

Application mobile pour l'identification du frelon asiatique

Travail de Bachelor

Non Confidentiel

Département : TIC

Filière : Informatique et systèmes de communication

Orientation : Informatique logicielle

Chollet Bastian

23 septembre 2024

Supervisé par :

Dutoit Fabien

Préambule

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d'auteur.

HEIG-VD

Le Chef du Département

Yverdon-les-Bains, le 17 juin 2024

Authentification

Le soussigné, Chollet Bastian, atteste par la présente avoir réalisé seul ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées.

Chollet Bastian

Yverdon-les-Bains, le 17 juin 2024

Résumé

Le frelon asiatique (*Vespa Velutina*) a été introduit accidentellement en France en 2004 et a atteint la Suisse ces dernières années. Les autorités tentent de lutter contre cette espèce qui se nourrit notamment d'abeilles mellifères, représentant ainsi un fléau pour les apiculteurs.

L'objectif de ce travail est de proposer une solution numérique portable cross-plateforme permettant d'identifier cette espèce à l'aide d'un cliché réalisé depuis la caméra d'un téléphone portable.

Cet énoncé a servi de prétexte afin d'étendre cette solution à un usage plus générique en développant une librairie permettant de créer des applications cross-plateformes simples de classification d'images. L'objectif de ce travail reposant d'avantage sur la réalisation de cette librairie et non pas sur l'obtention d'un modèle performant de classification du frelon asiatique.

Ce travail a été réalisé en plusieurs phases distinctes allant de la réalisation d'un modèle de classification d'image à l'aide de *TensorFlow* et *Keras* jusqu'à son intégration dans une application *Flutter*.

Premièrement, nous avons réalisé une analyse approfondie des technologies et solutions disponibles dans l'objectif d'établir un état de l'art de la situation actuelle.

En deuxième temps, nous avons réalisé un modèle fonctionnel de classification de *Vespa Velutina* parmi d'autres espèces d'insectes nuisibles. Ce modèle a par la suite été intégré dans une application *Flutter* de démonstration.

Finalement, cette application utilise une librairie de classification d'image que nous avons réalisée pour l'occasion. Elle fait appel à d'autres solutions existantes et propose une abstraction de ces dernières.

Les résultats obtenus sont encourageants et incitent à approfondir ce travail en proposant une solution universelle indépendante du langage cross-plateforme utilisé.

Table des matières

Préambule	I
Authentification.....	II
Résumé.....	III
Chapitre 1 Introduction	1
1.1 Contexte	1
1.2 Cible	1
1.3 Identification des besoins.....	1
1.3.1 Besoins fonctionnels.....	2
1.3.2 Besoins non-fonctionnels	2
1.4 Objectif	2
1.5 Fonctionnalités	2
1.5.1 Fonctionnalités principales.....	2
1.5.2 Fonctionnalités optionnelles	3
1.6 Planification	3
1.7 Organisation	4
Chapitre 2 Recherche et état de l'art	5
2.1 Modèles deep learning.....	5
2.1.1 État de l'art	6
2.1.2 Point de situation sur la recherche	8
2.1.3 Solution choisie	8
2.2 Datasets.....	9
2.3 Architectures de réseau de neurones	10
2.3.1 Architectures proposées	11
2.3.2 Résultats obtenus	12
2.3.3 Solution choisie	13
2.4 Intégration de modèle dans une application cross platform	14
2.4.1 Création du modèle.....	14
2.4.2 Export du modèle	14
2.4.3 Solution cross-platform	16
2.4.4 Inférence du modèle sur mobile	18
2.5 Modélisation et architecture d'un prototype	18
2.5.1 Réalisation du modèle.....	18
2.5.2 Exportation du modèle.....	19
2.5.3 Réalisation de l'application	19
2.5.4 Import et inférence du modèle	19

Chapitre 3 Implémentation	22
3.1 Architecture.....	22
3.2 Entraînement du modèle	24
3.2.1 Dataset complémentaire.....	24
3.2.2 Adaptations des tailles des datasets	24
3.2.3 Visualisation des données	26
3.2.4 Pré-processing.....	26
3.2.5 Entraînement du modèle	27
3.3 Implémentation de l'application de démonstration	28
3.4 Implémentation du <i>Dart Package</i>	30
3.4.1 Contexte	30
3.4.2 Implémentation des interfaces	31
3.4.3 Implémentation des fonctions internes	34
Chapitre 4 Résultats obtenus	42
4.1 Validation du modèle	42
4.2 Mesure de performance du <i>Dart Package</i>	45
4.2.1 Méthodologie de test.....	45
4.2.2 Espace mémoire	45
4.2.3 Vitesse d'inférence	46
4.2.4 Retours sur notre implémentation.....	48
Chapitre 5 Axes d'amélioration.....	50
5.1 Modèle de classification du frelon asiatique.....	50
5.1.1 Dataset du frelon asiatique	50
5.1.2 Dataset complémentaire.....	51
5.1.3 Précision relative du modèle	52
5.2 <i>Dart Package</i> pour la classification d'images	52
5.2.1 Interprétation des headers du fichier tflite	52
5.2.2 Traitement de différents canaux de couleurs	53
5.2.3 Solution universelle	53
Chapitre 6 Conclusion.....	54
Bibliographie.....	55
Annexes	57

Table des figures

Figure 1 – Diagramme de Gantt de l'organisation générale du projet.....	4
Figure 2 – Types de problèmes traités par la vision par ordinateur ^[17]	6
Figure 3 – Carte de points chauds d'un modèle de classification ^[16]	7
Figure 4 – Fonctionnement de l'apprentissage par transfert ^[9]	10
Figure 5 – Étapes de convolutions de ShuffleNet ^[13]	12
Figure 6 – Espace mémoire et consommation d'énergie lors d'analyse de 1'000 images ^[18]	12
Figure 7 – Écarts quadratiques moyen par rapport au temps d'inférence ^[18]	13
Figure 8 – Librairies ONNX de flutter.....	16
Figure 9 – Résultat du prototype Android après inférence d'une image.....	21
Figure 10 - Architecture globale	23
Figure 11 - Échantillons du dataset du frelon asiatique	25
Figure 12 - Visualition des données dans le dataframe du notebook python	26
Figure 13 - Application de démonstration sous Android	29
Figure 14 - Liste des tâches supportées par MediaPipe Flutter ²⁴	31
Figure 15 - Convention d'arborescence pour un Dart package.....	32
Figure 16 - Diagramme de classe de la partie exposée du Dart package.....	33
Figure 17 - Représentation de l'event loop de Dart ^[5]	35
Figure 18 - Schéma des Isolates en Dart ^[5]	36
Figure 19 - Diagramme de classe des fonctions internes.....	39
Figure 20 - valeur de la fonction de coût au fil des epochs de l'apprentissage	42
Figure 21 - Matrice de confusion du modèle sur le set de test.....	43
Figure 22 - Visualisation des données de tests avec leurs prédictions	44
Figure 23 - Heatmap des données de tests avec leur prédiction	44
Figure 24 - Heatmap d'une version antérieure du modèle.....	44
Figure 25 - Snapshot de consommation de mémoire de l'appareil lors d'une inférence	46
Figure 26 - Temps d'exécution moyen du modèle séparé par inférence et normalisation	46
Figure 27 - Comparatifs des augmentations de performance en fonction des options sélectionnées par rapport au cas de référence	47
Figure 28 - Temps d'inférence moyen selon le nombre de threads utilisé	48
Figure 29 - Image du set d'entraînement.....	51
Figure 30 - Image issue du set d'entraînement.....	52
Figure 31 - Schéma des donnée d'un fichier .tflite ^[11]	53

