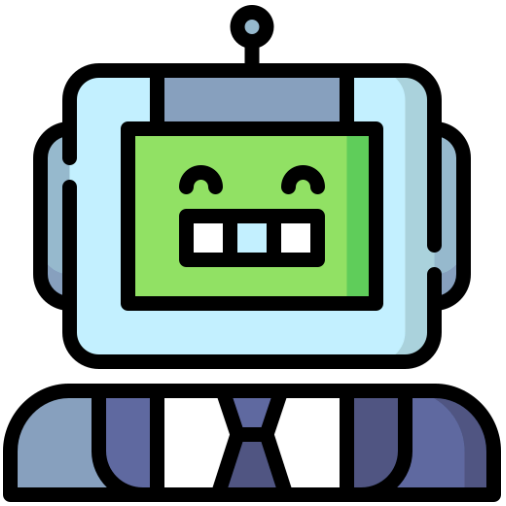
le cnam
école d'ingénieur·e·s

Rapport - Intelligence Artificielle

Conception d'une IA capable de reconnaitres certains objets



ultralytics

Picsellia

DREAM
CATCHER

Réalisé par :

Florian Spindler

Hamza Louda

William Boyard

Liens

Meilleurs model :
https://app.picsellia.com/0192f6db-86b6-784c-80e6-163debb242d5/model/01936427-7f8a-7dcc-a220-979309f2c978/version/0194eabe-cdde-720a-8c7a-3cb78b0c822b

Git Hub : https://github.com/leHofficiel/cnam-ia

Guide d'installation et lancement du projet

Prérequis

Avant de commencer, merci de bien vouloir avoir :

- Un environnement Python 3.11
- Accès au dépôt GitHub du projet

Ce guide ne supporte que l'utilisation de Windows en collaboration avec une carte graphique NVIDIA.

Installation

1. Clonez le dépôt GitHub :

```
git clone git@github.com:leHofficiel/cnam-ia.git
```

Pour pouvoir installation le programme, il faut se placer à la racine du dépôt git :

```
cd cnam-ia
```

2. Installation des dépendances Python:

```
pip3 install -r requirements.txt
```

3. Mise à jour de l'environnement ultralitics:

```
C:\Users\User\AppData\Roaming\Ultralytics\settings.json
```

Modifier les 3 variables suivantes

```
"datasets_dir": "Chemin absolu vers le projet cnam-ia",
  "weights_dir": "weights",
  "runs_dir": "runs",
```

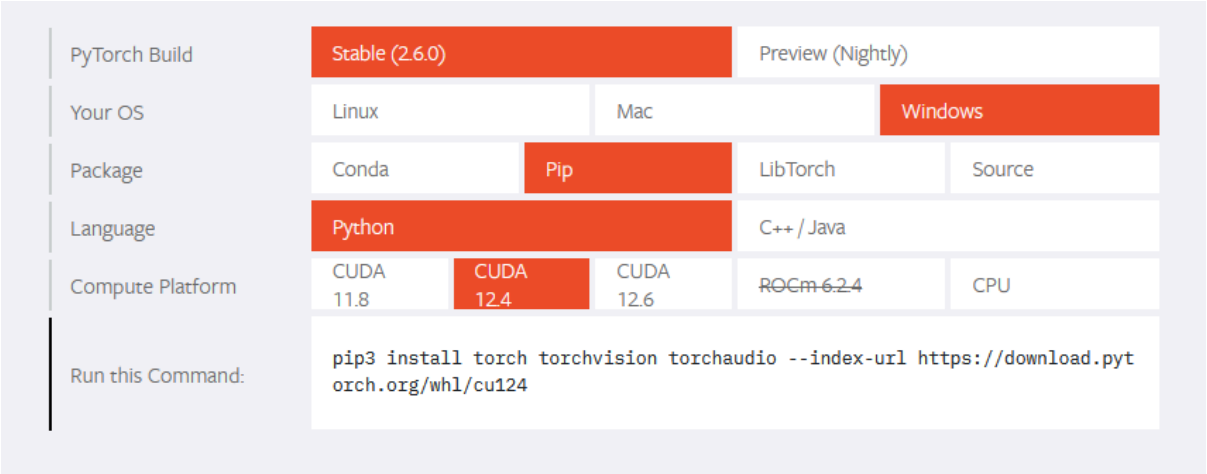
4. Installation de CUDA:

- Installer CUDA toolkit 12.4
- Désinstaller Pytorch avec cette commande avant de réinstaller la bonne version

```
pip uninstall torch torchvision
```

- Installer Pytorch à l'aide de la commande pip3 install disponible après sélection de la configuration

Exemple de configuration



Exécution du train

Configuration initiale

Avant de lancer l'entraînement, vous devez configurer le projet correctement.
Un fichier `config.yaml` est disponible à la racine du dépôt. Ce fichier permet de renseigner les informations suivantes pour interagir avec Picsellia:

```yaml
config:
  ORGA_NAME: ""
  API_TOKEN: ""
  DATASET_ID: ""
  PROJECT_NAME: ""
  EXPERIMENT_NAME: ""
```

De la même manière, un fichier `config_train.yaml` est également disponible afin de renseigner les hyperparamètres que vous voulez utiliser pour l'entrainement :

```yaml
config_train:
  data: "./dataset/yolo_config.yaml"
  epochs: 10
  batch: 32
  imgsz: 640
  device: "cuda"
  workers: 8
  optimizer: "auto"
  lr0: 0.01
  patience: 50
  seed: 4
  mosaic: 0
  close_mosaic: 0
```

Lancement du programme

Avant de lancer la détection, vous devez configurer le projet correctement.
Un fichier `config.yaml` est disponible à la racine du dépôt. Ce fichier permet de renseigner les informations suivantes pour interagir avec Picsellia:

```yaml
config:
  ORGA_NAME: ""
  API_TOKEN: ""
  DATASET_ID: ""
  PROJECT_NAME: ""
  EXPERIMENT_NAME: ""
```

Pour pouvoir exécuter le programme, il faut se placer à la racine du dépôt git :

```
cd cnam-ia
```

Cas 1 : Le dataset n’a pas encore été téléchargé

Exécutez la commande suivante pour nettoyer et préparer les données :

```
py .\SmarterBabySmarter\src\client_picsellia.py -clear
```

Cas 2 : Le dataset a déjà été téléchargé

Si le dataset a déjà été téléchargé, utilisez simplement la commande suivante :

```
py .\SmarterBabySmarter\src\client_picsellia.py
```

Exécution du modèle

Une fois votre modèle entraîné, vous pouvez l'exécuter selon trois modes différents :

1. Exécution sur des images

Pour exécuter le modèle sur un ensemble d'images, utilisez les commandes suivantes :

```
py .\SmarterBabySmarter\src\exec_model.py -image <chemin relatif du
dossier contenant les images>
```

2. Exécution sur des vidéo

Si vous souhaitez tester le modèle sur des vidéos, utilisez la commande suivante :

```
py .\SmarterBabySmarter\src\exec_model.py -video <chemin relatif du
dossier contenant les vidéos>
```

3. Exécution en temps réel (Webcam)

Pour exécuter le modèle en direct à partir d'une webcam, lancez la commande :

```
py .\SmarterBabySmarter\src\exec_model.py -webcam
```

Présentation du projet

Le projet SMART (Smart Merchandise Automated Recognition Technology) est une solution basée sur la Computer Vision permettant la reconnaissance automatique d'un ensemble défini de 10 produits à partir d'images, de vidéos ou de flux caméra en temps réel.

