CentraleSupélec

université PARIS-SACLAY

# Rapport de projet S8

**Application Android pour la localisation et la reconnaissance d'objets dans un flux vidéo**

*Réalisé par :*
Apolline Bornet - apolline.bornet@student-cs.fr
Matéo Dischler - mateo.dischler@student-cs.fr
Rebecca Bayssari - rebecca.bayssari@student-cs.fr

*Encadré par :*
Jean-Luc Colette - jean-luc.collette@centralesupelec.fr
Michel Ianotto - michel.ianotto@centralesupelec.fr

10 juin 2024

# Table des matières

# Chapitre 1

# Introduction

## 1.1 Contexte et Objectif

La détection d'objets est un domaine fascinant et en constante évolution, offrant de nombreuses opportunités tant pour la recherche que pour les applications pratiques, comme la sécurité, la navigation assistée ou le commerce de détail. Avec les avancées rapides en intelligence artificielle et en vision par ordinateur, les capacités des systèmes de détection d'objets se sont considérablement améliorées, permettant des applications de plus en plus sophistiquées et diversifiées.

Dans ce contexte dynamique, notre projet se concentre sur le développement d'une application Android dédiée à la localisation et à la reconnaissance d'objets dans un flux vidéo en temps réel. Les smartphones, omniprésents dans notre quotidien, offrent une plateforme idéale pour exploiter ces technologies de pointe grâce à leur facilité d'utilisation et leur accessibilité. En effet, une application mobile permet de démocratiser l'accès à des fonctionnalités avancées sans nécessiter d'équipement spécialisé ou de connaissances techniques approfondies de la part des utilisateurs.

## 1.2 Structure du rapport

Dans un premier temps, nous détaillerons le processus de génération de notre modèle de détection d'objets. Nous commencerons par un état de l'art des modèles de détection d'objets existants. Ensuite, nous présenterons le modèle que nous avons utilisé ainsi que la base de données employée. Nous expliquerons le processus d'entraînement et effectuerons une analyse des performances obtenues.
Dans un second temps, nous décrirons la construction et l'intégration de ce modèle dans une application Android en utilisant Android Studio.

# Chapitre 2

# Détection d'objet

## 2.1 Etat de l'art

La détection d'objets est une tâche cruciale en vision par ordinateur, consistant à identifier et à localiser des objets d'intérêt dans une image ou une vidéo. Ces avancées reposent en grande partie sur les réseaux de neurones, des modèles d'apprentissage automatique inspirés du fonctionnement du cerveau humain. Les réseaux de neurones convolutifs (Convolutional Neural Networks ou CNN), en particulier, ont révolutionné le domaine en permettant une extraction automatique et hiérarchisée des caractéristiques des images, conduisant à des performances inédites en termes de précision et de vitesse.

Les réseaux de neurones convolutifs (CNN) se composent de plusieurs couches de neurones artificiels qui filtrent et analysent les données d'entrée de manière hiérarchique. Les couches convolutives permettent de détecter des motifs locaux tels que les bords, les textures et les formes, qui sont ensuite combinés dans des couches plus profondes pour identifier des objets complexes. Cette capacité à apprendre des représentations riches et discriminatives à partir des données brutes a fait des CNN l'épine dorsale des modèles modernes de détection d'objets.

Les modèles de détection d'objets utilisant des CNN peuvent être classés en deux grandes catégories : les modèles en un temps (one-stage) et les modèles en deux temps (two-stage). Chacune de ces approches présente des avantages et des inconvénients, en fonction des besoins spécifiques en termes de vitesse et de précision.

**Modèles en un temps (one-stage)**
Les modèles en un temps effectuent la détection d'objets en une seule étape, en prédisant directement les classes et les emplacements des objets à partir des caractéristiques extraites de l'image. Ces modèles sont généralement plus rapides et mieux adaptés aux applications en temps réel. Parmi les modèles en un temps les plus connus, on peut citer :

- YOLO (You Only Look Once) : Développé par Joseph Redmon et al, YOLO est l'un des modèles de détection d'objets les plus populaires pour les applications en temps réel. Il divise l'image en une grille et prédit simultanément les classes et les boîtes englobantes pour plusieurs régions de l'image.
- SSD (Single Shot MultiBox Detector) : Introduit par Wei Liu et al., SSD améliore la précision tout en maintenant une grande vitesse de traitement. Il utilise des convolutions de différentes tailles pour détecter des objets à différentes échelles, ce qui le rend efficace pour la détection d'objets de diverses tailles.
- RetinaNet : Proposé par Tsung-Yi Lin et al., RetinaNet introduit la "Focal Loss", qui accorde plus d'importance aux exemples difficiles à classifier, améliorant ainsi la précision des détections tout en conservant une rapidité d'exécution.

**Modèles en deux temps (two-stage)**
Les modèles en deux temps divisent le processus de détection en deux étapes distinctes : d'abord, ils génèrent des propositions de régions d'intérêt (Region Proposals) qui pourraient contenir des objets, puis ces propositions sont affinées et classifiées. Ces modèles sont généralement plus précis mais plus lents que les modèles en un temps. Les modèles en deux temps les plus célèbres incluent :

- R-CNN (Regions with Convolutional Neural Networks) : Proposé par Ross Girshick et al., R-CNN est l'un des premiers modèles de détection en deux temps. Il génère des propositions de régions à l'aide de sélecteurs de recherche sélective et utilise des réseaux de neurones convolutifs pour extraire des caractéristiques de chaque région.

- Fast R-CNN : Également développé par Ross Girshick, Fast R-CNN améliore l'efficacité en intégrant la phase d'extraction de caractéristiques dans le réseau de détection, réduisant ainsi les redondances de calcul.
- Faster R-CNN : Présenté par Shaoqing Ren et al., Faster R-CNN introduit le Réseau de Propositions de Régions (Region Proposal Network, RPN), qui génère des propositions de régions directement au sein du réseau de détection, améliorant ainsi significativement la vitesse et la précision.
- Mask R-CNN : Une extension de Faster R-CNN proposée par Kaiming He et al., Mask R-CNN ajoute une branche supplémentaire pour la segmentation des objets, permettant ainsi non seulement de détecter et de classifier les objets, mais aussi de générer des masques de segmentation pour chaque objet détecté.

Les progrès dans les modèles de détection d'objets, qu'ils soient en un temps ou en deux temps, ont considérablement amélioré les capacités de vision par ordinateur. Les modèles en un temps, tels que YOLO et SSD, offrent des solutions rapides et adaptées aux applications en temps réel, tandis que les modèles en deux temps, comme Faster R-CNN et Mask R-CNN, fournissent une précision et une robustesse accrues pour des applications nécessitant une détection précise. Le choix du modèle dépend largement des exigences spécifiques de l'application envisagée, notamment en termes de vitesse et de précision.

## 2.2 YOLO

Pour notre application de détection et de reconnaissance d'objets en temps réel, nous avons choisi d'utiliser le modèle YOLO (You Only Look Once). Ce choix s'explique par le fait que YOLO est un modèle en un temps, ce qui le rend particulièrement rapide et approprié pour les applications nécessitant une détection en temps réel. De plus, YOLO bénéficie d'une abondante documentation et d'une large communauté de développeurs, facilitant ainsi son implémentation et son optimisation.

YOLO (You Only Look Once) est un modèle de détection d'objets développé par Joseph Redmon et ses collègues, introduit pour la première fois en 2016 dans l'article "You Only Look Once : Unified, Real-Time Object Detection". YOLO a été conçu pour offrir une alternative rapide et précise aux méthodes de détection d'objets existantes, répondant aux exigences croissantes des applications en temps réel comme la surveillance, la robotique, et les véhicules autonomes.

Contrairement aux modèles en deux temps (two-stage) qui séparent la génération des propositions de régions et la classification des objets en deux étapes distinctes, YOLO adopte une approche unifiée en une seule étape. Voici le fonctionnement détaillé de YOLO :

- Division de l'Image en Grilles : L'image d'entrée est divisée en une grille de S×S cellules. Chaque cellule est responsable de prédire plusieurs boîtes englobantes (bounding boxes) et leurs scores de confiance pour déterminer la probabilité qu'une boîte contienne un objet.
- Prédictions : Chaque cellule de la grille prédit B boîtes englobantes et leurs scores de confiance. De plus, chaque boîte est associée à une classe d'objet parmi C classes possibles. Ainsi, pour chaque cellule, YOLO prédit B boîtes englobantes, B scores de confiance et C scores de classe.
- Représentation des Boîtes : Les prédictions des boîtes incluent les coordonnées (x,y) du centre de la boîte, sa largeur w, sa hauteur h et un score de confiance. Les coordonnées sont normalisées par rapport à la taille de la cellule de la grille, ce qui facilite l'apprentissage.
- Suppression Non Maximale (NMS) : YOLO utilise l'algorithme de suppression non maximale pour éliminer les prédictions redondantes en conservant uniquement les boîtes avec le score de confiance le plus élevé pour chaque objet détecté.