Liste des tableaux

Tableau 1 - Répartition des classes dans le premier dataset étudié	9
Tableau 2 - Nombre de paramètres à entraîner selon l'architecture ^[2]	13
Tableau 3 - Nombre d'images par ensemble des datasets utilisés	25
Tableau 4 - Nombre d'image de chaque dataset après filtrage	26
Tableau 5 - Score du modèle sur le set de test par classe prédictive	43

Liste des codes sources

Code 1 - Paramètres appliqués pour l'augmentation de données.....	27
Code 2 - Instanciation du modèle pré-entraîné	27
Code 3 - Snippet d'utilisation du package tflite_flutter.....	30
Code 4 - Exemple d'Isolate avec closure	37
Code 5 - Utilisation des isolates de tflite_flutter.....	37
Code 6 - Méthodes de normalisation par pixel	40
Code 7 - Association prédiction - label dans un cas binaire	41

Chapitre 1

Introduction

1.1 Contexte

Le frelon asiatique (*Vespa Velutina*) a été introduit accidentellement en France en 2004 et se répand depuis en Europe. Cette espèce invasive, dont la principale source de nourriture est les abeilles, est combattue par les autorités suisses depuis qu'elle a été identifiée pour la première fois à Genève en 2020. Aujourd'hui elle a colonisé le pied du Jura jusque dans la région bâloise.

Les rencontres avec des frelons, même européens (*Vespa Crabro*), restent exceptionnelle pour la majorité de la population. De ce fait, il n'est pas toujours évident d'identifier correctement l'espèce exacte rencontrée. D'autant plus que le caractère et la taille impressionnante de l'insecte suscite aisément la peur.

En ajoutant que les espèces endémiques doivent être conservées si celles-ci ne présentent aucune menace directe, le travail des autorités peut vite être ralenti si ces dernières interviennent sur des fausses alertes remontées par des citoyens n'ayant aucune connaissance sur l'apparence de l'espèce à éradiquer.

C'est dans ce contexte que nous souhaitons proposer une solution mobile cross-plateforme permettant l'identification de l'espèce nuisible à l'aide de la caméra du téléphone. L'idée principale étant de pouvoir réaliser cette identification à l'aide d'un modèle d'intelligence artificielle (machine learning) embarqué directement sur le téléphone. Une fois l'espèce nuisible identifiée, il sera possible de participer à la localisation du nid en fournissant les images capturées ainsi que les coordonnées géographiques ou les trajectoires de vols.

1.2 Cible

Cette application s'adresse à un large public que celui-ci soit familier avec la technologie ou non. Il s'adresse à tout possesseur de smartphone équipé d'une caméra peu importe le fabricant de ce dernier. Cette solution trouvera son utilité auprès de personnes vivant sur le plateau helvétique ainsi que dans le jura, là où l'espèce a été majoritairement observée. Elle permet à tout un chacun de participer activement dans la lutte contre le frelon asiatique en s'assurant de ne pas fournir d'informations erronées en marquant des espèces endémiques.

L'aspect communautaire en partageant les données de localisations, de trajectoire de vol ainsi que les images capturées seront utiles à toutes les autorités compétentes participant à la destruction des nids. Elles permettent également aux biologistes de mieux observer le territoire occupé par l'espèce ainsi que l'évolution de ce dernier.

Les apiculteurs ou toutes personnes aptes à capturer l'insecte de façon sécurisée pourra également introduire les trajectoires de vols de l'insecte une fois celui-ci relâché. Ces données étant d'autant plus précieuses puisqu'elles permettent une triangulation du nid.

1.3 Identification des besoins

Comme il a été présenté plus haut, l'application se doit de proposer plusieurs fonctionnalités afin de couvrir les besoins énoncés selon le contexte, ainsi que des besoins non-fonctionnel devant s'adapter au public cible.

1.3.1 Besoins fonctionnels

- **Accès à l'appareil photo** : L'application doit pouvoir accéder à la caméra du téléphone pour prendre des photos et utiliser les captures obtenues dans l'application.
- **Identification du frelon sur image** : Chaque image capturée doit être analysée et identifiera la présence ou non d'un objet défini en amont par l'application (dans notre cas, le frelon asiatique).
- **Insertion des coordonnées géographiques** : Les utilisateurs doivent disposer d'une solution simple pour indiquer les coordonnées géographiques du lieu de capture de l'image.
- **Tracés sur une carte** : L'application doit permettre à l'utilisateur de saisir des trajectoires de vol sur une carte.
- **Envoi de données à un serveur** : Si le résultat de l'identification s'avère être l'espèce recherchée, l'utilisateur pourra fournir les coordonnées géographiques ainsi que les images capturées à un serveur.

1.3.2 Besoins non-fonctionnels

- **Cross-platform** : L'application se doit d'être fonctionnelle que celle-ci soit lancé depuis un appareil Android ou iOS. Ceci doit être transparent pour l'utilisateur.
- **Précision du modèle** : La détection doit être suffisamment précise et ce même sur des images de faibles résolutions ou avec un sujet de petite taille afin que les utilisateurs ne se mettent pas en danger lors de la capture d'images.
- **UI/UX** : L'application doit être simple d'utilisation afin que le plus large public puisse l'utiliser sans difficultés particulières.

1.4 Objectif

L'objectif principal de ce travail de bachelor est d'explorer la faisabilité d'embarquer un modèle deep learning de reconnaissance d'objets ou de classification d'images dans une solution cross-plateforme. Le défi étant que l'exécution du modèle ne doit pas être déléguée à un serveur, mais réalisée directement sur le smartphone.

Le contexte du projet étant très précis, il est possible que certaines limitations se présentent, notamment concernant la recherche d'un dataset d'images de frelons asiatiques suffisamment fourni et annoté correctement, nécessaire à la construction d'un modèle. Dans l'hypothèse où une telle limitation venait à se présenter, nous préférerons contourner les contraintes du contexte initial en choisissant d'autres datasets, sur d'autres cas d'utilisation, afin de mener à terme l'objectif principal.

1.5 Fonctionnalités

Nous avons distingué les besoins en fonctionnalités principales, devant obligatoirement être incluses dans le livrable final, et en fonctionnalités optionnelles qui seront implémentées si le temps le permet.

