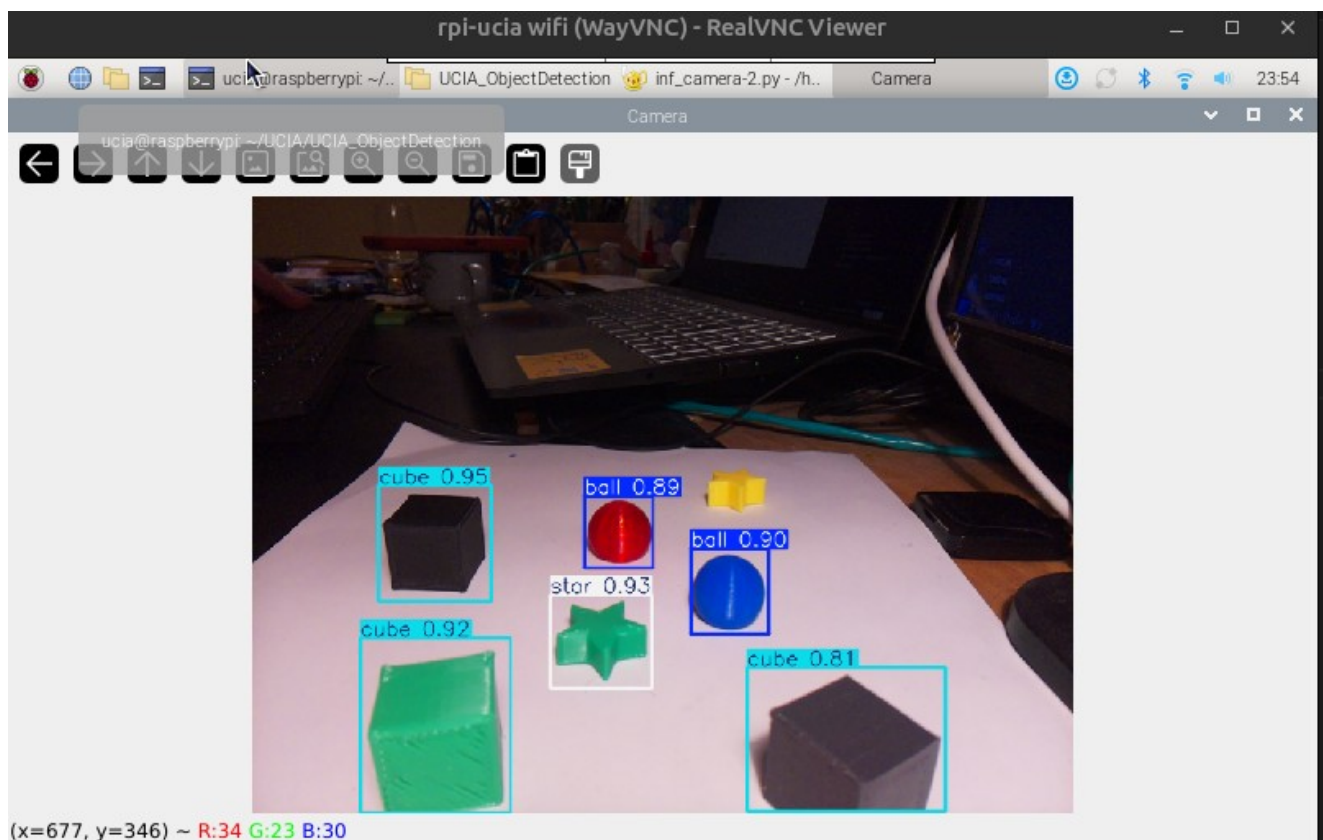


Figure 18: Fenêtre graphique d'affichage des objets détectés.

6 Conclusions

Avec les configurations d'entraînement choisies, on obtient des temps d'inférence sur RPi4 tout à fait convenables :

yolov8n_batch-08_epo-080

format onnx : environ 0.6 s

format ncnn : environ 0.5 s

yolo11n_batch-04_epo-100

format onnx : environ 0.6

format ncnn : environ 0.5 s

Cette étude montre qu'on peut entraîner un réseau YOLO à détecter les petits objets 3D du cahier des charges UCIA et que leur exploitation sur RPi4 donne des temps d'inférence inférieurs à la secondes.