### 2.2.1 Présentation de YOLOv8

Dans le domaine de la vision par ordinateur, la série YOLO (You Only Look Once) est reconnue pour son efficacité et sa précision dans la détection d'objets en temps réel. YOLOv8, développé par Ultralytics, est la dernière itération de cette série, apportant des améliorations significatives en termes de performances et d'expérience de développement par rapport aux versions antérieures telles que YOLOv5. Cette version avancée combine haute précision et rapidité d'exécution, se distinguant nettement des versions précédentes grâce à ses optimisations substantielles.

### 2.2.2 Avantages et performance de YOLOv8

YOLOv8 se distingue par son équilibre optimal entre précision et vitesse, rendant ce modèle particulièrement adapté pour des applications exigeantes telles que la surveillance de sécurité, la navigation autonome, et la reconnaissance d'objets dans divers environnements industriels et urbains. Cette amélioration de la précision est soutenue par des ajustements fins dans l'architecture du réseau, y compris de meilleurs mécanismes d'attention et des stratégies de fusion de caractéristiques plus sophistiquées.

La comparaison entre YOLOv8 et les versions antérieures, telles que YOLOv7, YOLOv6.2.0, et YOLOv5.7.0, montre des améliorations tangibles tant en termes de précision qu'en matière de performances de traitement. Comme le montre le graphique ci-dessous, YOLOv8 atteint des niveaux plus élevés de mAP (mean Average Precision) tout en maintenant ou en réduisant la latence, illustrant ainsi son efficacité accrue.

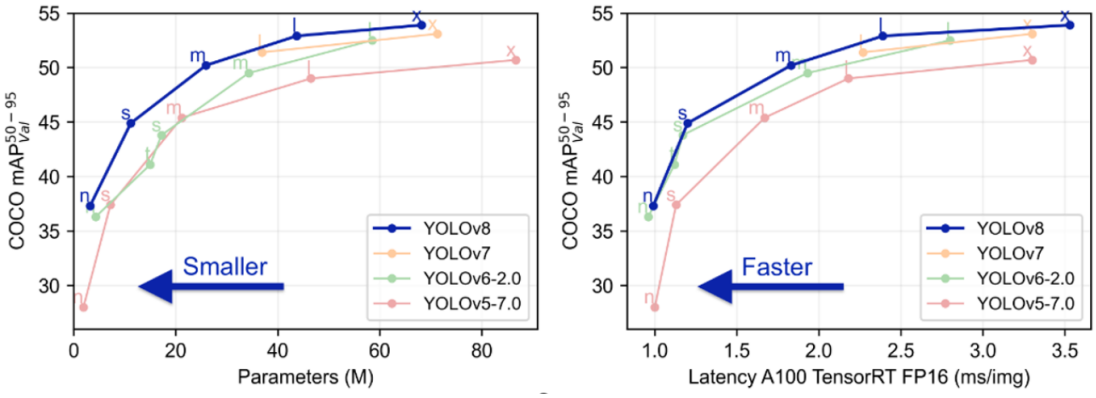


FIGURE 2.1 – Comparaison de performance entre YOLOv8 et ses prédécesseurs : précision et rapidité

De même,la récente évaluation de YOLOv8 sur le benchmark Roboflow 100 a démontré son excellente capacité à généraliser à de nouveaux domaines. Ce benchmark, issu de Roboflow Universe, comprend 100 jeux de données représentatifs de divers domaines. YOLOv8 y a surpassé ses prédécesseurs, YOLOv5 et YOLOv7, montrant moins d'outliers et une meilleure mAP globale. Les graphiques de performance confirment que YOLOv8 présente moins de variance et de meilleures performances moyennes, excellant systématiquement par rapport aux modèles précédents dans différentes catégories de Roboflow 100.

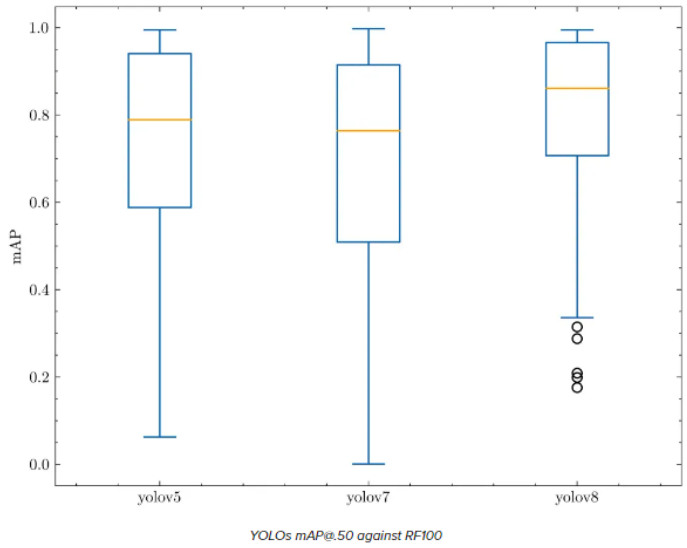


FIGURE 2.2 – Analyse de la précision sur le Benchmark RF100 : Comparaison des versions YOLOv5, YOLOv7 et YOLOv8

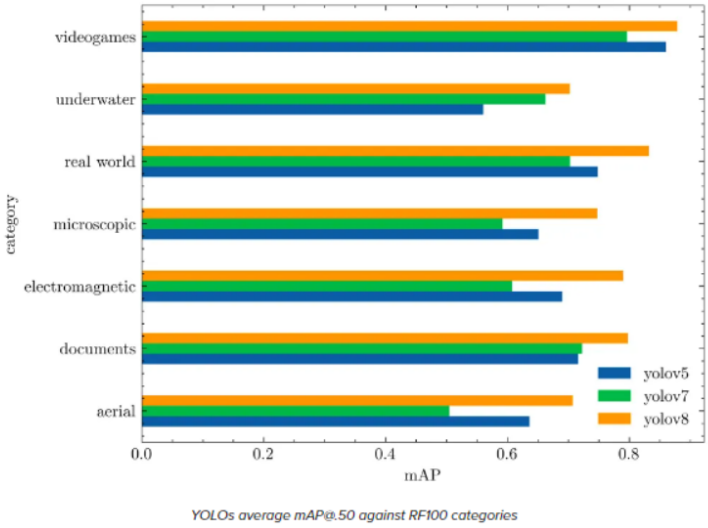


FIGURE 2.3 – Évaluation de précision des modèles YOLO dans divers domaines d'application

### 2.2.3 Architecture de Yolov8

L'architecture de YOLOv8 comprend trois composants principaux : le backbone, le neck et le head. Chacun joue un rôle crucial dans le processus de détection d'objets.

- **Backbone :**
  Le backbone de YOLOv8 emploie des blocs CSP pour une extraction efficace des caractéristiques, minimisant la redondance et améliorant l'efficience du calcul. Il divise les entrées en deux flux parallèles, où l'un passe par des transformations à travers des convolutions et des connexions résiduelles, et l'autre flux passe directement au bout du bloc où les deux sont ensuite concaténés. Le choix des paramètres comme le kernel, le stride, et le padding est crucial pour l'optimisation de cette extraction. Ces éléments permettent de contrôler la résolution et la dimensionnalité des sorties du backbone, affectant directement la précision et la vitesse de l'architecture.
  Un élément notable dans le backbone de YOLOv8 est l'intégration du bloc SPPF (Spatial Pyramid Pooling - Fast), qui est une évolution de la technique classique de Spatial Pyramid Pooling (SPP). Le SPPF améliore la capacité du réseau à gérer des entrées de différentes tailles et à capturer des informations contextuelles à diverses échelles spatiales. Il effectue un pooling multidimensionnel sur les cartes de caractéristiques à la sortie du backbone, générant des sorties de taille fixe indépendamment de la taille de l'entrée, ce qui est essentiel pour les couches suivantes qui nécessitent une taille d'entrée standardisée.
- **Neck :**
  Le neck utilise l'architecture FPN pour enrichir les caractéristiques extraites du backbone. Il intègre des opérations d'upsample pour aligner les cartes de caractéristiques de différentes résolutions, les préparant pour une fusion plus efficace. Ce processus crée une pyramide de caractéristiques enrichies, facilitant la détection d'objets à différents niveaux de détail. Les connexions de raccourci entre le backbone et le neck, ainsi qu'entre le neck et le head, sont essentielles pour la transmission efficace des informations à travers le réseau, aidant à maintenir l'intégrité des données et à améliorer l'apprentissage.
- **Head :** Le head de YOLOv8 est responsable de la prédiction des classes et des boîtes englobantes des objets détectés. Il interprète les caractéristiques combinées provenant de la pyramide de caractéristiques enrichies par le neck pour réaliser les prédictions finales, assurant une détection précise et rapide des objets à différentes échelles.