Mise en contexte

Dans un contexte d'automatisation et de reconnaissance d'objets, SMART vise à améliorer la reconnaissance de produits en magasin ou en entrepôt à l'aide de modèles de deep learning entraînés sur un dataset spécifique.

Présentation des produits détectables

Les produits détectables sont :

- Canettes
- Bouteilles en plastiques
- Pepito
- Kinder Country
- Kinder Tronky
- Kinder Pinguy
- Tic-Tac
- Sucette
- Capsule
- Mikado

Partie technique

Les technologies et les outils utilisés

Langage de programmation:
- Python 🐍

Frameworks et Bibliothèques d'IA :
- Ultralytics YOLO 🟢
- Picsellia 📊

Infrastructure et Environnements d'exécution
- Un GPU NVIDIA (CUDA) ⚡

Environnements de développement :
- PyCharm❤️

Outils de versioning et de gestion du code :
- Git 📒
- GitHub ☁️

Les hyperparamètres

Un hyperparamètre est un paramètre d'un modèle d'apprentissage qui est défini avant l'entraînement et qui influence la manière dont le modèle apprend. Les hyperparamètres sont définis à l'avance et ajustés manuellement ou via des techniques d'optimisation.

Les hyperparamètres du modèle influencent la performance de la détection. Nous avons expérimenté différents réglages sur les hyperparamètres suivant :

- **Learning rate**

  Détermine la vitesse à laquelle le modèle met à jour ses poids pendant l'entraînement

- **Batch size**

  Nombre d'exemples utilisés pour chaque mise à jour des poids du modèle

- **Nombre d'époques**

  Nombre de fois où l'algorithme voit l'ensemble des données d'entraînement.

- **Augmentations de données**

  Techniques utilisées pour enrichir le dataset et améliorer la généralisation en modifiant les images :

  Flip horizontal/vertical : Fait pivoter les images.
  Rotation, scaling : Modifie la taille et l'angle.
  Modification des couleurs : Change la luminosité, contraste, bruit.
  Mosaic : Combine plusieurs images pour améliorer la robustesse.

- YOLO propose différentes **tailles de modèles** selon le compromis **performance/rapidité** :

  Nano : Très rapide, faible précision.
  Small : Bon compromis rapidité/précision.
  Medium/Large : Plus précis, mais plus lent.
  XL : Très puissant, mais lourd à entraîner.

Le choix des bons hyperparamètres peut grandement améliorer la précision et la performance du modèle.
Exemple :

- Si le nombre d'époques est trop grand alors le modèle peut avoir un overfitting.
- Si le learning rate est trop élevé alors le modèle n'arrivera pas à converger.

Notre IA sans l'utilisation des hyperparamètres

La première étape de notre expérimentation a consisté à sélectionner la variante de YOLO (n, s ou m) qui servirait de base pour la suite du projet. Afin d'accélérer l'entraînement, nous avons utilisé les hyperparamètres par défaut de YOLO afin d'évaluer les performances brutes de chaque modèle, à l'exception de la taille de lot (batch size), qui a été ajustée pour optimiser l'utilisation des ressources GPU.

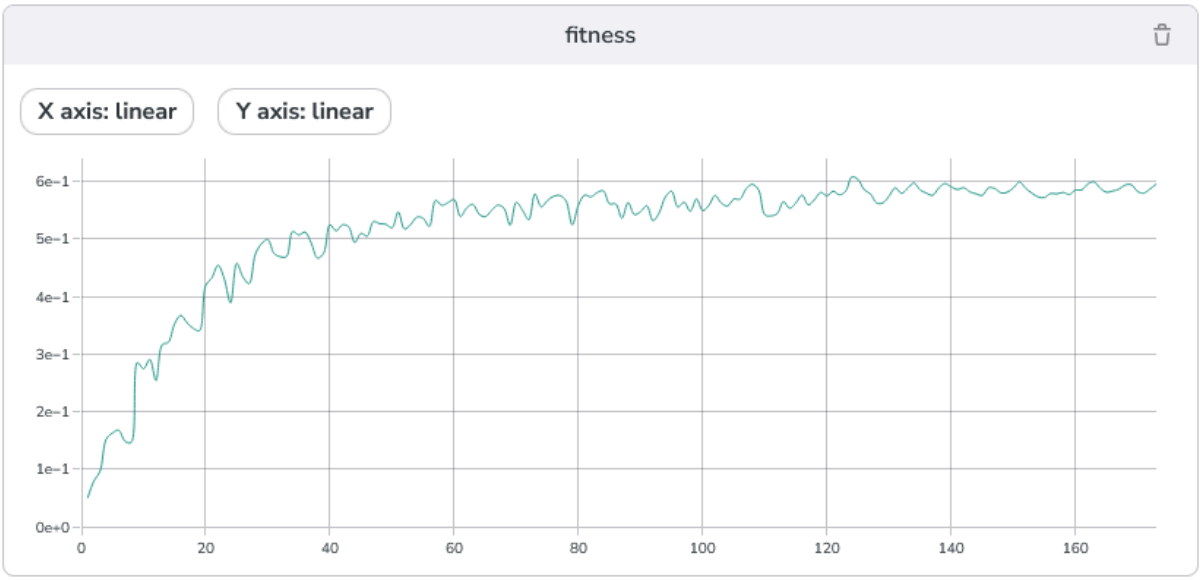
Voici la configuration utilisée :

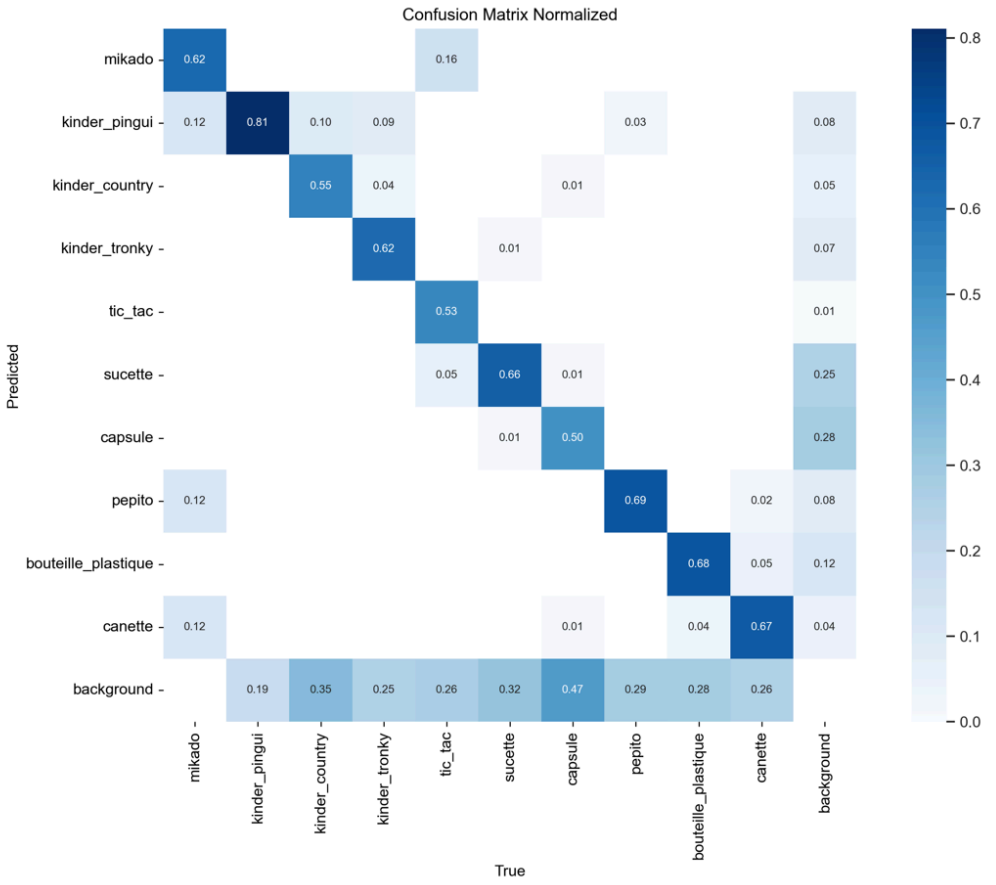
```yaml
config_train:
 data: "./dataset/yolo_config.yaml"
 epochs: 200
 batch: 32
 imgsz: 640
 device: "cuda"
 workers: 8
 optimizer: "auto"
 patience: 50
 seed: 42
 mosaic: 0
 close_mosaic: 0
```