À noter l'importance de placer les objets sur un fond blanc de préférence, avec un bon éclairage.

Pour les performances de détection des objets, il conviendra de faire des essais dans les conditions réelles d'exploitation de la carte RPi4 pour trouver la combinaison d'entraînement qui donne les meilleurs résultats.

Dans l'état actuel de l'étude , la détection des couleurs des objets ne fonctionne de façon robuste que pour les couleurs primaires Rouge, Vert et Bleu.

7 Glossaire

Nom anglais	Nom français	signification
<i>Epoch</i>	Époque	Une itération de l'entraînement du réseau de neurones sur l'ensemble complet des données.
<i>Batch</i>	Lot	Sous -ensemble du jeu complet des données fourni pour l'entraînement du réseau de neurones.. Dans cette étude : un paquet d'images fournies pour un entraînement du réseau de neurones
<i>Batch size</i>	Taille de lot	Méta-paramètre qui fixe la taille du lot fourni au réseau de neurones. Dans notre étude c'est le nombre d'images fournies à chaque entraînement.
<i>Overfitting</i>	Sur-entraînement	C'est un défaut de l'entraînement, où le réseau est sur-entraîné avec les données d'entraînement, et par suite il devient moins performant pour faire des déductions correctes sur de nouvelles images qu'il n'a jamais vues.
<i>Patience</i>	Patience	Nombre d'époques à attendre sans amélioration des mesures de validation avant d'arrêter l'entraînement. Permet d'éviter l' <i>overfitting</i> en arrêtant l'entraînement lorsque les performances atteignent un plateau.
<i>Précision</i>	Precision	Dans le contexte de la détection d'objets, désigne le pourcentage d'objets correctement détecté
<i>Rappel</i>	Recall	La capacité du modèle à identifier toutes les instances d'objets dans les images.

8 Annexes

8.1 Prise d'images avec la caméra de RPi4

1 take_image.py

```
#####  
# Jean-Luc.Charles@mailo.com  
# 2024/11/21 - v1.0  
#####  
  
from picamera2 import Picamera2, Preview  
import sys, time  
  
picam2 = Picamera2()  
picam2.preview_configuration.main.size = (800, 600)  
picam2.configure("preview")  
picam2.start_preview(Preview.QTGL, width=800, height=600)  
picam2.start()  
  
n = 1  
rep = input("numéro image pour démarrer [Q:quit] ? ")  
  
if rep.lower() == 'q':  
    picam2.stop()  
    sys.exit()  
else:  
    n = int(rep)  
  
while True:  
    rep = input(f"ENTER -> image suivante {n:03d} [Q:quit] ...")  
    if rep.lower() == 'q':  
        break  
  
    picam2.capture_file(f"objets3D-{n:03d}.jpg")  
    time.sleep(1)  
    n += 1  
  
picam2.stop()
```