YOLOv8 représente un jalon dans les architectures de détection d'objets grâce à l'intégration de techniques avancées telles que les blocs CSP, SPPF, et C2F, ainsi que les configurations de bottleneck. Chaque composant, du backbone au head en passant par le neck, est optimisé pour fonctionner de manière cohérente, fournissant des performances exceptionnelles pour les applications en temps réel

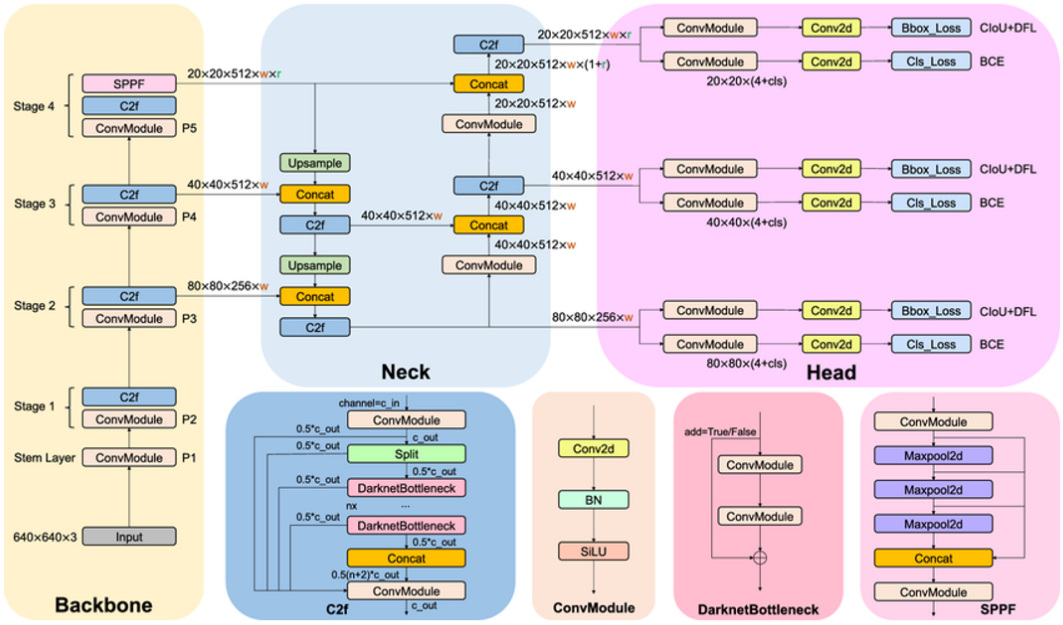


FIGURE 2.4 – Architecture Détaillée de YOLOv8 : Composants et Flux de Traitement

### 2.2.4 Intégration de YOLOv8 avec TensorBoard

L'amélioration de la compréhension et du réglage des modèles de vision par ordinateur, tels que YOLOv8 d'Ultralytics, devient plus accessible lorsque l'on examine de près leurs processus de formation. La visualisation de la formation des modèles aide à obtenir des aperçus sur les schémas d'apprentissage, les métriques de performance et le comportement général. L'intégration de YOLOv8 avec TensorBoard facilite ce processus de visualisation et d'analyse, permettant des ajustements plus efficaces et informés du modèle. TensorBoard, l'outil de visualisation de TensorFlow, est essentiel pour l'expérimentation en apprentissage automatique. Il propose une gamme d'outils de visualisation cruciaux pour le suivi des modèles d'apprentissage machine, incluant le suivi des métriques clés telles que la perte et la précision, la visualisation des graphes des modèles, et l'affichage des histogrammes des poids et biais au fil du temps. Il offre également des capacités pour projeter les embeddings dans des espaces de dimensions inférieures et afficher des données multimédias.

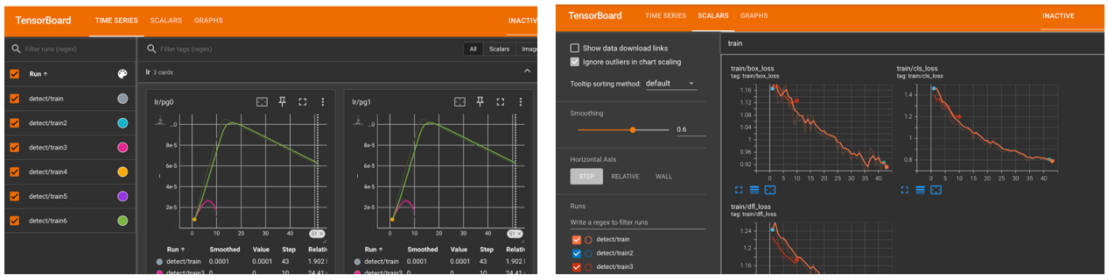


FIGURE 2.5 – Visualisation des Métriques de Formation avec TensorBoard pour YOLOv8

### 2.2.5 Conclusion

En résumé, YOLOv8 se démarque en tant que solution de pointe pour détecter des objets, combinant une précision remarquable et une rapidité remarquable avec une facilité d'utilisation et une accessibilité accrues grâce à des outils de visualisation avancés. Sa popularité croissante dans différents domaines témoigne de sa capacité à satisfaire les exigences des applications de vision par ordinateur modernes, ce qui fait de YOLOv8 un choix privilégié pour les chercheurs et les professionnels du domaine.

## 2.3 Base de données

Maintenant que nous avons choisi notre modèle, nous devons sélectionner la base de données sur laquelle nous allons l'entraîner.

La base de données que nous avons choisi (voir figure 2.6) est constitué de 3343 images de fleurs réparties en 13 classes , qui sont les suivantes :

- Lantana commun
- Hibiscus
- Jatropha
- Souci
- Rose
- Champaka
- Chitrak
- Chèvrefeuille
- Guimauve indienne
- Melastome de Malabar
- Shankupushpam
- Lis araignée
- Tournesol

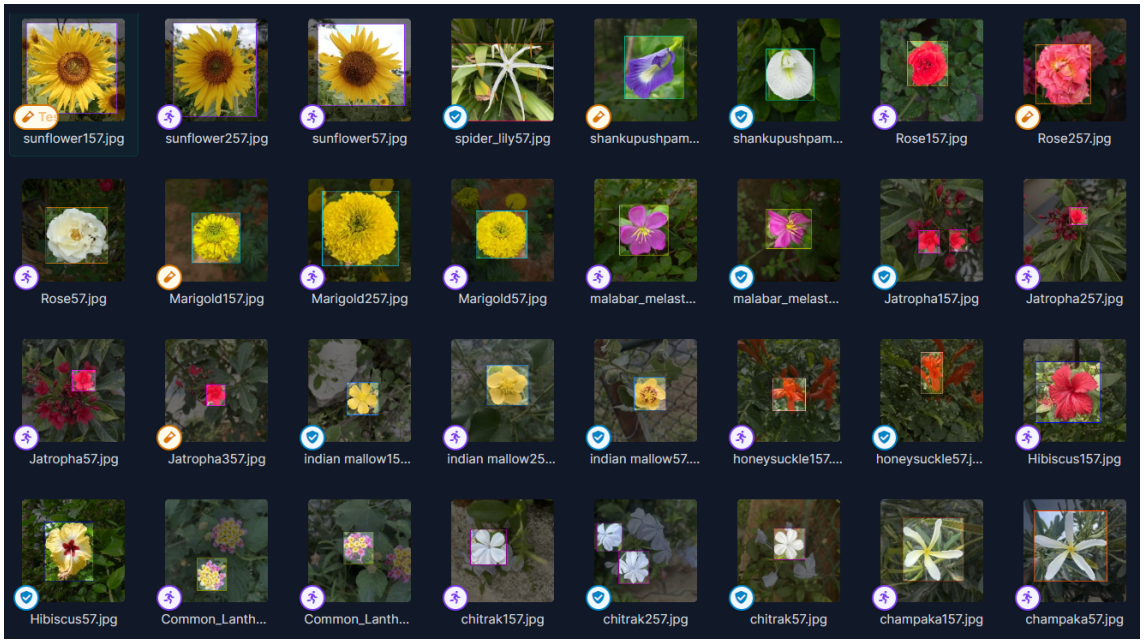


FIGURE 2.6 – Extrait de la base de données

Cette base de données sera divisé en trois sous-ensembles :

- L'ensemble d'entraînement qui va servir à apprendre les caractéristiques des différentes classes de fleurs au modèle.
- L'ensemble de validation qui sert à évaluer les performances du modèle pendant l'entraînement et à ajuster les hyperparamètres pour éviter le surapprentissage.
- L'ensemble de test, quant à lui, sert à évaluer les performances finales du modèle sur des données qu'il n'a jamais vues afin de vérifier sa capacité de généralisation.