- **epochs** : Limite maximale fixée à **200 epochs**, bien que l'entraînement puisse s'arrêter plus tôt grâce au critère de patience.
- **batch size** : 32, pour équilibrer vitesse et utilisation mémoire.
- **imgsz** : 640 pixels, une taille d'image standard pour le compromis entre précision et rapidité.
- **device** : Entraînement effectué sur **GPU (CUDA)**.
- **workers** : 8 processus en parallèle pour accélérer le chargement des données.
- **optimizer** : Automatique, laissant YOLO choisir l'optimiseur le plus adapté.
- **patience** : Arrêt anticipé si aucune amélioration significative après 50 epochs.

Modèle YOLO-N (Nano) : Gahyun

L'entraînement du modèle **Gahyun** a été effectué sur 200 epochs. Ci-dessous, le graphique représentant les résultats.

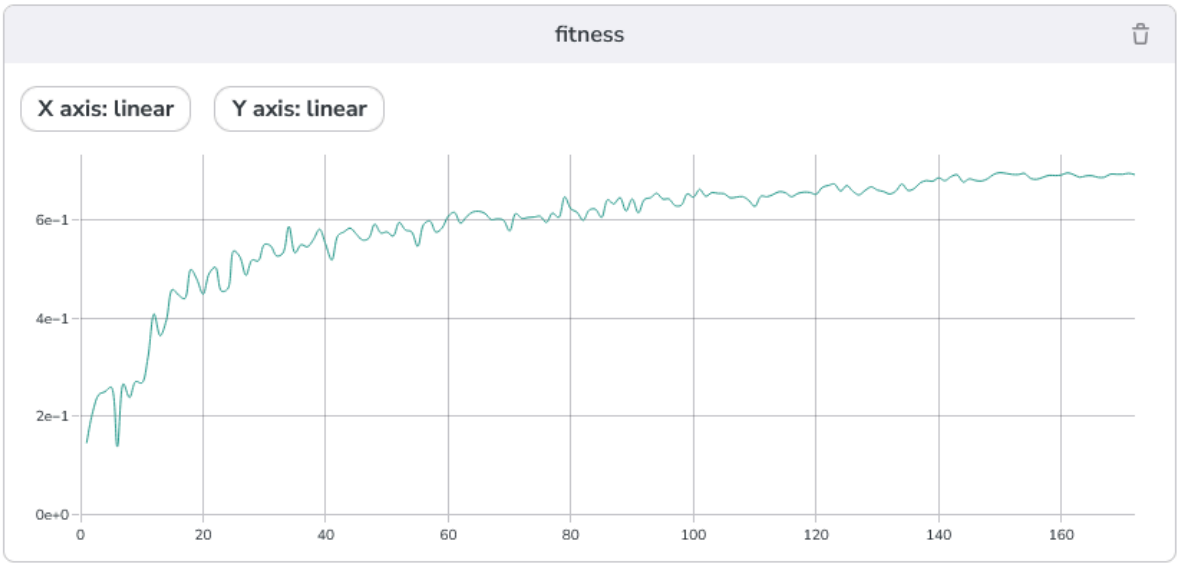




Le training a été interrompu à 175 epochs car la patience a été atteinte. La fitness maximum enregistrée lors du training est de 0.60. La durée totale de l'entraînement était de 25 minutes.

Modèle YOLO-S (Small) : SUA_Kim_Bora

Pour le modèle **SUA_Kim_Bora**, nous avons également lancé un entraînement sur 200 epochs. Vous trouverez ci-dessous le graphique illustrant les résultats :