1.5.1 Fonctionnalités principales

- **Photographier** : L'application accèdera à la caméra du smartphone et permettra la capture d'images directement depuis l'application.
- **Analyser l'image** : Une fois la capture réalisée, l'utilisateur pourra la soumettre à une analyse par un modèle de deep learning embarqué qui indiquera si l'image contient bel et bien un frelon asiatique dans le cas où le contexte initial aura pu être respecté.

Si aucune limitation n'a été rencontrée, et que le temps alloué à l'intégration d'un modèle deep learning cross-plateforme aura pris moins de temps qu'initialement prévu, les fonctionnalités suivantes pourront être intégrées :

- **Se localiser** : Afin de faciliter l'entrée de la position où le sujet a été identifié, l'application mettra en place un service de géolocalisation afin de connaître sa position.
- **Envoi des coordonnées géographiques** : L'utilisateur pourra indiquer sur une carte la position relativement exacte de l'endroit où a été identifié le sujet en vue de transmettre les coordonnées géographiques à un serveur distant.
- **Envoi d'images** : Si l'utilisateur le souhaite, il pourra ajouter les images saisies en plus des coordonnées géographiques. Celle-ci devront être encodée pour pouvoir être transmise au serveur distant.

1.5.2 *Fonctionnalités optionnelles*

- **Sauvegarde locale** : Un système de sauvegarde local permettra à l'utilisateur de reprendre une saisie interrompue de façon volontaire ou involontaire.
- **Tracé de trajectoire de vol** : Si l'utilisateur souhaite réaliser une triangulation du nid en ayant capturé des spécimens identifiés en amont, il pourra indiquer sur une carte la direction du vol du sujet à l'aide d'un glissement de doigt.
- **Envoi des trajectoires de vol** : De la même façon que l'utilisateur transmet des coordonnées géographiques, il pourra transmettre les trajectoires indiquées
- **Consultation de l'historique des données** : Le serveur distant stockera les données reçues et il sera possible de consulter ces données à tout moment.

1.6 Planification

Ce rapport sera divisé en plusieurs sections retracant le parcours réflexif effectué ainsi que les implantations réalisées. En premier lieu, un état de l'art sera réalisé afin d'identifier les technologies disponibles en termes de classifications d'images, de langages de programmation *cross-platform*, et de bibliothèques permettant l'intégration de modèle deep learning sur mobile. Nous pourrons ainsi situer le projet sur l'existant tout en relevant les points nécessitant des approfondissements.

Par la suite de nos observations, nous entraînerons un modèle de classification d'image en utilisant les outils analysés à notre avantage. Nous mettrons en évidence le déroulé de l'implémentation, les obstacles rencontrés et comment ceux-ci ont été surmontés.

Une fois le modèle à notre disposition, nous documenterons l'implémentation de la solution permettant son intégration dans une application mobile de démonstration. Il s'agira ici de détailler les différents composants de l'application en mettant l'accent sur les points-clés comme les différents packages installés et leurs utilisations.

Un point de situation servira de conclusion à ce rapport, faisant état du projet à la fin du temps imparti. Nous parlerons des éventuels retards et de leurs impacts sur le résultat final. Nous comparerons l'état actuel du projet avec les objectifs initiaux et nous finirons par les différents axes d'améliorations qu'il serait possible d'exploiter pour porter ce projet vers d'autres horizons.

1.7 Organisation

L'organisation globale peut être consultée via le diagramme de Gantt ci-dessous.

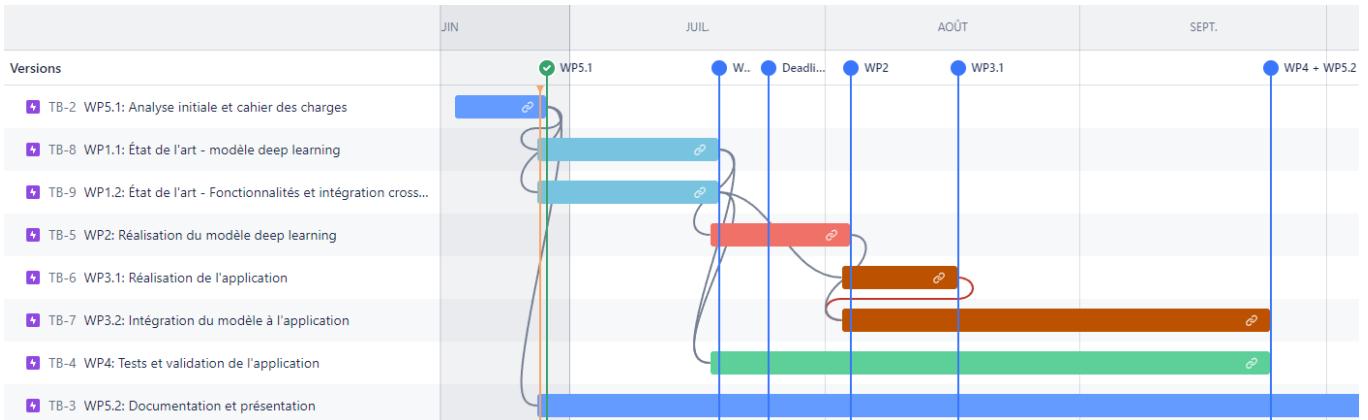


Figure 1 – Diagramme de Gantt de l'organisation générale du projet

Chapitre 2

Recherche et état de l'art

Ce chapitre contient l'état de l'art des technologies disponibles et utiles à ce travail de Bachelor. Il est divisé en plusieurs sections, chacune traitant un bloc composant l'application.

Pour rappel, nous parlerons ici des différents types de modèle deep learning se prêtant au mieux à l'identification du frelon asiatique, puis nous réaliserons une synthèse des différents jeux de données trouvés en exposant leurs forces et limitations. Une fois ces deux éléments choisis, nous pourrons statuer sur une architecture pré-entraînée adaptée à une utilisation sur plateforme mobile.

Ensuite, nous nous concentrerons sur la recherche de solutions existantes ou non permettant la réalisation de l'intégration d'un modèle deep learning entraîné à une application *cross-platform*. Dans un premier temps, cette section sera indépendante du modèle deep learning choisi puisque nous sommes, dans l'idéal, à la recherche d'une solution faisant abstraction de la couche machine learning. La solution ne devrait pas être couplée au modèle exporté qu'on lui fournit.

Finalement, une analyse architecturale mettra les éléments retenus jusqu'ici en commun et explicitera leurs intégrations dans l'application prototype finale.