Pour pouvoir entraîner le modèle YOLO (You Only Look Once) sur cette base de données, il est nécessaire que les données soient structurées d'une manière spécifique (voir figure 2.7).

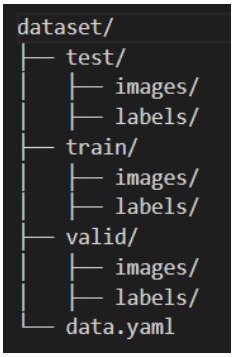


FIGURE 2.7 – Structure de la base de données

Dans les dossiers "labels", se trouvent des fichier de format *.txt*. Ils contiennent les annotations des images. Chaque ligne correspond à un objet à détecter sur l'image. Le premier nombre (un entier) correspond au numéro de la classe à laquelle l'objet appartient. Les deuxième et troisième nombres correspondent aux coordonnées normalisées du centre de l'objet. Enfin, les quatrième et cinquième nombres correspondent à la longueur et la largeur de l'objet.

La présence du fichier YAML est indispensable. Dans ce fichier, se trouvent les chemins des différents ensembles d'images et les labels correspondants. Ce fichier permet de définir la configuration et la structure des données pour que le modèle puisse les utiliser correctement lors de l'entraînement.

En analysant la base de données, nous pouvons constater que les éléments à détecter se trouvent tous au milieu de l'image et sont assez grands (voir figure 2.8). Cela signifie que notre modèle sera uniquement capable de détecter les éléments qui sont principalement situés au centre de la zone de détection.

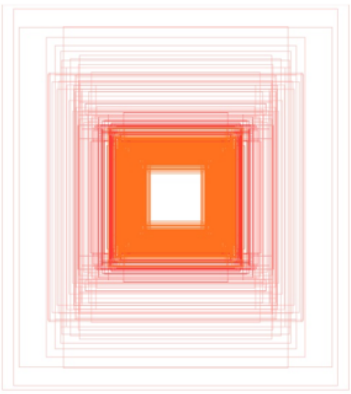

FIGURE 2.8 – Emplacements des bounding boxes dans les images

Cette particularité de la base de données a plusieurs implications sur les performances de notre modèle YOLO. En effet, comme les objets à détecter sont tous positionnés au centre, le modèle peut développer une dépendance à cette caractéristique. Par conséquent, si des objets sont placés en dehors de cette zone centrale lors de la phase d'inférence ou sont trop petits, le modèle pourrait ne pas les détecter correctement.

Pour atténuer cette limitation, il pourrait être bénéfique de diversifier la base de données en incluant des images où les objets sont situés à différentes positions et sous différentes tailles dans le cadre. Ainsi, le modèle pourrait apprendre à détecter les objets indépendamment de leur position dans l'image et de leur taille, améliorant ainsi sa robustesse et sa capacité de généralisation.

Nous avons trouvé notre dataset sur Roboflow (sur ce lien). Roboflow est une plateforme qui permet de télécharger des bases de données d'images annotées avec une structure correspondant aux exigences de divers modèles de vision par ordinateur, notamment celle compatible avec YOLOv8 (voir figure 2.7). Cette structure est essentielle pour garantir une intégration fluide des données dans notre pipeline d'entraînement et d'inférence.

Roboflow a grandement accéléré notre processus de développement, nous faisant gagner un temps précieux. En effet, nous n'avons pas eu besoin de capturer et d'annoter les images manuellement, ce qui aurait nécessité des ressources considérables en termes de temps et de main-d'œuvre. Cependant, il est important de noter que notre dépendance aux bases de données disponibles sur Roboflow limite quelque peu notre flexibilité et notre capacité à personnaliser l'ensemble de données selon nos besoins spécifiques.

## 2.4 Processus d'entraînement

Maintenant que nous avons le modèle et la base de données, nous allons pouvoir passer à l'entraînement. Pour cela, nous utiliserons le package Ultralytics, qui a été développé pour faciliter l'entraînement et le déploiement de modèles de détection d'objets, notamment pour YOLOv8.

Dans un premier temps, nous chargerons le modèle YOLOv8 pré-entraîné sur le jeu de données COCO, qui est une référence courante pour les tâches de détection d'objets. En utilisant ce modèle pré-entraîné comme point de départ, nous bénéficierons des connaissances déjà acquises sur la détection d'une large gamme d'objets dans des scènes variées.

Ensuite, nous demanderons à ce modèle de s'entraîner sur notre propre base de données. Au cours de cet entraînement, le modèle ajustera ses paramètres pour mieux reconnaître les classes d'objets qui nous intéressent, dans notre cas des fleurs.

À la fin de l'entraînement, le modèle générera des poids dans un fichier *.pt*, qui est un fichier de PyTorch. Ce fichier contient les valeurs des paramètres du modèle après l'entraînement. Enfin, nous convertirons ce fichier *.pt* en un fichier avec une extension *.tflite*. Le format *.tflite* est un choix populaire pour le déploiement de modèles de détection d'objets sur des appareils mobiles en raison de sa compatibilité, de sa performance et de son efficacité.

Cependant, lors de l'entraînement sur nos PC en local, nous n'avons réussi à effectuer que 20 epochs en 12 heures. Nous avons donc dû utiliser un GPU, qui est une unité de traitement graphique qui en raison de leur architecture parallèle ont la capacité de manipuler efficacement de grandes quantités de données. Cela nous a permis de réduire considérablement le temps d'entraînement, atteignant jusqu'à 100 epochs en seulement 12 minutes pour la même base de données et les mêmes paramètres.

Cependant, notre utilisation du GPU est limitée à 2 heures par session, ce qui nous empêche d'exploiter des bases de données plus grandes et plus diversifiées.

## 2.5 Analyse des performances

Une fois notre modèle entrainé (sur 200 epochs) nous pouvons évaluer les performances des poids choisis (fichier *.pt*).

Dans un premier temps, analysons les courbes d'entrainements (voir figure 2.9). Les courbes correspondent toutes à des mesures de performances en fonction des *epochs*. Ces performances sont mesurées sur l'ensemble d'entrainement quand le titre contient *train* et sur l'ensemble de validation quand il contient *val*.

La ***box loss*** est une mesure de l'erreur entre les boîtes prédictives générées par le modèle et les boîtes réelles. Cette perte est calculée en utilisant la fonction de perte de régression de boîte :

$$L_{\text{box}} = \lambda_{\text{coord}} \left( (\hat{x} - x)^2 + (\hat{y} - y)^2 \right) + \lambda_{\text{size}} \left( (\hat{w} - w)^2 + (\hat{h} - h)^2 \right)$$

où $\hat{b} = (\hat{x}, \hat{y}, \hat{w}, \hat{h})$ représente les prédictions du modèle pour le centre de la boîte $(\hat{x}, \hat{y})$, la largeur $\hat{w}$ et la hauteur $\hat{h}$, et $b = (x, y, w, h)$ les coordonnées, longueur et largeur réelles correspondantes. Les termes $\lambda_{\text{coord}}$ et $\lambda_{\text{size}}$ sont des hyperparamètres qui pondèrent les différentes composantes de la perte.

La ***cl loss*** est une mesure de l'erreur entre les classes prédictives des objets dans les boîtes englobantes et les classes réelles. Cette perte est calculée en utilisant la fonction de perte d'entropie croisée, qui compare les probabilités de classe prédites par le modèle aux classes réelles.