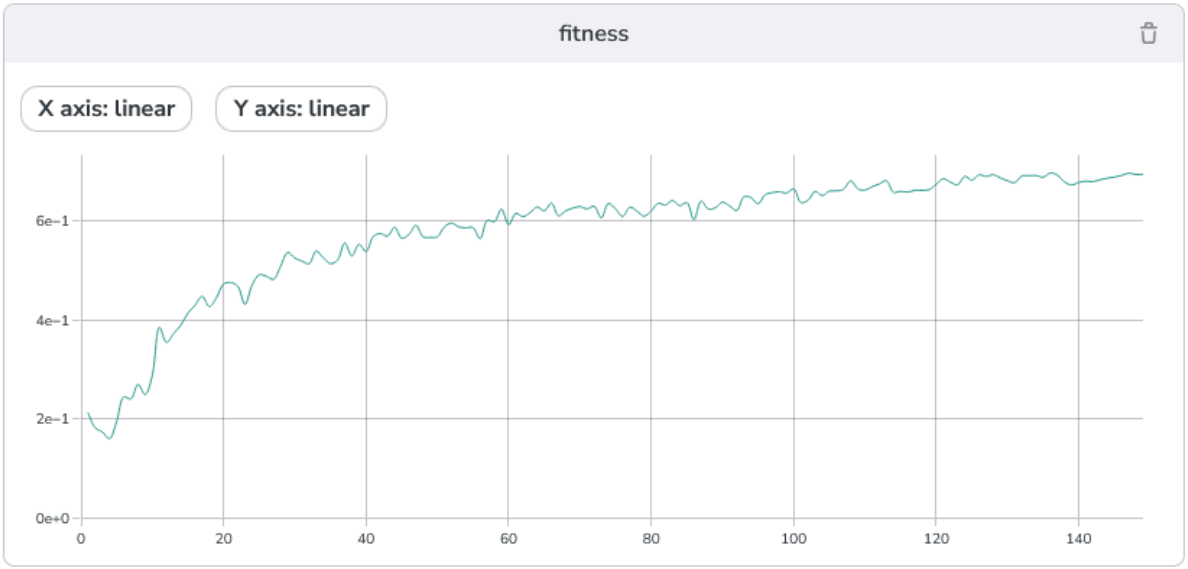


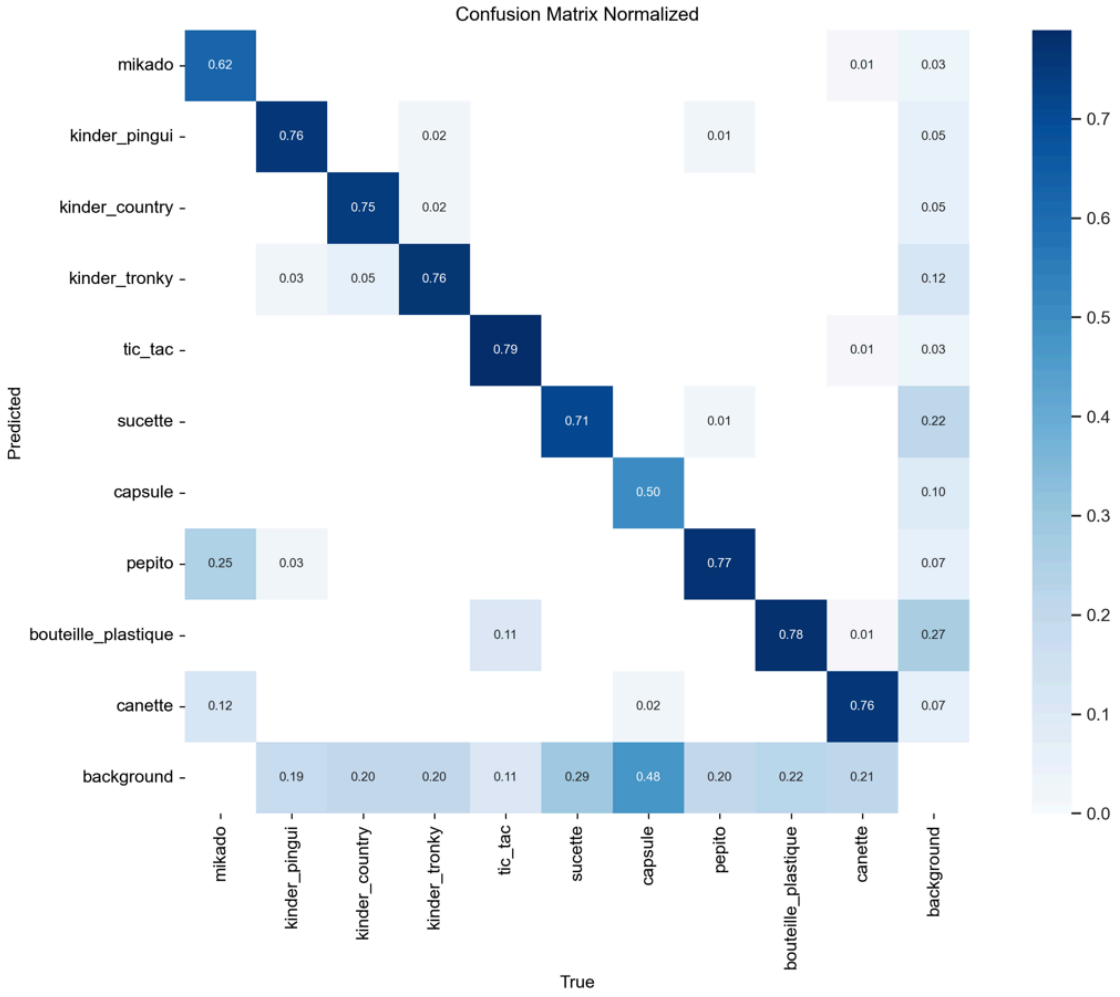
L'entraînement a été interrompu prématurément suite à un problème technique rencontré. L'incident a été causé par une **perte de connexion** avec **Picsellia…**. De ce fait, la matrice de confusion et les autres métriques n'ont pas pu être générées et par conséquent ne sont pas disponibles dans les artifacts sur Picsellia.

La fitness maximale atteinte lors du training est de 0.694. La durée totale de l'entraînement était de 35 minutes.

Modèle YOLO-M (Medium) : Handong

Pour le modèle **Handong**, nous avons lancé un entraînement sur 150 epochs. Vous trouverez ci-dessous le graphique illustrant les résultats :





Le training est stoppé à 150 epochs. La fitness maximale atteinte lors du training est de 0.694. La durée totale de l'entraînement était de 50 minutes.

Conclusion sur les différents modèles

Au cours de nos premières expérimentations, nous avons comparé trois variantes de YOLO: **YOLO-N (Nano), YOLO-S (Small) et YOLO-M (Medium)**. Chacune de ces versions présente des avantages et des inconvénients en termes de précision, rapidité et consommation de ressources.

Bien que **YOLO-S** et **YOLO-M** offrent de **meilleures performances** en termes de précision, ils demandent davantage de ressources informatiques et un temps d'entraînement plus long.

Après analyse, nous avons choisi **YOLO-N** comme modèle de référence pour la suite du projet, pour plusieurs raisons :

1. **Rapidité d'entraînement** 🏃: Son temps d'entraînement est nettement plus court, ce qui nous permet d'obtenir des résultats plus rapidement.

2. **Consommation de ressources** ⚡ : Le modèle N sollicite moins le processeur et la carte graphique, laissant la possibilité d'exécuter d'autres tâches simultanément sur la même machine.
3. **Suffisance pour notre cas d'utilisation** 🎯: Malgré des performances globalement plus modestes que YOLO-s ou YOLO-m, les résultats de YOLO-n demeurent satisfaisants pour notre application. Ils constituent un bon compromis entre efficacité et rapidité, sans saturer les ressources disponibles.

En conclusion, **YOLO-N** s'est imposé comme le choix le plus adapté pour notre besoin actuel, alliant efficacité, rapidité et faible consommation de ressources. Ce choix nous permet de poursuivre nos expérimentations de manière plus agile et itérative, tout en gardant la possibilité d'explorer d'autres modèles si nécessaire.

Notre IA avec les meilleurs hyperparamètres

Recherche des meilleurs hyperparamètres

Une fois notre modèle sélectionné, il reste néanmoins possible d'améliorer ses performances. Pour ce faire, il est nécessaire de procéder à un **ajustement des hyperparamètres**, c'est-à-dire de modifier certains paramètres (taux d'apprentissage, fonctions de régularisation, etc.) afin d'optimiser les résultats. Cette étape est cruciale pour **affiner la précision**, **réduire le surapprentissage** et tirer pleinement profit de la capacité d'apprentissage de notre modèle.

Dans cette section, nous allons détailler :

1. **Les hyperparamètres clés** à considérer pour notre architecture YOLO.
2. **Les méthodes** pour trouver les meilleurs hyperparamètres.

L'objectif principal de cette phase est de déterminer la configuration d'hyperparamètres la plus adaptée à notre cas d'utilisation. Cela nous permettra d'optimiser à la fois la **précision** du modèle et sa **vitesse d'exécution**, tout en tenant compte des **contraintes matérielles** de notre environnement de calcul.

Les hyperparamètres clés

Les hyperparamètres essentiels sur lesquels nous pouvons agir incluent notamment le **taux d'apprentissage** (*learning rate*), le **nombre d'époques** (*epochs*), la **patience** (pour l'*early stopping*) et le **choix de l'optimiseur** (*optimizer*). Bien d'autres hyperparamètres entrent également en jeu, certains liés à l'augmentation des données (*data augmentation*), d'autres plus spécifiques et parfois moins maîtrisés. Leur complexité peut rendre leur compréhension et leur ajustement plus difficiles, mais ils n'en demeurent pas moins importants pour affiner la performance du modèle.