8.2 Programmes Python d'entraînement

1 train_YOLOv8.py

```
#####
# Jean-Luc.Charles@mailo.com
# 2024/12/18 - v1.1
#####

from pathlib import Path
from ultralytics import YOLO
from time import sleep

#
# this program must be run from the UCIA_ObjectDetection directory
#

def main(VER):

    BATCH = {'V1': (2, 4, 8, 10, 16, 20, 30), 'V2': (2, 4, 8, 16, 32), 'V2.1': (2, 4, 8, 16, 32) }
    EPOCH = (20, 40, 60, 80, 100)
    YOLO_SIZE = ('n', 's')
    model_dir = Path('./Training/YOLO-pretrained')
    data_path = {'V1': './datasets/V1.0_yolov8-50train-10val-2test_27nov2024/data.yaml',
                 'V2': './datasets/V2.0_yolo8-163train-38val-1test_09dec2024/data.yaml',
                 'V2.1': './datasets/V2.1_yolo8-128train-32val_12dec2024/data.yaml' }

    for size in YOLO_SIZE:

        yolo = 'YOLOv8' + size
        yolo_weights = yolo.lower() + '.pt'

        for batch in BATCH[VER]:
            for epoch in EPOCH:
                project = f'Training/YOLO-trained-{VER}/UCIA-{yolo}'
                name = f'batch-{batch:02d}_epo-{epoch:03d}'

                best = Path(project, name, 'weights', 'best.pt')
                print(f'{best}')
                if not best.exists():
                    model = YOLO(model_dir / yolo_weights) # load a pretrained model
                    model.train(data=data_path[VER],
                                epochs=epoch,
                                imgsz=640,
                                batch=batch,
                                patience=100,
                                cache=False,
                                workers=0,
                                project=project,
                                name=name,
                                exist_ok=True,
                                pretrained=True,
                                optimizer='auto',
                                seed=1234)

                print(f'looking for <best.onnx>... ')
                best_onnx = Path(project, name, 'weights', 'best.onnx')
                if not best_onnx.exists():
                    print("\t exporting <best.pt> to <best.onnx>...", end='')
                    model = YOLO(best) # load a custom trained model
                    model.export(format="onnx", int8=True, data=data_path[VER])
                    print(" done.")

                print(f'looking for <best.ncnn_model>... ')
                best_ncnn_model = Path(project, name, 'weights', 'best_ncnn_model')
                if not best_ncnn_model.exists():
                    print("\t exporting <best.pt> to <best_ncnn_model>...", end='')
                    model = YOLO(best) # load a custom trained model
                    model.export(format="ncnn", int8=True, data=data_path[VER])
                    print(" done.")

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('-v', '--version', action="store", dest='version', required=True, help="Work version: 'V1', 'V2' or 'V2.1'")
    args = parser.parse_args()
    version = args.version
    main(version)
```

2 train_YOLO11.py

```
#####
# Jean-Luc.Charles@mailo.com
# 2024/11/1 - v1.1
#####

from pathlib import Path
from ultralytics import YOLO
from time import sleep

#
# this programm must be run from the UCIA_ObjectDetection directory
#

def main(VER):

    BATCH = {'V1': (2, 4, 8, 10, 16, 20, 30), 'V2': (2, 4, 8, 16, 32), 'V2.1': (2, 4, 8, 16, 32)}
    EPOCH = (20, 40, 60, 80, 100)
    YOLO_SIZE = ('n', 's')

    model_dir = Path('./Training/YOLO-pretrained')

    data_path = {'V1': './datasets/V1.0_yolo11-50train-10val-2test_27nov2024/data.yaml',
                 'V2': './datasets/V2.0_yolo11-163train-38val-1test_09dec2024/data.yaml',
                 'V2.1': './datasets/V2.1_yolo11-128train-32val_12dec2024/data.yaml' }

    for size in YOLO_SIZE:

        yolo = 'YOLO11' + size
        yolo_weights = yolo.lower() + '.pt'

        for batch in BATCH[VER]:
            for epoch in EPOCH:
                project = f'Training/YOLO-trained-{VER}/UCIA-{yolo}'
                name = f'batch-{batch:02d}_epo-{epoch:03d}'

                best = Path(project, name, 'weights', 'best.pt')
                print(f'{best}')
                if not best.exists():
                    model = YOLO(model_dir / yolo_weights) # load a pretrained model
                    model.train(data=data_path[VER],
                               epochs=epoch,
                               imgsz=640,
                               batch=batch,
                               patience=100,
                               cache=False,
                               workers=0,
                               project=project,
                               name=name,
                               exist_ok=True,
                               pretrained=True,
                               optimizer='auto',
                               seed=1234)

                print(f'looking for <best.onnx>... ')
                best_onnx = Path(project, name, 'weights', 'best.onnx')
                if not best_onnx.exists():
                    print(f'\texporting <best.pt> to <best.onnx>...', end='')
                    model = YOLO(best) # load a custom trained model
                    model.export(format="onnx", int8=True, data=data_path[VER])
                    print(" done.")

                print(f'looking for <best_ncnn_model>... ')
                best_ncnn_model = Path(project, name, 'weights', 'best_ncnn_model')
                if not best_ncnn_model.exists():
                    print(f'\texporting <best.pt> to <best_ncnn_model>...', end='')
                    model = YOLO(best) # load a custom trained model
                    model.export(format="ncnn", int8=True, data=data_path[VER])
                    print(" done.")

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('-v', '--version', action="store", dest='version', required=True, help="Work version: 'V1', 'V2' or 'V2.1'")
    args = parser.parse_args()
    version = args.version
    main(version)
```