Soit $\hat{p}_i$ la probabilité prédite pour la classe $i$ par le modèle, et $y_i$ l'appartenance à la classe réelle (1 si l'objet appartient à la classe $i$, 0 sinon). La perte de classification peut être exprimée comme suit :

$$L_{\text{cl}} = -\sum_{i=1}^{C} y_i \log(\hat{p}_i)$$

où $C$ est le nombre total de classes.

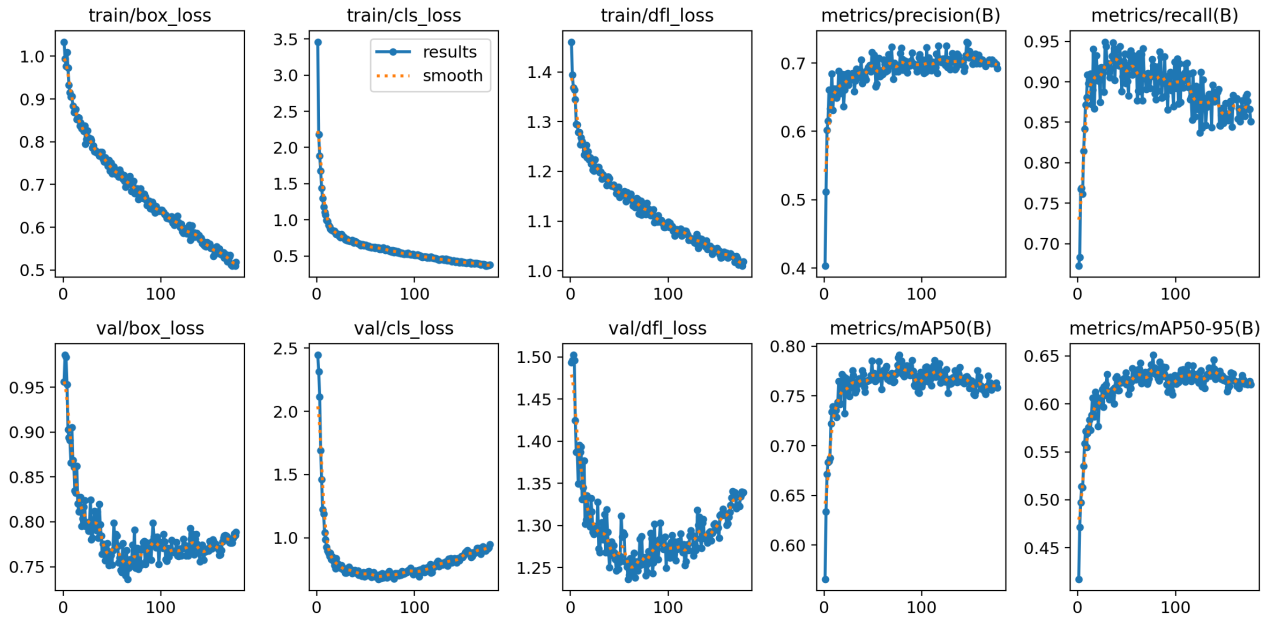


FIGURE 2.9 – Courbes d'entrainement

Cette formule pénalise fortement les erreurs de classification en augmentant la perte lorsque la probabilité prédite pour la classe correcte est faible.

La ***def loss*** est une mesure de l'erreur liée aux prédictions de la présence ou de l'absence d'objets dans les boîtes englobantes. Cette perte aide le modèle à ajuster ses prédictions pour minimiser les faux positifs (prédire un objet alors qu'il n'y en a pas) et les faux négatifs (ne pas prédire un objet alors qu'il y en a un). Elle est calculée en utilisant la fonction de perte d'entropie croisée binaire, car il s'agit d'un problème de classification binaire.

Soit $\hat{p}$ la probabilité prédite de la présence d'un objet dans une boîte englobante, et $y$ sa réel existence (1 si un objet est présent, 0 sinon). La perte de détection d'objet faux peut être exprimée comme suit :

$$L_{\text{df}} = -\left[y \log(\hat{p}) + (1-y) \log(1-\hat{p})\right]$$

Cette formule pénalise les prédictions incorrectes en augmentant la perte lorsque la probabilité prédite de présence d'un objet est haute. Elle pénalise également les manque de prédictions lorsque la probabilité prédite de présence d'un objet est basse.

La ***precision*** est définie comme le rapport entre le nombre de prédictions correctes et le nombre total de prédictions faites (les correctes et les fausses prédictions). Elle mesure la probabilité que la prédiction du modèle soit correcte.

Soit $TP$ (True Positives) le nombre de prédictions correctes (le modèle prédit un objet et il y a effectivement un objet) et $FP$ (False Positives) le nombre de prédictions fausses (le modèle prédit un objet mais il n'y en a pas). La précision est calculée comme suit :

$$\text{Précision} = \frac{TP}{TP + FP}$$

Le ***recall*** est définie comme le rapport entre le nombre de prédictions correctes et le nombre total de prédictions à faire (tous les endroits où il y a un objet). Il mesure la capacité d'un modèle à bien repérer tous les objets à classifier.

Soit $TP$ (True Positives) le nombre de prédictions correctes (le modèle prédit un objet et il y a effectivement un objet) et $FN$ (False Negatives) le nombre prédictions manquées (le modèle prédit qu'il n'y a pas d'objet alors qu'il y en a un). Le rappel est calculé comme suit :

$$\text{Rappel} = \frac{TP}{TP + FN}$$

La mesure **mAP50** est la moyenne de la précision moyenne (*mean Average Precision* ou mAP) lorsque le seuil d'Intersection over Union (IoU) est fixé à 0.5. En d'autres termes, une prédiction est considérée comme correcte si l'IoU entre la boîte prédite et la boîte réelle est supérieur ou égal à 0.5. La formule pour mAP est donnée par :

$$\text{mAP} = \frac{1}{N} \sum_{i=1}^{N} \text{AP}_i$$

où $N$ est le nombre de classes et $\text{AP}_i$ est la précision moyenne pour la classe $i$.

La mesure **mAP50-95** est la moyenne de la précision moyenne (*mean Average Precision* ou mAP) calculée à plusieurs seuils d'IoU, allant de 0.5 à 0.95 avec un pas de 0.05. Ceci donne une évaluation plus stricte et détaillée de la performance du modèle. La formule pour mAP50-95 est donnée par :

$$\text{mAP50-95} = \frac{1}{10} \sum_{t=0.5}^{0.95} \text{mAP}_t$$

où $\text{mAP}_t$ est la moyenne de la précision moyenne à un seuil d'IoU $t$.

Nous pouvons constater sur les courbes d'entraînement (voir figure 2.9) que les erreurs diminuent continuellement sur les graphiques de type *train*. Cependant, ce n'est pas le cas pour les courbes de type *val*. En effet, vers 70 epochs, les courbes commencent à remonter. Ce phénomène est caractéristique d'une situation d'overfitting, c'est-à-dire que le modèle se calque trop précisément sur les images d'entraînement et perd sa capacité à généraliser à de nouvelles images.
Ainsi, pour obtenir le meilleur modèle possible, nous avons sélectionner les poids du modèle à la fin de la 70ème epochs.

Dans un second temps, analysons la matrice de confusion (voir figure 2.10) réalisée par le modèle constitué des poids après 70 epochs (fichier .pt) sur l'ensemble de validation . La classe "background" indique que le modèle n'a reconnu aucun élément dans l'image. Nous pouvons constater que le modèle a le plus de difficultés avec cette classe. En effet, il reconnaît des fleurs, en particulier les Jatropha, là où il n'y en a pas (comme indiqué par la dernière colonne de la matrice de confusion) et n'arrive pas à les reconnaître là où il y en a (dernière ligne de la matrice). Il faut améliorer sa précision et son rappel. Toutefois, l'erreur sur les Jatropha peut s'expliquer simplement : en analysant la base de données, nous pouvons constater que les Jatropha ne sont pas toujours toutes annotées en raison de leur nombre par image. Ainsi, le modèle reconnait des Jatropha, non pas là où il n'y en a pas, mais là où elles n'ont pas été annotées.
En omettant donc l'erreur sur les Jatropha, le pourcentage d'erreur de classification global est de 10%, ce qui est satisfaisant.