Les méthodes de recherche

Nous avons principalement exploré **trois méthodes** pour déterminer les meilleurs hyperparamètres :

1. **Recherche aléatoire** : Nous avons procédé à des essais en sélectionnant de manière aléatoire différentes valeurs pour les hyperparamètres, afin de couvrir un large éventail de configurations possibles.
2. **Ajustements progressifs** : À partir des résultats obtenus, nous avons affiné manuellement certains hyperparamètres en fonction des tendances observées (amélioration de la précision, réduction du sur-apprentissage, etc.), ce qui nous a permis d'effectuer des ajustements plus fins et ciblés.
3. **Recherche automatique (Model Tuning)** : Nous avons également utilisé des outils de *model tuning* automatisés qui testent et comparent différentes combinaisons d'hyperparamètres, en s'appuyant sur des algorithmes de recherche (comme l'optimisation bayésienne) pour accélérer et optimiser la sélection.

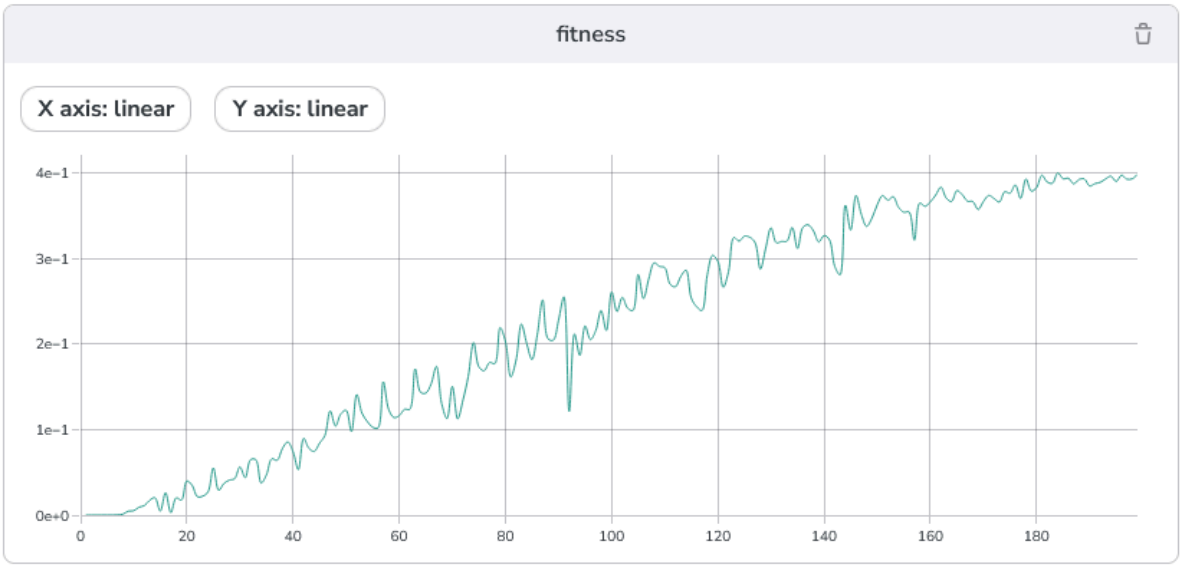
Exécution de la recherche

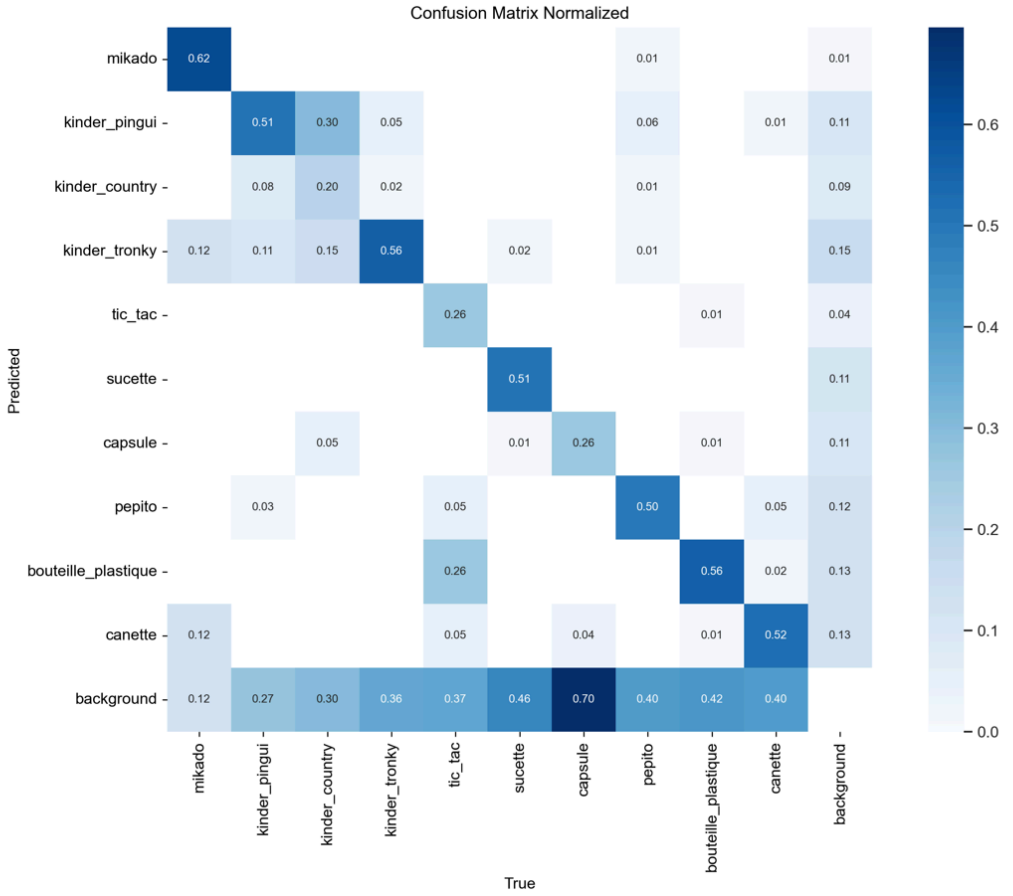
Model **Yoohyeon**

Pour ce modèle, nous avons décidé de reprendre les mêmes hyperparamètres que pour le modèle **Gahyun** à quelques exceptions près. Notre objectif était de modifier aléatoirement certains paramètres, notamment le **taux d'apprentissage** (*learning rate*) et l'**optimiseur**, afin d'observer l'impact de ces changements sur les performances.