2.1 Modèles deep learning

Plusieurs fois au cours de ce rapport, nous avons fait mention de modèle « deep learning ». Notons ici que le terme est générique et ne définit pas un modèle à proprement parlé. Il s'agit plutôt d'une famille d'apprentissages automatiques fondée sur l'apprentissage de représentations de données. Dit autrement, le terme définit une technique utilisée par un ensemble de modèles comme l'extraction de caractère dans une image, dans un son, une vidéo, etc... Toutefois, par souci de simplicité, nous utiliserons ce terme pour parler de modèles de machine learning traitant des images au travers d'un réseau neuronal convolutif.

Le réseau de neurones convolutifs (CNN¹) sont un élément fondamental pour la vision par ordinateur. Depuis son invention en 1988 par Kunihiko Fukushima, il a été largement amélioré et agrémenté de nouvelles possibilités au cours des 12 dernières années avec l'augmentation de puissance des processeurs graphiques (GPU) et la démocratisation de l'intelligence artificielle générative. Ils ont donc la possibilité de répondre à diverses tâches comme la classification d'images, la détection d'objet et la segmentation d'images. Nous allons développer ces trois dernières tâches car elles représentent des solutions pertinentes à notre situation.

¹ *Convolutional neural networks*

Computer Vision Problem Types

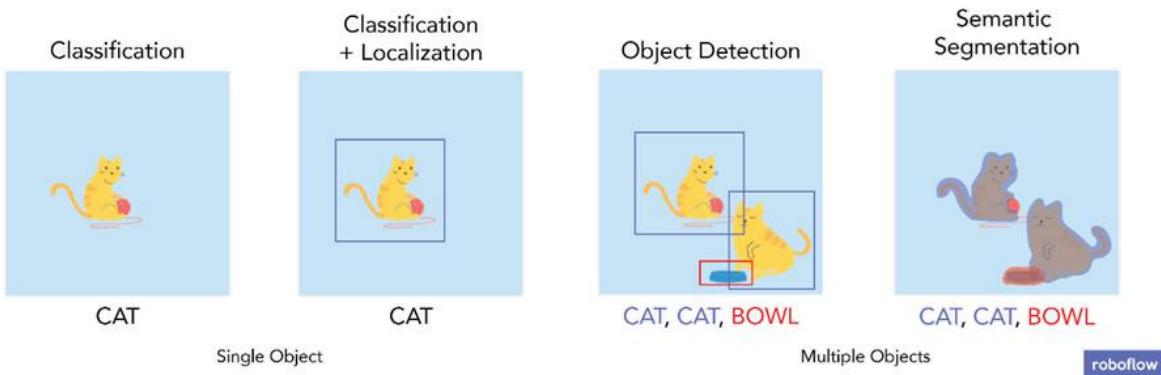


Figure 2 – Types de problèmes traités par la vision par ordinateur^[17].

La vision par ordinateur permet la résolution de plusieurs types de tâches comme la classification, la détection d'objet ou la segmentation sémantique

2.1.1 État de l'art

Certaines techniques ont été volontairement omises dans ce rapport car jugées trop complexes pour les besoins de ce travail. C'est le cas notamment de la segmentation d'image, représentée à droite dans la **Figure 2**, ou de la détection de points clés (technique ayant pour but d'identifier certains points importants d'un objet dans une image, comme les jambes et bras d'une personne).

2.1.1.1 Classification d'images

Nous entendons par classification d'image, le processus permettant de catégoriser l'appartenance d'une image à une classe parmi celles d'un ensemble prédéfini. Plus particulièrement dans cette tâche, il s'agit de porter un regard sur l'ensemble de l'image sans spécifier au modèle les régions d'intérêts permettant de prédire la classe correctement. Par conséquent, et comme le démontre la 1^e image de gauche de la **Figure 2**, cette technique applique une seule classe par entrée.

2.1.1.1.1 Avantages

Ce type de modèle constitue une base qui sera présente dans tous les autres que nous traiterons ici. De ce fait, l'architecture de ces modèles seront souvent moins complexes que les autres tâches ce qui pourrait avoir de nombreux avantages bénéfiques sur une plateforme mobile.

Nous pouvons, par exemple, penser à l'espace mémoire moindre occupé par de tels modèles, mais aussi à des temps d'exécution plus courts ou à des besoins en ressources plus faible, ce qui peut s'avérer critique pour des modèles de smartphone d'entrée de gamme.

L'entraînement de tels modèles pourraient également s'avérer plus rapide, bien que pour ce travail, il ne s'agisse pas vraiment d'un avantage à considérer étant donné que l'entraînement sera réalisé sur un ordinateur disposant de d'avantages de ressources.

2.1.1.1.2 Limitations

Les limitations que présentent un classificateur d'images sont essentiellement liées à la précision de ce dernier. Puisque nous n'indiquons pas la région d'intérêt dans l'image permettant de déterminer sa classe, nous perdons un certain contrôle sur le fonctionnement même du modèle. Ce dernier pourrait, par exemple, déterminer la classe « frelon asiatique » en se basant sur l'arrière-plan de l'image car dans la majorité des cas, les photos de l'insecte ont lieu dans un décors naturel, résultant ainsi par des faux négatifs lorsque l'image reçue présente une scène en intérieur. La **Figure 3** illustre un cas similaire

où un modèle apprend la classe « cheval » non pas en détectant l'animal, mais en détectant le watermark présent sur la photo. Ainsi, placer ce watermark sur n'importe quelle autre image aura pour effet que le modèle labellise cette image comme étant un cheval.

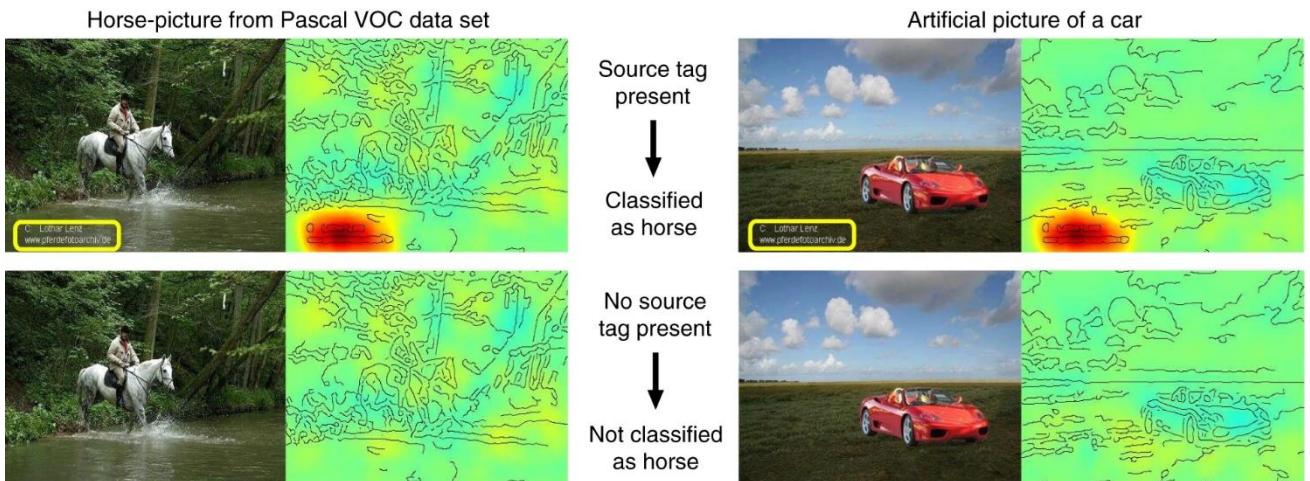


Figure 3 – Carte de points chauds d'un modèle de classification^[16]