3 eval_YOLOv8n.py

```
#####
# Jean-Luc.Charles@mailo.com
# 2024/11/18 - v1.1
#####

from pathlib import Path
from ultralytics import YOLO
from time import sleep
import sys

def main(VER):

    BATCH = {'V1': (2, 4, 8, 10, 16, 20, 30), 'V2': (2, 4, 8, 16, 32), 'V2.1': (2, 4, 8, 16, 32) }
    EPOCH = (20, 40, 60, 80, 100)
    YOLO_SIZE = ('n', 's')

    model_dir = Path('./Training/YOLO-pretrained')

    data_path = {'V1': './datasets/V1.0_yolo8-50train-10val-2test_27nov2024/data.yaml',
                 'V2': './datasets/V2.0_yolo8-163train-38val-1test_09dec2024/data.yaml',
                 'V2.1': './datasets/V2.1_yolo8-128train-32val_12dec2024/data.yaml'}

    header = '#meta-params\tpre[ms]\tinf[ms]\tloss[ms]\tpost[ms]\t'
    header += 'prec\trecall\tmAP50\tmAP50-95\tfitness\n'
    header += '#pre:preprocessing; inf:inference; post:postprocessing; prec:precision\n'

    results_dir = Path('./Training/Results')
    if not results_dir.exists():
        results_dir.mkdir()

    for size in YOLO_SIZE:
        yolo = 'YOLOv8' + size
        yolo_weights = yolo.lower() + '.pt'

        # load any network for the first time:, because there is an overhead in computing the first time
        model = YOLO(f'./Training/YOLO-trained-{VER}/UCIA-YOLOv8{size}/batch-08_epo-020/weights/best.pt')
        metrics = model.val(batch=8, imgsz=640, data=data_path[VER], workers=0)

        results_file = Path(results_dir, f'results_yolov8{size}-{VER}.txt')
        F_out = open(results_file, "w", encoding="utf8")
        F_out.write(header)

        for batch in BATCH[VER]:
            for epoch in EPOCH:
                project = f'./Training/YOLO-trained-{VER}/UCIA-{yolo}'
                name = f'batch-{batch:02d}_epo-{epoch:03d}'

                best = Path(project, name, 'weights', 'best.pt')
                print(best)

                if best.exists():
                    model = YOLO(best) # load a pretrained model
                    # Validate the model
                    metrics = model.val(batch=batch, imgsz=640, data=data_path[VER], workers=0)
                    F_out.write(f'{name}\n')
                    for key in metrics.speed:
                        F_out.write(f'\t{metrics.speed[key]:.2f}\n')
                    for key in metrics.results_dict:
                        F_out.write(f'\t{metrics.results_dict[key]:.3f}\n')

                    F_out.write('\n')
                del model
            F_out.close()

    if __name__ == "__main__":
        import argparse
        parser = argparse.ArgumentParser()
        parser.add_argument('-v', '--version', action="store", dest='version', required=True, help="Work version: 'V1', 'V2' or 'V2.1'")
        args = parser.parse_args()
        version = args.version
        main(version)
```