Voici la configuration de l'entraînement :

```
config_train:
 data: "./dataset/yolo_config.yaml"
 epochs: 200
 batch: 32
 imgsz: 640
 device: "cuda"
 workers: 8
 optimizer: "AdamW"
 patience: 50
 seed: 42
 mosaic: 0
 close_mosaic: 0
 lr0: 0.025
```





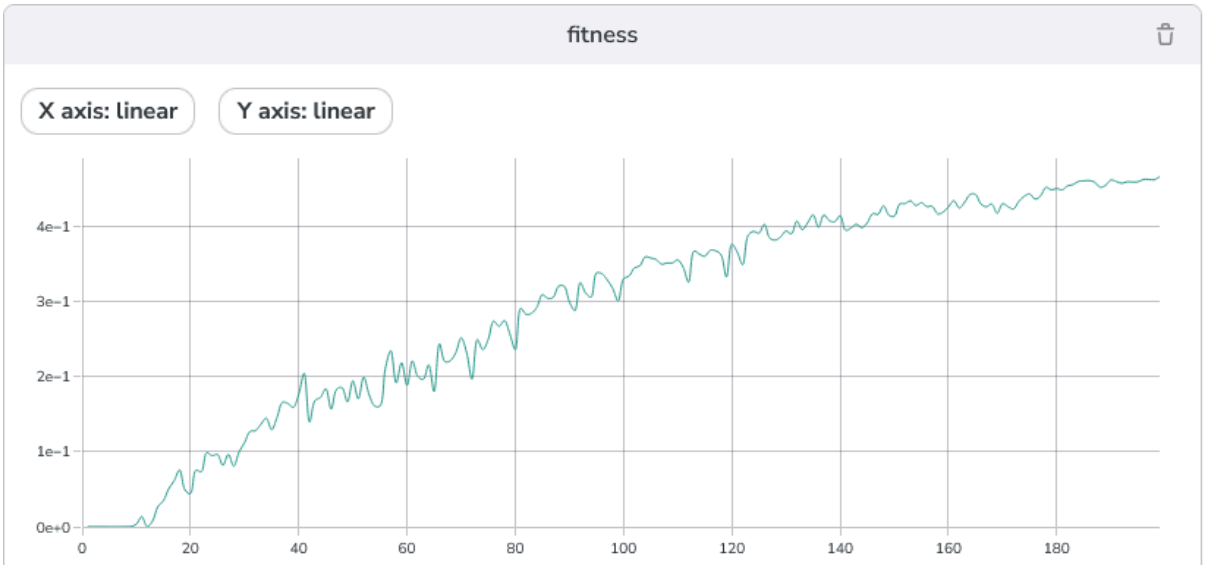
Malheureusement, cette approche aléatoire ne s'est pas avérée concluante. Les performances obtenues sont même inférieures à celles observées avec les paramètres par défaut. Cela souligne l'importance d'une méthode plus systématique et réfléchie pour sélectionner les hyperparamètres, plutôt qu'une simple approche par essais aléatoires.

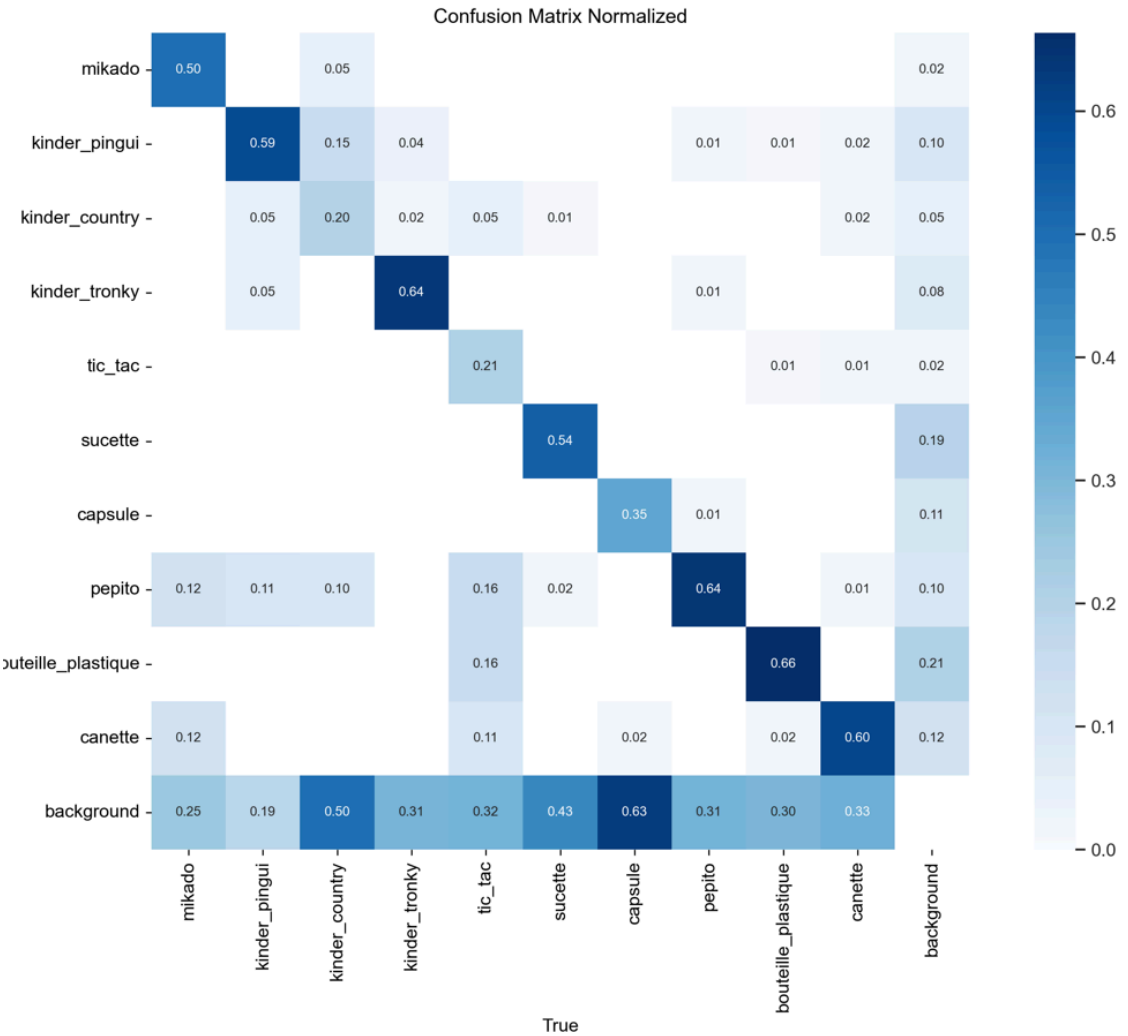
Model **Jiu**

Afin de **poursuivre nos expérimentations**, nous avons décidé d'employer un algorithme de *model tuning* pour ce modèle, en l'exécutant avec des paramètres de base sur **80 époques** et **70 itérations**. À l'issue de ce processus, nous avons obtenu la configuration d'hyperparamètres suivante :

```
lr0: 0.00923
lrf: 0.01177
momentum: 0.98
weight_decay: 0.00044
warmup_epochs: 3.06095
warmup_momentum: 0.82572
box: 6.68439
cls: 0.42109
dfl: 1.20487
hsv_h: 0.01034
hsv_s: 0.63122
hsv_v: 0.39781
degrees: 0.0
translate: 0.12352
scale: 0.76731
shear: 0.0
perspective: 0.0
flipud: 0.0
fliplr: 0.48081
bgr: 0.0
mosaic: 0.88263
mixup: 0.0
copy_paste: 0.0
```

Nous avons ensuite entraîné le **modèle Jiu** avec ces paramètres sur **200 époques**. **Malheureusement**, les résultats obtenus ont été décevants : la *fitness* maximale atteinte s'est limitée à **0,49**.