Maintenant, comparons les performances des modèles *.pt* et *.tflite*. En effet, le format *tflite* est plus compact donc prend moins de place mémoire ce qui est essentiel pour la création d'une application mobile mais il perd en performance. Le tableau 2.1 résume la comparaison des performances des deux formats sur l'ensemble de test. *Fitness* est une mesure de performance qui est une combinaison linéaire des précédentes.

| Mesure | Format .pt | Format .tflite | Ecart relatif |
|---|---|---|---|
| Précision (B) | 0.693108143 | 0.663495793 | 4.27239958 |
| Rappel (B) | 0.919848328 | 0.861804509 | 6.310151039 |
| mAP50 (B) | 0.79089023 | 0.755507747 | 4.473754025 |
| mAP50-95 (B) | 0.650985776 | 0.614110448 | 5.664536657 |
| Fitness | 0.664976222 | 0.628250178 | 5.522910788 |

TABLE 2.1 – Tableau comparatif des performances entre formats

Nous pouvons constater que la perte de performance est de 5.52% lors de conversion ce qui est raisonnable et acceptable.

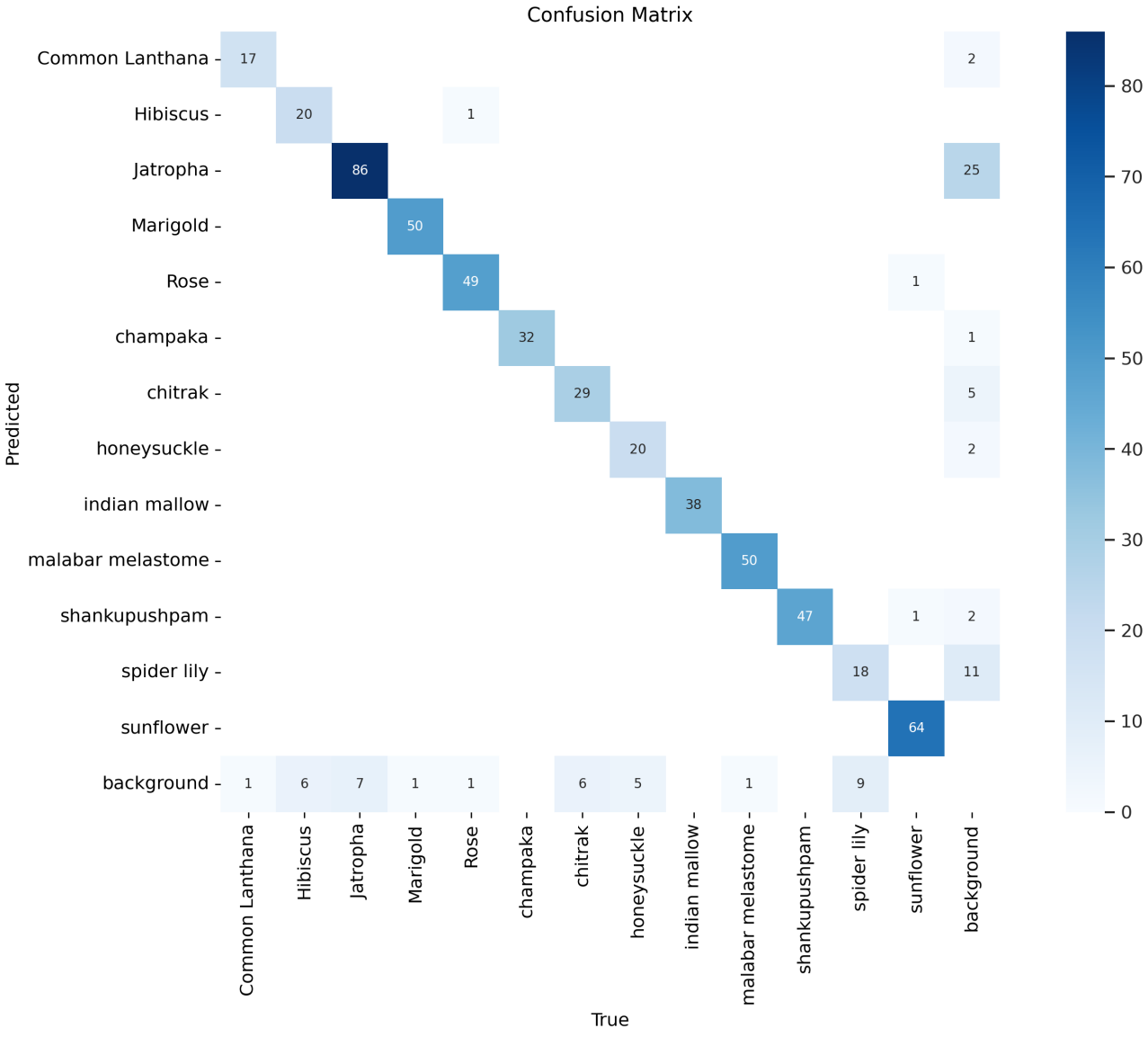


FIGURE 2.10 – Matrice de confusion

# Chapitre 3

# Application Android

L'application consiste en une seule vue, qui reprend le flux vidéo de la caméra arrière, l'analyse à l'aide d'un modèle de détections d'objets et affiche les résultats obtenus sous la forme de boites encadrant les objets avec le type reconnu.

Un menu en bas de l'application permet de choisir entre plusieurs modèles disponibles, pour identifier différents objets, en fonction des classes que le modèle permet de reconnaître. Les modèles disponibles sont ceux de races de chats/chiens et d'espèces de fleurs.

L'application est codée en Kotlin, qui est un langage récent permettant de coder des applications Android. Le langage est néanmoins similaire au Java, et en particulier, les bibliothèques Java sont directement compatibles avec Kotlin. Le code est, par ailleurs, plus concis et permet d'éviter certaines erreurs de type, en particulier avec le type `null`.

## 3.1 Structure de l'application

L'application est composée de quatre classes ainsi qu'une interface :

— `MainActivity` : Elle s'assure du lancement de l'application, de sa fermeture par l'utilisateur et des différentes interactions entre les autres classes de l'application.

— `Detector` : Elle permet de créer un détecteur avec des paramètres à définir lors de l'instanciation. C'est ce qui va permettre d'analyser les images du flux vidéo pour en renvoyer les informations (position et seuil de confiance par exemple) sur les éléments qui sont détectés.

— `Overlay` : Cette classe permet de définir un overlay qui affichera à l'aide d'un canvas des boîtes autour des objets qui sont détectés.

— `DetectorListener` : C'est une interface qui permet d'exécuter des actions en fonction des résultats de la méthode permettant d'analyser les images de la classe Detector.

— `Box` : Cette classe est purement utilitaire et permet de gérer les boites à afficher sous forme d'objet, ce qui simplifie la gestion de leurs paramètres pour les afficher sur l'overlay.

Cette structure est en partie inspirée d'un tutoriel écrit par Aarohi Singla, dont le repo GitHub est disponible en cliquant ici.

### 3.1.1 Schéma des interactions entre les différentes classes

Les relations entre les différentes classes et leur rôle peut être schématisé ainsi :