En h. à g. le modèle identifie la classe "cheval" non pas grâce à la présence de l'animal, mais grâce au watermark. La présence de ce dernier dans une image sans cheval se verra tout de même attribuer la classe éponyme

L'autre défaut que présente les classificateurs sont leur difficulté à identifier correctement une classe si le sujet de l'image est trop petit ou occulté. Il s'agit d'une conséquence directe au fait que le modèle utilise l'image dans sa totalité pour déterminer une classe. Puisque, dans notre cas, nous nous intéressons à des sujets de petites tailles et dont les photographies risquent d'être prises à des points de vue distants, cela pourrait s'avérer être une limitation importante à considérer, d'autant plus si on ajoute à cela les faibles zooms des appareils mobiles et la résolution basse des appareils d'entrée et milieu de gamme.

2.1.1.2 Détection d'objets

La détection d'objets est un procédé disposant de nombreux cas d'utilisations comme le pilotage de véhicules autonomes ou la surveillance par vidéo. Comme le montre la 3^e image de la **Figure 2**, il ne s'agit plus de classifier une image dans son ensemble, mais d'identifier une région dans celle-ci et de lui attribuer une classe (voir, dans certains cas, une probabilité d'appartenance à cette dernière).

Ce procédé ouvre la porte à la détection de multiples objets au sein d'une même image, chaque objet pouvant être attribué à une classe distincte.

Par le passé, ce procédé nécessitait l'utilisation d'un CNN pour établir des régions de proposition dans l'image puis un autre réseau pour établir une classification pour chaque région. Ces dernières années, d'autres architectures ont vu le jour permettant d'accélérer ce processus et permettant la sélection de régions et la classification en une seule étape.

2.1.1.2.1 Avantages

Les résultats des détecteurs d'objets sont plus facilement interprétables une fois l'image analysée par le modèle. Plutôt que d'obtenir une simple classe, nous disposerons également d'une région où se trouve l'élément identifié, sa taille et également la probabilité d'appartenance à la classe. Cela peut s'avérer être avantageux afin de mieux identifier ce que le modèle voit. Cette caractéristique s'avère fort utile si on dispose d'un cliché d'une ruche d'abeille attaquée par un frelon. Le modèle pourrait mettre en avant l'insecte dans l'image.

Il est plus facile de guider le modèle sur les caractéristiques de l'objet à identifier via le positionnement de l'objet dans l'image. On réduit ainsi l'activation neuronale des zones non pertinentes pour la classification.

Par extension, les détecteurs d'objets apprennent mieux des images où le sujet est occulté, difforme ou de petite taille puisque nous aiguillons la zone à analyser limitant les comportements imprévisibles qui pouvaient naître des classificateurs.

Comme mentionné précédemment, les images à analyser pourront certainement inclure des insectes de petites tailles, en mouvement (donc partiellement flou) ou pris en photo de loin. La détection d'objets pourrait s'avérer plus performante pour identifier un frelon asiatique dans ces scénarios.

La détection d'objet offre la possibilité de classifier plusieurs éléments au sein d'une même image. Ce qui n'est pas possible dans un classificateur.

2.1.1.2.2 Limitations

Les architectures pré-entraînées de détection d'objet peuvent présenter de grandes différences les unes avec les autres ce qui aura, par conséquence, de fortes variations dans les performances et les ressources nécessaires. Cela est également vrai pour le temps d'apprentissage et d'exécution.

Dans la globalité, ces modèles prennent plus d'espace mémoire et consomment d'avantages de ressources qu'un simple classificateur. Ces ressources étant de facto plus faibles sur mobile, certaines architectures pourraient mal fonctionner ou ne pas fonctionner du tout. Le temps d'exécution pourrait également s'en trouver augmenté.

La réelle nécessité de détecter la position de l'objet est discutable. Même si elle apporte de précieuses informations quant au fonctionnement du modèle et offre d'avantages de scénarios d'utilisation, les contraintes que le modèle peut imposer à un téléphone mobile peuvent péjorer le choix de cette solution. En plus de cela, la détection d'objets s'utilise davantage dans des contextes vidéo où il y a une nécessité de suivre la position d'un objet au cours du temps.

2.1.2 Point de situation sur la recherche

Avant d'orienter notre choix vers une solution adaptée à notre problématique, il est important de souligner le fait que la recherche actuelle sur l'intégration de modèles deep learning sur téléphones mobiles n'est pas un sujet traité de façon exhaustive. Les recherches que nous avons trouvées^{[1],[18],[19]} se concentrent bien trop souvent sur un nombre d'architecture limité, ou en exécutant les modèles sur un faible nombre de smartphone voir dans certains cas en ignorant complètement l'aspect cross-platform en excluant volontairement certains OS.

Il en découle des résultats variés et variables exposant des métriques différentes. Certaines études mettent en lumière les coûts en termes de temps et de consommation de batterie, alors que d'autres mettent en avant la précision du modèle et le nombre de paramètre de ce dernier.

Ainsi, les observations jusqu'ici ont été faites sur un ensemble faible d'études. Toutefois, puisque l'objectif principal de ce travail s'axe plutôt sur la faisabilité de l'intégration du modèle deep learning sur un smartphone, les informations récoltées sont suffisantes pour nous orienter sur un choix éclairé.

2.1.3 Solution choisie

Les deux techniques présentées ci-dessus nous ont semblé être les plus pertinentes et adaptées pour ce projet. Nous avons décidé d'orienter notre choix final sur une technique de classification d'image.

En effet, les contraintes de ce modèle peuvent être aisément contournée pour les besoins de ce travail. Tout d'abord, nous n'avons pas réellement besoin d'identifier plusieurs éléments au sein d'une même image. À termes, nous pourrions imaginer que l'application envoie le cliché à un serveur accessible par des autorités, permettant à un œil humain d'identifier plusieurs individus sur l'image reçue.

En second temps, la problématique liée au sujet de l'image qui serait trop petit peut être contournée si on invite l'utilisateur à recadrer son cliché en ne sélectionnant que la zone contenant l'insecte à identifier. Cela permet non seulement aux utilisateurs de saisir le cliché depuis un point de vue éloigné à des fins sécuritaires tout en obtenant une image avec un sujet mieux centré et finalement plus facile à reconnaître.

Les classificateurs d'images offrent une solution simple à entraîner et à utiliser et s'avèrent donc être de bons candidats pour tester leur portabilité dans un téléphone. Les autres avantages qu'ils ont à offrir dans notre contexte ont d'ores et déjà été explicité plus haut.