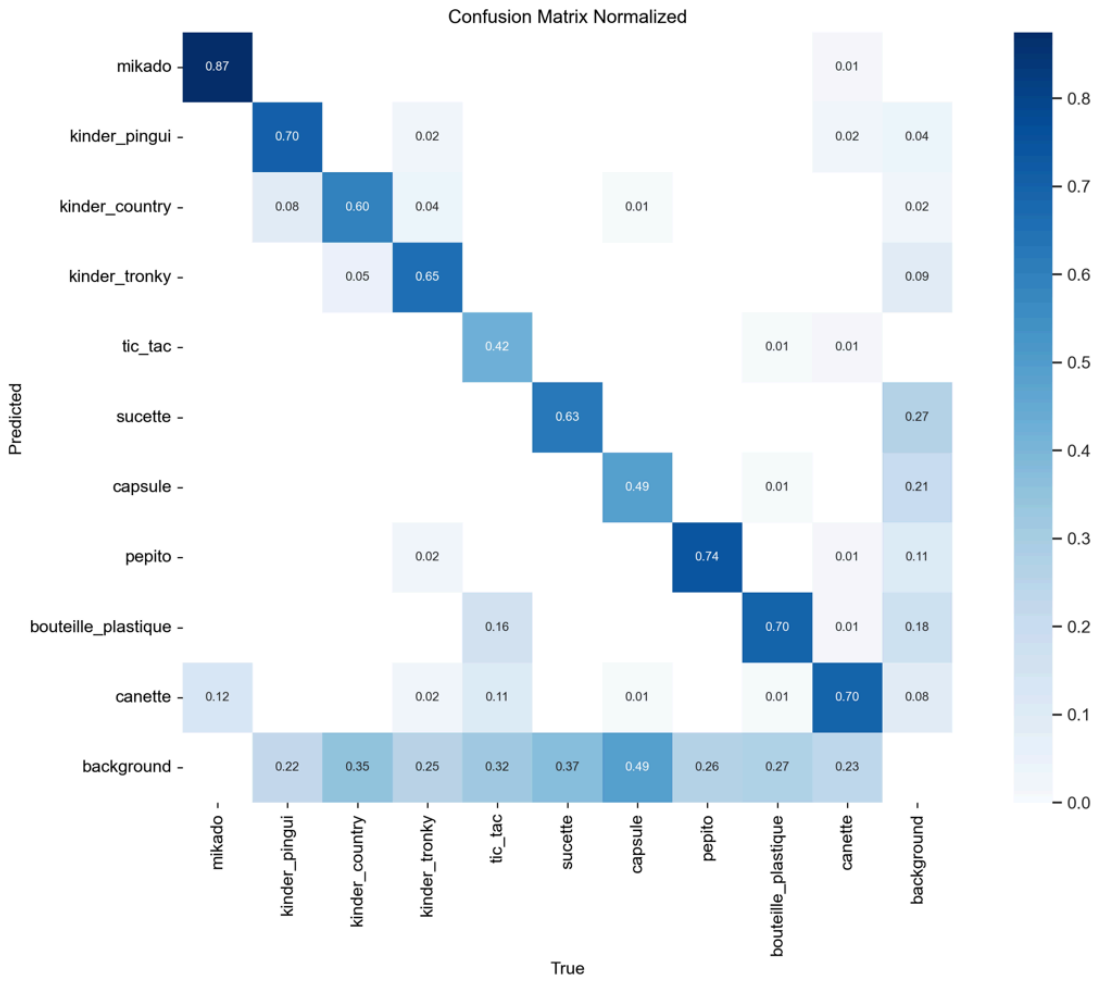
Malgré le recours à un algorithme d'optimisation automatique pour régler les hyperparamètres, les performances du modèle Jiu restent en deçà de nos attentes.

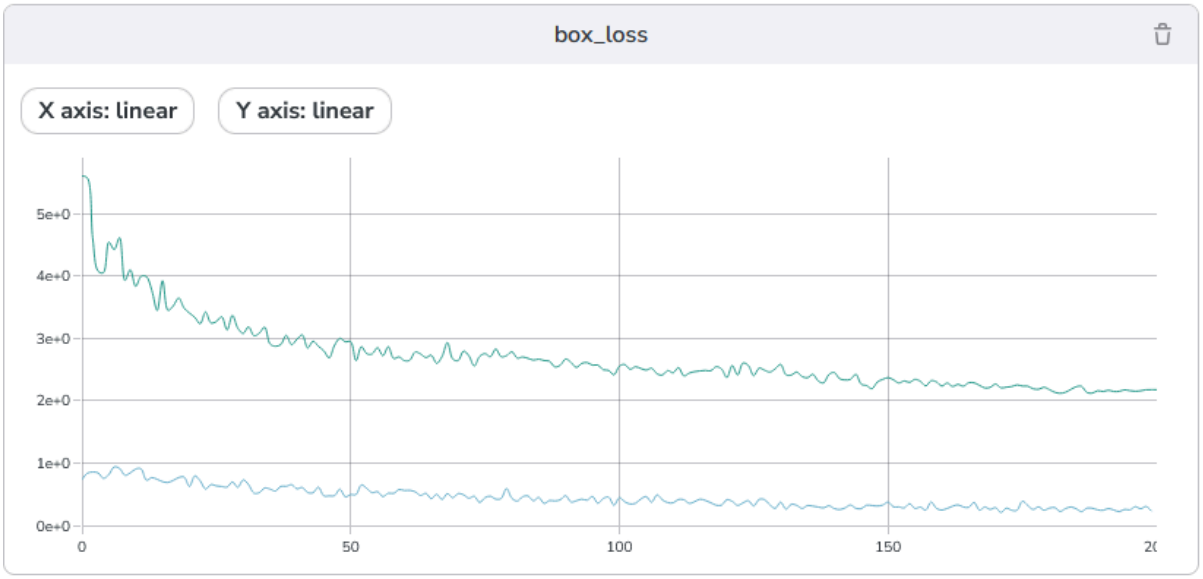
Model Dami

Nous avons décidé de faire un autre tuning car le précédent n'était pas concluant. Les paramètres utilisés pour ce tuning étaient ceux de base avec 80 epoch et 70 itérations. Cela nous a permis d'obtenir cela comme paramètre.

```
lr0: 0.00179
lrf: 0.01518
momentum: 0.86333
weight_decay: 0.00051
warmup_epochs: 3.09569
warmup_momentum: 0.44584
box: 5.1931
cls: 0.74142
dfl: 1.16798
hsv_h: 0.01322
hsv_s: 0.41755
hsv_v: 0.20555
degrees: 0.0
translate: 0.0979
scale: 0.4822
shear: 0.0
perspective: 0.0
flipud: 0.0
fliplr: 0.16048
bgr: 0.0
mosaic: 0.0
mixup: 0.0
copy_paste: 0.0
```

Nous avons donc lancé le modèle **Dami** avec ces paramètre pour une durée de 200 epochs


Confusion Matrix Normalized
mikado
kinder_pingui
kinder_country
kinder_tronky
tic_tac
sucette
capsule
pepito
bouteille_plastique
canette
background
Predicted
True
0.87 0.01
0.70 0.02 0.02 0.04
0.08 0.60 0.04 0.01 0.02
0.05 0.65 0.09
0.42 0.01 0.01
0.63 0.27
0.49 0.01 0.21
0.02 0.74 0.01 0.11
0.16 0.70 0.01 0.18
0.12 0.02 0.11 0.01 0.01 0.70 0.08
0.22 0.35 0.25 0.32 0.37 0.49 0.26 0.27 0.23
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0


box_loss
X axis: linear
Y axis: linear
5e+0
4e+0
3e+0
2e+0
1e+0
0e+0
0
50
100
150

Ce modèle parvient à obtenir une **fitness de 0,63**, un résultat tout à fait satisfaisant au regard de nos précédentes expérimentations. Toutefois, l'examen des métriques et des courbes d'entraînement met en évidence un **phénomène d'underfitting** : le modèle semble ne pas exploiter pleinement l'ensemble des informations disponibles dans les données.