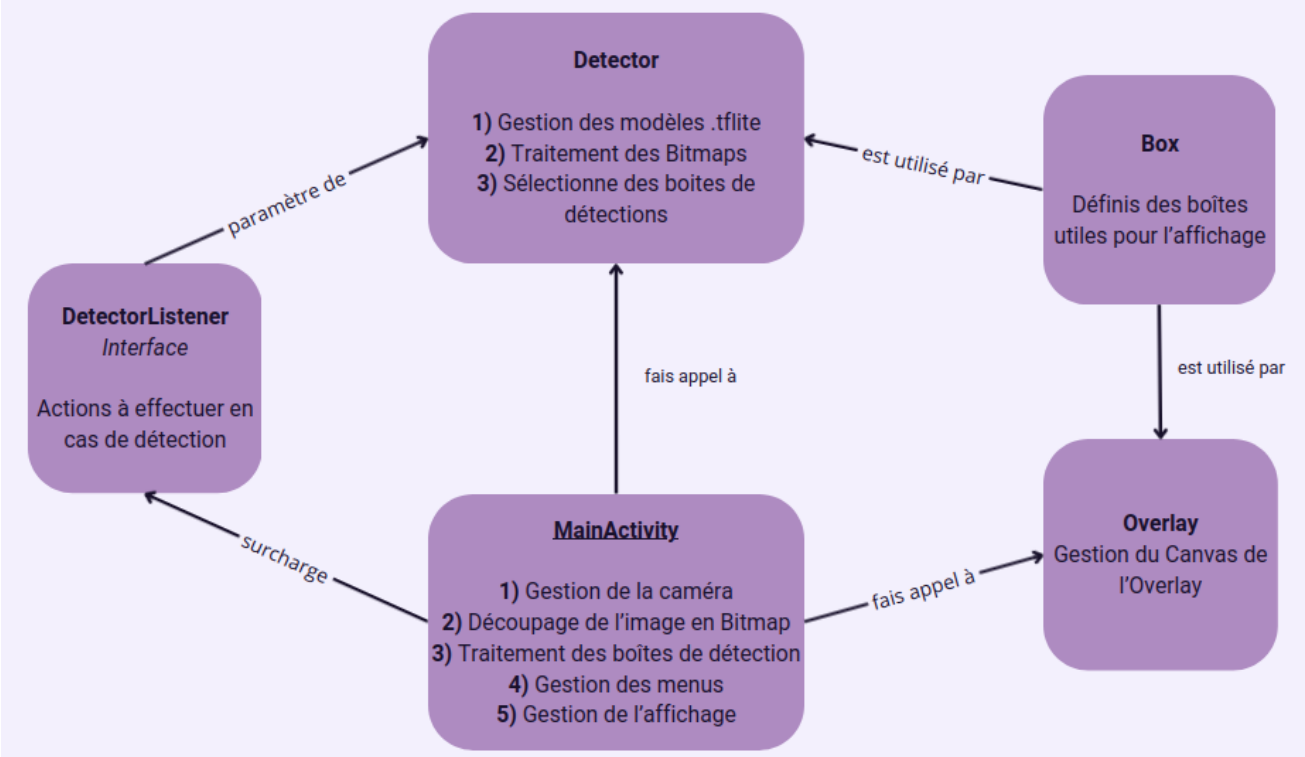


FIGURE 3.1 – Schéma du fonctionnement de l'appli

**Explications** :

Lors du lancement de l'application, l'activité `MainActivity` prend en charge le lancement de la caméra, la gestion des permissions (à la caméra) et relie la sortie du flux vidéo au layout pour avoir un retour vidéo en direct. En parallèle, elle crée une instance de `Detector`, avec comme paramètre un `DetectorListener` et un modèle choisi par défaut.

La classe `MainActivity` étend l'interface `DetectorListener` pour surcharger les deux fonctions définies au sein de l'interface.

Lors de la création de l'instance de Détecteur, la fonction `setup()` est également appelée. Elle permet de charger le modèle utilisé par défaut en s'adaptant aux paramètres du modèle (la taille des images en entrée par exemple). Une fois la caméra correctement configurée, les images capturées sont converties en Bitmap, et ensuite passées en paramètres à la méthode `detect` du détecteur. En sortie, cette fonction renvoie différentes boites, qui sont des instances de la classe `Box`, dont les paramètres permettent d'encadrer les objets qui ont été repérés avec le label associé.

Si le modèle identifie un objet sur une image, la fonction `onDetect` du `DetectorListener` est appelée, qui met à jour l'affichage des boites. La classe `Overlay` étend la classe `View`, ce qui nous permet de l'ajouter directement dans le fichier `activity_main.xml` comme une vue à part entière. Ce qu'elle affiche est un `Canvas` sur lequel sont dessinées les boîtes que le détecteur a sélectionnées.

Pour finir, un menu est géré par l'activité `Main`. Un modèle est chargé par défaut, via deux variables qui contiennent sous forme de `String` les noms du fichier `.tflite` et du fichier texte des labels associés au modèle. Chaque entrée du menu correspond à un type d'objet que l'on souhaite identifier (Chiens/Chats, Fleurs, etc.). Si un clic est effectué sur une des catégories, le modèle est relancé avec comme paramètres les noms des labels et du modèle permettant d'identifier cette catégorie.

### 3.1.2 Gestion des threads

Plusieurs threads sont utilisés dans l'application, avec chacun leur fonction particulière. Cela permet d'avoir un traitement des images plus rapides et limiter les ANR (Application Not Responding), quand le thread principal est bloqué sur une tache en cours.

En tout, sept threads sont lancés par l'application :

— Le thread UI : c'est le thread principal, en charge des interactions entre l'application et l'utilisateur, comme les clics de bouton ou les gestes tactiles.

— Le thread `cameraExecutor` : c'est un thread lancé lors du lancement de l'application (dans le `onCreate` de `MainActivity`) qui convertit les images capturées par la caméra en Bitmap.

— Les quatre threads du modèle : Lors de la mise en place du modèle dans le détecteur, quatre threads sont lancés, un nombre qu'on a pu retrouver dans d'autres exemples intégrant un `tflite`. Ces threads servent exclusivement au traitement effectué par le modèle (qui reçoit en entrée des `TensorImage`, cf. plus bas).

Un septième thread est lancé lors de l'instanciation de `cameraProviderProcess`. Voici son implémentation :

```
val cameraProviderProcess = ProcessCameraProvider.getInstance(this)
    cameraProviderProcess.addListener({
        cameraProvider  = cameraProviderProcess.get()
        bindCamera()
    }, ContextCompat.getMainExecutor(this))
```

L'appel à getInstance() est un processus asynchrone. Si le lancement de ce processus avait lieu sur le thread principal, ce dernier risquerait de se bloquer tant que l'initialisation n'est pas finie. Pour palier ce problème, la fonction getInstance() renvoie en un `ListenableFuture`. Un `Future` est une promesse, qui sera réalisée sur un autre thread, dont la finalité (ici le `cameraProviderProcess`) sera de nouveau accessible au thread principal, une fois que le process s'est correctement initialisé. Pendant ce temps, le thread principal exécute la suite du code.

Ici, nous avons un `ListenableFuture`, ce qui signifie que l'on peut ajouter un `Listener` à notre variable qui va vérifier l'état du Future et va permettre d'exécuter le code entre crochet, dès que le`Future` change d'état. Dans notre exemple, cela signifie simplement qu'on récupère l'instance de ce processus (avec `get()`) une fois qu'il est correctement initialisé.

On peut schématiser les threads au sein de l'application ainsi :

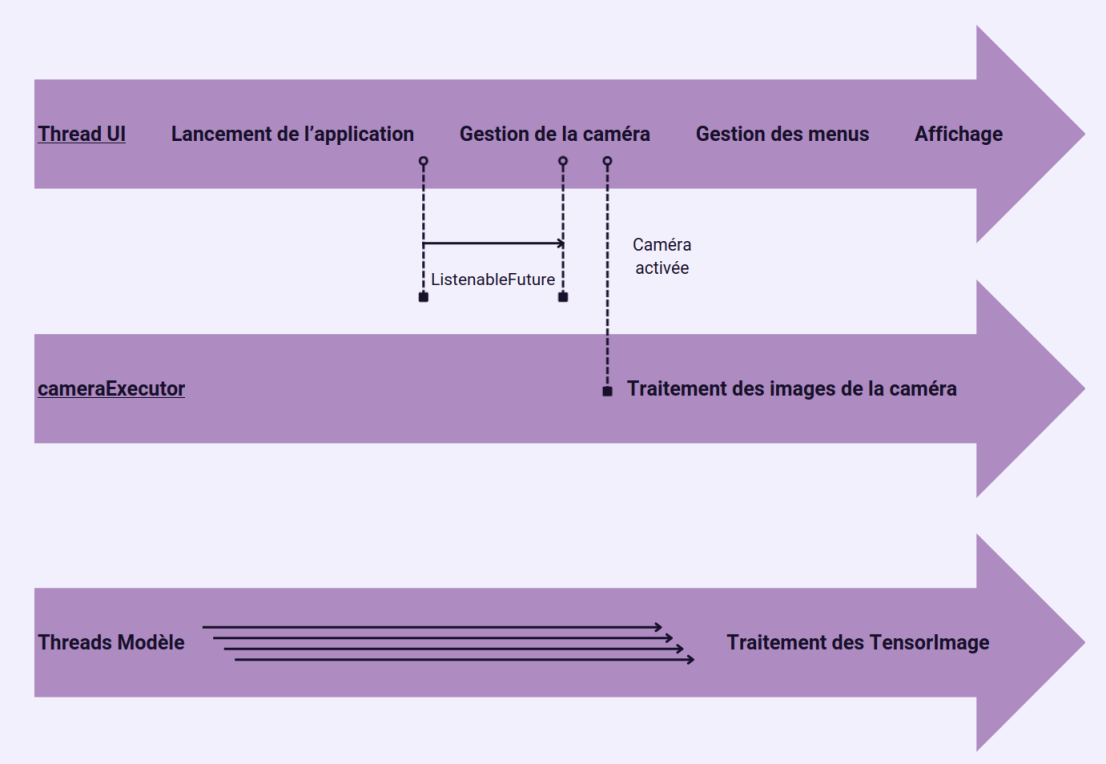


FIGURE 3.2 – Schéma des threads dans l'application

### 3.1.3 Gestion de la caméra

L'activation de la caméra commence par l'obtention du processus `cameraProvider`, comme on l'a montré plus tôt à l'aide d'un `ListenableFuture`.