4 eval_YOLOv11.py

```
#####
# Jean-Luc.Charles@mailo.com
# 2024/12/18 - v1.1
#####

from pathlib import Path
from ultralytics import YOLO
from time import sleep

def main(VER):

    data_path = {'V1': './datasets/V1.0_yolo11-50train-10val-2test_27nov2024/data.yaml',
                 'V2': './datasets/V2.0_yolo11-163train-38val-1test_09dec2024/data.yaml',
                 'V2.1': './datasets/V2.1_yolo11-128train-32val_12dec2024/data.yaml'}

    BATCH = {'V1': (2, 4, 8, 10, 16, 20, 30), 'V2': (2, 4, 8, 16, 32), 'V2.1': (2, 4, 8, 16, 32) }

    EPOCH = (20, 40, 60, 80, 100)
    YOLO_SIZE = ('n', 's')

    header = '#meta-params\tpre[ms]\tinfr[ms]\tloss[ms]\tpost[ms]\t'
    header += 'prec\trecall\tmAP50\tmAP50-95\tfitness\n'
    header += '#pre:preprocessing; inf:inference; post:postprocessing; prec:precision\n'

    results_dir = Path('./Training/Results')
    if not results_dir.exists():
        results_dir.mkdir()

    for size in YOLO_SIZE:
        yolo = 'YOLO11' + size
        yolo_weights = yolo.lower() + '.pt'

        # load any network for the first time: because there is an overhead in computing the first time
        model = YOLO(f"Training/YOLO-trained-{VER}/UCIA-YOLO11{size}/batch-08_epo-020/weights/best.pt")
        metrics = model.val(batch=8, imgsz=640, data=data_path[VER], workers=0)

    results_file = Path(results_dir, f"results_yolo11{size}-{VER}.txt")
    F_out = open(results_file, "w", encoding="utf8")
    F_out.write(header)

    for batch in BATCH[VER]:
        for epoch in EPOCH:
            project = f"Training/YOLO-trained-{VER}/UCIA-{yolo}"
            name = f"batch-{batch:02d}_epo-{epoch:03d}"

            best = Path(project, name, 'weights', 'best.pt')
            print(best)

            if best.exists():
                model = YOLO(best) # load a pretrained model
                # Validate the model
                metrics = model.val(batch=batch, imgsz=640, data=data_path[VER], workers=0)
                F_out.write(f'{name}\n')
                for key in metrics.speed:
                    F_out.write(f'\t{metrics.speed[key]:.2f}\n')
                for key in metrics.results_dict:
                    F_out.write(f'\t{metrics.results_dict[key]:.5f}\n')

            F_out.write('\n')
        del model
    F_out.close()

if __name__ == "__main__":

    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('-v', '--version', action="store", dest='version', required=True, help="Work version: 'V1', 'V2' or 'V2.1'")
    args = parser.parse_args()
    version = args.version
    main(version)
```