2.2 Datasets

La section précédente nous a fait nous orienter sur une tâche de classification d'images. De ce fait, nous devons désormais rechercher un jeu de données contenant des images du frelon asiatique annotées pour entraîner notre modèle.

Le *Vespa Velutina* étant une espèce invasive dans plusieurs pays dont notamment la France et l'Espagne, cette espèce a déjà été le sujet d'observations et de nombreux clichés divers et variés sont trouvable sur internet. L'idéal étant de disposer d'un ensemble de cliché annoté correctement, ce qui est chose possible au travers de diverses plateformes web mettant à disposition des datasets en open source comme *Roboflow*².

À la date de la mise en ligne du sujet de ce travail, un premier jeu de donnée a été suggéré³. Ceci disposait de 589 images annotées répartie sur 5 classes de la façon suivante :

Classes	Nombre d'images
Asian Hornet (Frelon asiatique)	280
Bee (Abeille)	103
Hornet (Frelon non-asiatique)	99
Wasp (Guêpe)	94
Null (Aucun)	13

Tableau 1 - Répartition des classes dans le premier dataset étudié

Bien que ce dataset mettent à disposition un ensemble de classes pertinentes pour notre besoin, il reste néanmoins de taille trop faible pour espérer une précision suffisante. D'autant plus que les différentes classes sont mal équilibrées. Nous pourrions réduire le nombre de classe à 2 en regroupant les images ne contenant pas de frelons asiatiques en une seule catégorie, mais dans tous les cas, il serait nécessaire de procéder à de l'augmentation de données avec un tel jeu.

Nos recherches ont conduit à d'autres données annotées⁴ de 7675 images de frelons asiatiques. À la différence du jeu précédent, celui-ci ne propose qu'une seule classe. Néanmoins, il dispose également d'une version avec augmentation de donnée, élevant le nombre total d'image à 18'425. Les opérations d'augmentations comprennent des rotations, des rognages, de changements de luminosité, de flou et d'exposition.

Cette large ressource pourrait s'avérer excellente pour disposer d'un modèle suffisamment précis. Toutefois, il sera nécessaire d'agrémenter ces données d'une classe additionnelle contenant des images diverses et variées, voir des insectes à exclure pour affiner le modèle à ne détecter que l'espèce

² Accessible ici : <https://universe.roboflow.com/>

³ Accessible ici : <https://universe.roboflow.com/use-case-asian-hornet-detection/asian-hornet-detection-a6ael/dataset/2>

⁴ Accessible ici : <https://universe.roboflow.com/cyp-puhyr/asian-hornet-2/dataset/1>

recherchée. L'autre solution consisterait à procéder à de l'apprentissage par transfert en sélectionnant un modèle pré-entraîné sur un ensemble de classe. Nous pourrions, par exemple, sélectionner un modèle entraîné sur un dataset particulier et le renforcer en y ajoutant notre nouvelle classe. La **Figure 4** illustre le fonctionnement de l'apprentissage par transfert.

How transfer learning works

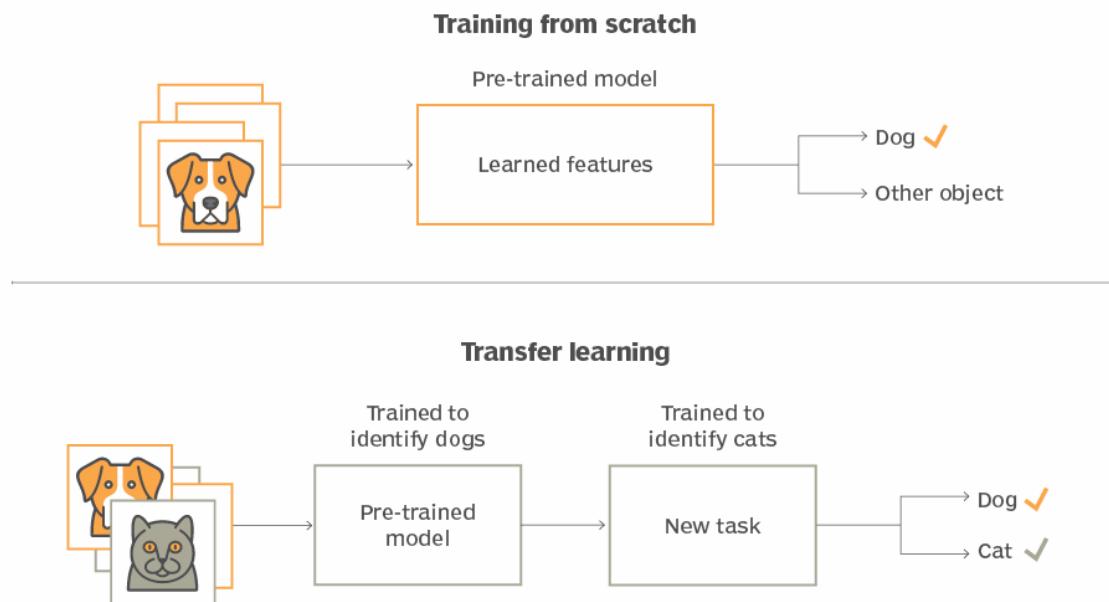


Figure 4 – Fonctionnement de l'apprentissage par transfert^[9]

En haut, un premier modèle entraîné à identifier des chiens. En bas, ce modèle est réutilisé et affiné pour permettre la détection de chat et de chien

En conclusion, nous porterons notre choix sur ce deuxième jeu de donnée. Celui-ci nous permet de retirer une partie du travail nécessaire pour disposer d'un panel large d'image puisqu'il a déjà été augmenté. Ainsi, nous nous assurons de disposer de données suffisamment fournies pour mener à bien l'entraînement du modèle et garantir une certaine qualité quant à sa précision.

Il sera toutefois nécessaire d'agrémenter ce jeu avec d'autres données et de disposer d'une annotation pour cette nouvelle classe à introduire et/ou de rechercher un modèle pré-entraîné sur lequel nous pourrons aisément appliquer de l'apprentissage par transfert.

2.3 Architectures de réseau de neurones

Par architectures de réseau de neurones nous parlons de toutes les structures de réseaux convolutifs existantes et découvertes au travers de la recherche. En effet, si nous souhaitons obtenir un modèle final le plus précis possible, il est préférable de se baser sur des architectures existantes.

Plusieurs paramètres sont à prendre en considération pour sélectionner un modèle performant sur mobile. Pour commencer, celui-ci doit disposer d'un nombre de paramètre le plus faible possible. En effet, si le nombre de paramètre est trop élevé, le modèle prendra plus d'espace mémoire et son temps d'exécution se verra rallongé. Or, sur un mobile, ces ressources sont plus faibles qu'un serveur ou même un ordinateur.