Model Siyeon

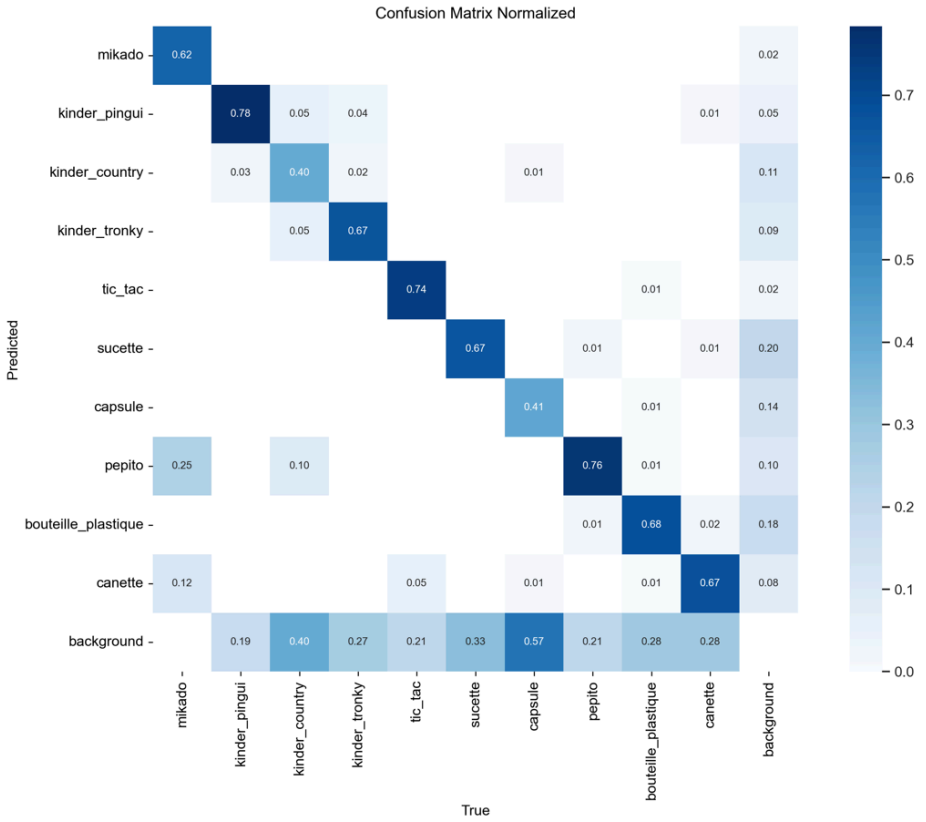
Suite aux résultats prometteurs obtenus avec le modèle **Dami**, nous avons décidé d'entraîner un nouveau modèle, **Siyeon**, en conservant les mêmes hyperparamètres.

La principale différence entre ces deux modèles réside dans le nombre d'epochs. Lors de l'analyse des performances de **Dami**, nous avons observé des signes d'**underfitting**, indiquant un **apprentissage insuffisant**. Cela suggérait que le modèle n'avait pas encore pleinement convergé et qu'un nombre plus élevé d'epochs pourrait permettre d'obtenir de meilleures performances.

C'est pourquoi, pour **Siyeon**, nous avons décidé d'augmenter le nombre d'époch afin d'évaluer si cette adaptation permettrait d'améliorer la capacité du modèle et réduire l'**underfitting** observé avec Dami.

```
lr0: 0.00179
lrf: 0.01518
momentum: 0.86333
weight_decay: 0.00051
warmup_epochs: 3.09569
warmup_momentum: 0.44584
box: 5.1931
cls: 0.74142
dfl: 1.16798
hsv_h: 0.01322
hsv_s: 0.41755
hsv_v: 0.20555
degrees: 0.0
translate: 0.0979
scale: 0.4822
shear: 0.0
perspective: 0.0
flipud: 0.0
fliplr: 0.16048
bgr: 0.0
mosaic: 0.0
mixup: 0.0
copy_paste: 0.0
```

Nous avons donc lancé le modèle **Siyeon** avec ces paramètre pour une durée de 500 epochs



Malgré l'augmentation du nombre d'époch par rapport au modèle **Dami**, les résultats obtenus avec **Siyeon** se sont révélés moins bons. Bien que nous ayons espéré améliorer les performances en allouant plus de temps d'entraînement, le modèle n'a pas réussi à surpasser Dami en termes de précision et de performance globale. L'entraînement a été interrompu prématurément car la **patience** a été atteinte : aucune amélioration significative des résultats n'a été observée pendant la durée allouée à la **patience**, ce qui a conduit à un arrêt anticipé. Cela suggère que l'augmentation du nombre d'époch n'a pas eu l'effet escompté, et que d'autres ajustements seraient nécessaires pour optimiser les performances du modèle **Siyeon**.

Difficultés rencontrés

Nous avons rencontré différentes difficultés de l'ordre technique ou fonctionnelle.

L'outil Picsellia nous a incités à approfondir de manière autonome nos connaissances. Nous avons rencontré certaines difficultés pour obtenir des réponses à nos questions, notamment concernant la gestion des logs et l'utilisation du SDK.

Par ailleurs, l'outil de pre-commit a posé plusieurs défis. Nous avons constaté des faux positifs que nous n'avons pas réussi à résoudre, ce qui a empêché leur utilisation effective. De surcroît, l'implémentation tardive des pre-commits n'a fait qu' exacerber ces difficultés.

Axes d'amélioration du projet

Plusieurs pistes d'amélioration sont envisageables :

Dans un premier temps, nous n'avons pas observé d'overfitting des modèles. Il serait intéressant d'augmenter le nombre d'époques et de relever le seuil de patience afin de forcer l'apparition de ce phénomène et de l'étudier.

Dans un second temps, nous pourrions envisager de lancer une phase de modèle tuning plus prolongée pour identifier et valider les hyperparamètres les plus performants.

Enfin, pour surpasser l'IA de Bastian, il serait nécessaire de disposer de ressources matérielles supplémentaires. Par exemple, l'acquisition de plusieurs cartes graphiques Nvidia RTX 4090 pourrait constituer une solution pour améliorer significativement les performances

Conclusion

Le meilleur modèle obtenu à ce jour est **celui de Handong**, reposant sur **YOLO M**, qui présente les performances les plus élevées. Toutefois, si l'on se limite exclusivement à la gamme **YOLO N**, c'est le **modèle Dami** qui offre les meilleurs résultats. Grâce à l'**optimisation de certains hyperparamètres**, il parvient en effet à surpasser le modèle YOLO N par défaut en termes de précision.

✨👑 Reine Handong 👑✨