5 process_results.py

```
#####
# Jean-Luc.Charles@mailo.com
# 2024/12/18 - v1.1
#####

import pandas as pd
from pathlib import Path

def main(VER):

    results_dir = Path('./Training/Results')
    out_file = Path(results_dir, f'processed_res-{VER}.txt')

    with open(out_file, "w", encoding="utf8") as stream_out:
        mess = 50*'*' + "\n* Sort by 'recall' & 'mAP50-95'\n" + 50*'*'
        print(mess)
        stream_out.write(mess+'\n')

    for version in ('v8n', '11n', 'v8s', '11s'):
        txt_file = Path(results_dir, f'results_yolo{version}-{VER}.txt')

        mess = f'\nFile <{txt_file}>'
        print(mess, end='')
        stream_out.write(mess)
        # read CSV file with panda:
        df = pd.read_csv(txt_file, sep='\t', header=0, skiprows=[1])
        # sort rows by descending order of columns "recall", "mAP50-95":
        df = df.sort_values(by=["recall", "mAP50-95"], ascending=False)
        # the first values in columns "recall" and "mAP50-95" are the max values:
        max_mAP50_90 = df['mAP50-95'].values[0]
        max_recall = df['recall'].values[0]

        mess = f'\n\tMax values -> "max_recall": {max_recall}, "max_mAP50-90": {max_mAP50_90}'
        print(mess)
        stream_out.write(mess+'\n')

        # selected significant columns
        df1 = df[['#meta-params', 'recall', 'mAP50-95', 'fitness']]
        # print the first 4 rows:
        mess = df1.head(4)
        print(mess)
        stream_out.write(str(mess)+'\n')

    mess = '\n' + 50*'*' + "\n* Sort by 'fitness'\n" + 50*'*'
    print(mess)
    stream_out.write(mess+'\n')

    for version in ('v8n', '11n', 'v8s', '11s'):
        mess = f'\nFile <{txt_file}>'
        print(mess, end='')
        stream_out.write(mess)
        # read CSV file with panda:
        df = pd.read_csv(txt_file, sep='\t', header=0, skiprows=[1])
        # now sort rows by descending order of column "fitness":
        df = df.sort_values(by=["fitness"], ascending=False)
        # the first values in column "fitness" is the max values
        max_fitness = df['fitness'].values[0]

        mess = f'\n\tMax values -> "fitness": {max_fitness}'
        print(mess)
        stream_out.write(mess+'\n')

        # selected significant columns
        df2 = df[['#meta-params', 'recall', 'mAP50-95', 'fitness']]
        # print the first 4 rows:
        mess = df2.head(4)
        print(mess)
        stream_out.write(str(mess)+'\n')

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('-v', '--version', action="store", dest='version', required=True, help="Work version: 'V1', 'V2' or 'V2.1'")
    args = parser.parse_args()
    version = args.version
    main(version)
```


8.3 Fichiers résultats

1 results_yolov8n_V2.txt :

```
#meta-params          pre[ms]  inf[ms]  loss[ms] post[ms] prec  recall mAP50 mAP50-95
fitness
# pre:preprocessing; inf:inference; post:postprocessing; prec:precision
```

Tableau 5: Évaluation des entraînements du réseau yolov8n.

2 results_yolov8s_V2.txt :

```
#meta-params      pre[ms]  inf[ms]  loss[ms] post[ms] prec  recall mAP50 mAP50-95
fitness
# pre:preprocessing; inf:inference; post:postprocessing; prec:precision
```

*Tableau 6: Évaluation des entraînement du réseau **yolov8s**.*

3 results_yolo11n_V2.txt

```
#meta-params          pre[ms]  inf[ms]  loss[ms] post[ms] prec  recall mAP50 mAP50-95
fitness
# pre:preprocessing; inf:inference; post:postprocessing; prec:precision
```

*Tableau 7: Évaluation des entraînement du réseau **yolov11n**.*

4 results_yolo11s_V2.txt

```
#meta-params      pre[ms]  inf[ms]  loss[ms] post[ms] prec  recall mAP50 mAP50-95
fitness
# pre:preprocessing; inf:inference; post:postprocessing; prec:precision
```

*Tableau 8: Évaluation des entraînement du réseau **yolo11s**.*

9 Références

UCIA Cahier des charges : Robot ROSA avec Intelligence Artificielle OpenSource et OpenHardware

Page du site roboflow pour l'accès public au jeu de données de l'étude :

<https://universe.roboflow.com/ucia/ucia-ia-object-detection/dataset/2>

« Ultralytics YOLO11: Faster Than You Can Imagine! », Ankan Ghosh, October 8, 2024

<https://learnopencv.com/yolo11/>

Article Wikipédia sur les indicateurs de précision :

https://fr.wikipedia.org/wiki/Pr%C3%A9cision_et_rappel

« Analyse approfondie des mesures de performance », site Ultralytics

<https://docs.ultralytics.com/fr/guides/yolo-performance-metrics/>

« The Complete Guide to Object Detection Evaluation Metrics: From IoU to mAP and More »

<https://medium.com/@prathameshamrutkar3/the-complete-guide-to-object-detection-evaluation-metrics-from-iou-to-map-and-more-1a23c0ea3c9d>