En second temps, nous devons considérer le temps de traitement, appelé aussi temps d'inférence. Un modèle lent à l'exécution pouvant entraîner des conséquences néfastes sur l'expérience utilisateur.

En troisième, le modèle doit avoir une consommation en énergie la plus faible possible. Une application ou le traitement d'image est gourmand en énergie sera inutilisable si on souhaite performer l'opération plusieurs fois.

Finalement, l'architecture retenue doit offrir une bonne précision. Ce point pouvant être impacté si nous tentons de satisfaire les autres critères précédemment. Il nous faudra donc trouver un équilibre entre performances sur mobile et qualité du modèle.

Notre investigation se base sur un travail de recherche (Bhatt, et al., 2021) visant à mesurer les performances de différentes architectures CNN sur mobiles dans un contexte de détection de mouvement d'yeux (*eye tracking*).

2.3.1 Architectures proposées

Le papier se focalise sur 4 architectures dont 3 d'entre elles ont pour point commun le fait qu'elles ont toutes été dimensionnées afin de maximiser les précisions obtenues sur le set *ImageNet*⁵. Nous vous proposons ci-dessous une rapide présentation de ces dernières.

2.3.1.1 LeNet-5

Il s'agit de l'architecture la plus simple en termes de structure. En effet, elle n'est constituée que de 5 couches dont 2 convolutives et 3 entièrement connectées. Malgré sa simplicité, elle s'avère efficace pour des tâches peu complexes et propose un nombre de paramètres faible ainsi qu'un temps d'inférence rapide.

2.3.1.2 AlexNet

Évolution de l'architecture précédente, elle rajoute 2 couches de convolution supplémentaires et une couche entièrement connectée supplémentaire. Elle performe mieux que LeNet-5, mais dispose également d'un nombre de paramètre très élevé, 1'000 fois plus que LeNet-5.

2.3.1.3 MobileNet-V3

Avec sa première version créée en 2017, *MobileNet* est une des premières tentatives d'architecture pensée pour des systèmes embarqués. Sa particularité réside dans son traitement des couches convolutives qui divisent les kernels normalement obtenu en deux. Par exemple, plutôt que d'obtenir un kernel 3x3 en sortie, le modèle créera un kernel 3x1 et 1x3. Cette technique réduit le nombre d'opérations nécessaire pour effectuer la convolution.

2.3.1.4 Shufflenet-V2

Dans la même optique de vouloir apporter une architecture légère pour être fonctionnel sur des systèmes embarqués, *Shufflenet*, dans sa deuxième version, propose de séparer les canaux (par exemple R, V et B) en deux. Les couches de convolution vont alors extraire des caractéristiques sur les images, puis ces caractéristiques seront mélangées (*Shuffle*) aux autres, créant de nouveaux kernels contenant les caractéristiques des différents canaux. Contrairement à *MobileNet*, le nombre de canaux en entrée et sortie reste identique. La **Figure 5** illustre le procédé de cette architecture lorsque celui-ci était encore à sa première version.

Les performances théoriques de ce modèle sont relativement similaires à ceux de *MobileNet*.

⁵ Dataset de 1000 classes d'objets différents, animaux, véhicules, outils, mobilier, etc...

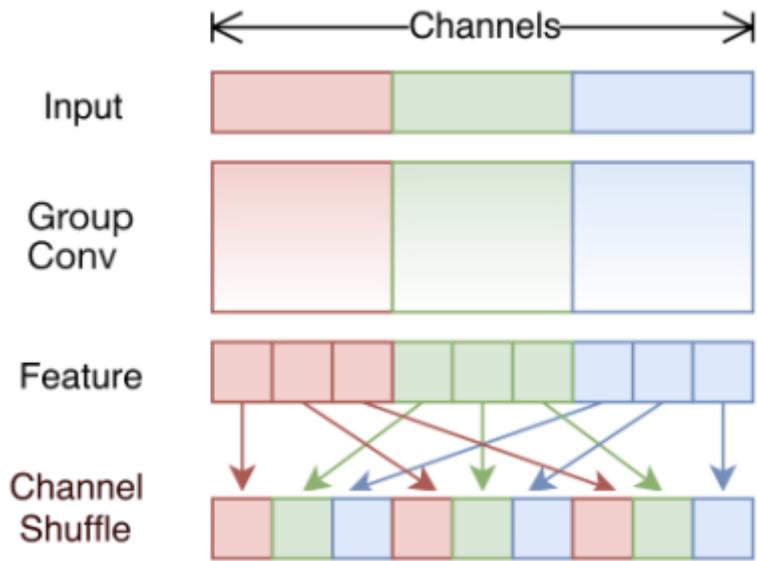


Figure 5 – Étapes de convolutions de ShuffleNet^[13]

Dans sa première version, l'architecture effectuait également un regroupement des couche 1x1. Ceci a été abandonné dans la V2

2.3.2 Résultats obtenus

L'étude a donc testé ces 4 architectures sur deux appareils mobiles : le Samsung Galaxy S9 et le Samsung Galaxy J7. À noter que les chercheurs sont allés plus loin en proposant des architectures différentes. Ainsi, en plus de tester ces modèles directement embarqué sur les téléphones mobiles, ils ont également testé les performances si ces derniers déléguait la tâche à un serveur proche (*Edge*) ou sur une infrastructure dans le cloud. Nous ne nous intéresserons qu'aux performances obtenues lorsque le modèle est présent sur les smartphones directement.

En ce qui concerne les performances en termes d'espace mémoire et de consommation de batterie, les résultats observés sont les suivants. Toutes les mesures ont été effectuées sur une tâche de classification de 1'000 images. La **Figure 6** affiche les résultats sous la forme d'un histogramme pour les deux modèles de smartphones utilisés.

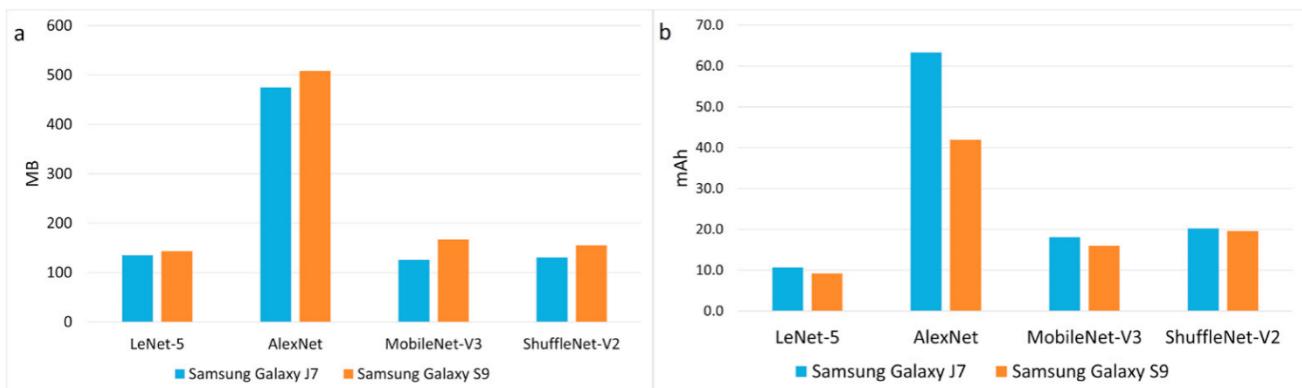


Figure 6 – Espace mémoire et consommation d'énergie lors d'analyse de 1'000 images^[18]

On remarque qu'AlexNet est le modèle ayant le plus d'impact sur l'usage de la mémoire et la consommation de batterie, ce qui n'est guère surprenant car il s'agit du modèle disposant du plus grand nombre d'hyperparamètres. Pour rappel, les nombres de paramètres des différents modèles est présenté dans le **Tableau 2**.