Il faut ensuite lier les différents éléments qui permettent de traiter les images à ce processus. Le premier est la sélection de la caméra. Cela se fait par un `CameraSelector`, auquel on demande d'accéder à la caméra arrière.

On définit ensuite une `Preview`, qui nous permet d'afficher le flux vidéo après avoir fait le lien entre l'instance et la vue dans le `.xml` de l'activité principale.

Pour finir, on crée une instance de `Camera`, en liant ces deux éléments (`Preview` et `CameraSelector`) au cycle de vie du processus que l'on a créé, ce qui permet que toutes ces actions s'arrêtent lorsque la caméra est désactivée.

### 3.1.4 Gestion des menus

La gestion des menus est assez simple, en particulier, car il existe des layouts et des classes prédéfinies. Il suffit de créer le layout dans un fichier XML, situé dans `/res/menu` et d'importer la classe `PopupMenu`, qui génère un petit menu déroulant.

On place ensuite un `Listener` sur ce menu, qui nous permet d'appeler un code spécifique en fonction du bouton qui a été cliqué. Depuis cet item, on met à jour le nom du fichier `.tflite` et celui des labels pour utiliser le modèle souhaité avant de reconfigurer le détecteur (en rappelant `setup()`).

### 3.1.5 Liaisons entre les affichages

Plutôt que d'utiliser des `findViewById`, dont les erreurs sont détectables uniquement à l'exécution, les liaisons sont effectuées par `ActivityMainBinding`. Cette classe est générée automatiquement, en rajoutant l'option :

```
    buildFeatures {
    viewBinding = true
    mlModelBinding = true
}
```

dans les paramètres du `build.gradle.kts`. Son activation va générer une classe de liaison pour chaque fichier XML. Par exemple, la classe `ActivityMainBinding` est créée automatiquement lorsque ce paramètre est activé pour le fichier XML `activity_main.xml`.

En plus de cela, chaque vue du fichier XML peut être accédée depuis l'id de la `View`. Par exemple, la vue `Overlay` dont l'id est `overlay` peut être accédée par `binding.overlay`. L'intérêt de cet accès est que l'on puisse exécuter les méthodes des classes associées à ces vues. C'est le cas dans ce code par exemple :

```
binding.overlay.apply {
    setResults(boundingBoxes)
    invalidate()
}
```

## 3.2 Intégration du modèle de détection d'objet

### 3.2.1 Ajout et lecture d'un fichier .tflite

Pour ajouter un fichier `.tflite`, il faut le placer dans le dossier `/app/src/main/res/assets`, avec le fichier texte des labels. Chaque ligne du fichier texte correspond à un élément que le modèle va pouvoir reconnaître. Il est nécessaire que le fichier `labels.txt`, utilisé dans l'appli soit le même que celui utilisé lors de l'entraînement du modèle.

Une fois les fichiers correctement placés, il existe plusieurs classes de TensorFlow pour les traiter. Parmi celles-ci, la classe `FileUtile` permet de charger un modèle `.tflite`. On peut ensuite définir l'ensemble des paramètres à la main, mais on peut profiter des métadonnées du fichier pour récupérer les paramètres du modèle directement. Les paramètres sont les suivants :

— `tensorWidth` : la longueur des images que le modèle peut traiter ;

— `tensorHeight` : la hauteur des images que le modèle peut traiter ;

— `numChannel` : le nombre de classes que le modèle peut détecter ;

— `numElements` : le nombre d'éléments que le modèle peut détecter en simultané sur une image.

Il suffit enfin de lire chaque ligne du fichier texte des labels et les enregistrer dans l'ordre dans un vecteur de `String`.

### 3.2.2 Transformation du flux vidéo

Comme les modèles que nous utilisons ne font que du traitement sur les images et non sur les vidéos, il est nécessaire de segmenter le flux de la caméra en image pour les données en paramètres d'entrée au détecteur.

Cette segmentation se fait sur un thread séparé, à l'aide d'un `imageAnalyzer`, qui permet de renvoyer sous forme de Bitmap chaque image. Comme l'exécution a lieu sur un thread à part et que l'image doit encore être utilisée par la suite, on utilise un `imageProxy` qui libère l'image en cours de traitement dès que ce dernier est fini.

On applique ensuite une rotation de l'image pour s'assurer que l'image est bien droite et limiter les erreurs de reconnaissance dues à une orientation du téléphone. Ce BitMap modifié est ensuite envoyé à l'instance du détecteur.

Le BitMap ne peut pas être donné en argument d'entrée tel quel au modèle, il faut d'abord le redimensionner en accord avec le paramètre du modèle (pour rappel, on récupère ces paramètres lors du chargement du fichier). Elle est ensuite convertie en une `TensorImage`, qui encapsule l'image dans un tenseur, qui sera ensuite modifié par un `imageProcessor`, afin de normaliser les pixels de l'image. Cela permet d'avoir une meilleure performance du modèle et de vérifier que le bon type de données a été utilisé. La conversion en `TensorImage` se fait en effet selon une catégorie de données bien précises, comme `Float32`, qui est une représentation des nombres à virgules sur 32 bits (4 octet).

Après l'ensemble de ces opérations, le `TensorImage` normalisé est donné comme paramètre d'entrée au modèle.

### 3.2.3 Sélection des résultats du modèle

Le modèle renvoie un seul vecteur, dans lequel sont stockés l'ensemble des résultats, avec :

— Les coordonnées des objets qui ont été repérés, correspondant au centre des boites de détection.

— Les taux de confiance de chaque classe pour cet élément détecté. Plus ce taux est haut, plus le modèle identifie cet objet à une classe donnée.

Il faut alors parcourir correctement (avec les bons indices) ce vecteur. Pour chaque élément, on extrait le meilleur taux de confiance. Si ce dernier est suffisamment élevé, on crée alors une boite (une instance de la classe `Box`) avec les coordonnées de cet objet.

Dans un souci de lisibilité, on ne souhaite pas que des boîtes se superposent. Dès lors, on trie l'ensemble des boîtes qui ont été retenues selon le taux de confiance pour s'assurer que les premières boîtes (qui sont donc plus fiables) ne s'intersectent pas avec les autres boîtes. La possibilité d'une intersection est calculée par `calculateIoU` (IoU pour Intersection over Union).

### 3.2.4 Affichage

Une fois les boîtes sélectionnées et placées dans un vecteur, il ne reste plus qu'à les afficher. Pour cela, un Overlay custom est utilisé. Cette classe est définie de telle manière à ce qu'à chaque instance créée, une méthode est appelée, mettant à jour les éléments dessinés.

Comme l'overlay étend `View`, nous pouvons y faire appel dans le fichier `main_activity.xml`, sous la forme d'une vue personnalisée. Cela nous permet d'y accéder via une instance d'`ActivityMainBinding`, que nous avons présenté plus haut. Dès lors, en définissant comme paramètres de classe une liste de boites à afficher dans Overlay, il suffit de faire appel au `seter` de ce paramètre avec la syntaxe `binding.overlay.apply{myFonction}` pour mettre à jour l'affichage quand on le souhaite. Cette mise à jour se fait dès qu'un nouvel élément est détecté.

# Chapitre 4

# Conclusion

## 4.1 Résultat et valeur ajoutée

Nous sommes très fiers du résultat final. Nous avons réussi à générer un modèle de détection d'objet performant entraîné sur une base de données personnalisée. Nous avons réussi à convertir le modèle en *tflite* malgré les problèmes d'incompatibilité des dépendances dépassées. Nous avons également réussi l'intégration du modèle dans l'application. De plus, nous avons pu ajouter une fonctionnalité supplémentaire qui permet de changer de modèle de détection et ainsi de reconnaître plusieurs types d'objets.

## 4.2 Perspectives

Si nous avions eu plus de temps, nous aurions pu rajouter des fonctionnalités à l'application, comme la possibilité d'enregistrer des détections, de changer le seuil de confiance de détection ainsi que d'autres paramètres. Pour ce qui concerne la génération du modèle de détection, avec plus de ressources, nous aurions pu entraîner notre modèle sur des bases de données plus diverses et ainsi combler les lacunes de notre modèle actuel.