Architecture	Nombre approximatif de paramètres
LeNet-5	60'000
MobileNet-V3	5.4 Mio
ShuffleNet-V2	7.4 Mio
AlexNet	60 Mio

Tableau 2 - Nombre de paramètres à entraîner selon l'architecture^[2]

De ce fait, on constate une corrélation entre le nombre de paramètre du modèle et sa taille en mémoire ainsi que sa consommation de batterie. En tant que tel, le nombre de paramètres n'est pas une information suffisante pour orienter notre choix, nous devons également observer la précision et le temps d'inférence du modèle. La **Figure 7** représente ceci en séparant les données par architecture mais aussi par méthodologie utilisée. Dans notre cas, seul le « *On-device inference* » nous intéresse.

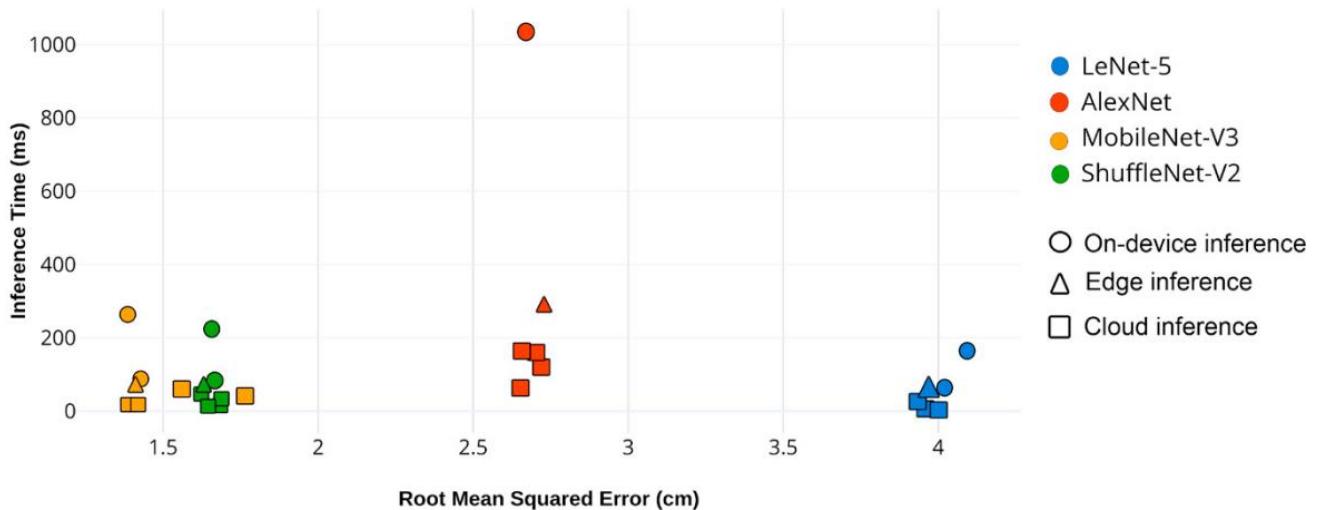


Figure 7 – Écarts quadratiques moyen par rapport au temps d'inférence^[18]

Si on observe les cercles dans le schéma ci-dessus, on constate également une corrélation entre le nombre de paramètre de l'architecture et le temps d'inférence. Ainsi, on retrouve AlexNet comme architecture avec le temps le plus élevé.

Ce schéma nous donne également l'indication que malgré les avantages en termes de consommation de ressource que peut offrir LeNet-5, cette architecture restent la moins performante en termes de précision.

L'architecture MobileNet-V3 et ShuffleNet-V2 sont toutes deux similaires en termes de performances. La première semble toutefois disposer d'une précision légèrement plus élevée. Nous choisirons donc une architecture MobileNet-V3 tout en conservant ShuffleNet-V2 comme alternative si besoin.

2.3.3 Solution choisie

L'architecture MobileNet-V3 dispose de nombreuses qualités démontrées au cours de cette étude qui oriente notre choix sur elle. En effet, même si cette architecture n'est, à priori, pas la plus rapide en termes d'exécution, elle offre en contrepartie une précision plus élevée ainsi qu'une consommation d'énergie et de mémoire moindre sur plateforme mobile.

Nous n'écartons toutefois pas la possibilité d'utiliser une architecture basée sur ShuffleNet-V2 si cette dernière s'avère plus performante lors de nos expérimentations.

2.4 Intégration de modèle dans une application cross platform

Cette section s'intéresse aux différentes étapes et solutions logiciels existantes qui permettent à terme de transposer un réseau de neurones convolutifs sur une plateforme mobile. Ce procédé passe par plusieurs étapes. Du choix de la librairie permettant de réaliser le modèle à son inférence sur mobile en passant par le choix du *framework* cross-platform. Nous tenterons d'exposer dans cette section, les différentes étapes ainsi que les solutions existantes.

2.4.1 Crédit du modèle

Dans le domaine de la recherche et dans le domaine professionnel, Python s'avère être le langage de programmation de prédilection en ce qui concerne la data science et par extension le machine learning. Ceci grâce à une communauté large et bien établie.

De surcroît, de grosses entreprises tel que Google et Meta ont investi dans le développement de librairie open source afin de faciliter l'apprentissage et la prise en main de ces outils. Nous pouvons retrouver Google derrière les librairies *TensorFlow* et *Keras*, et en ce qui concerne Meta, cette dernière est la créatrice de *PyTorch*.

Ces trois solutions facilitent l'accessibilité de la création de modèles deep learning. Chacune disposant d'avantages et inconvénients qui leur sont propres. *TensorFlow*, par exemple, dispose d'une API de plus bas niveau que *PyTorch* ou *Keras* donnant un plus large contrôle sur les détails d'implémentations et d'optimisation du modèle deep learning.

PyTorch et *Keras*, quant à eux, souhaitent mettre à disposition une API de haut niveau dans l'objectif de permettre aux utilisateurs de créer des prototypes fonctionnels rapidement en offrant un grand niveau d'abstraction en ce qui concerne le nombre de couches, et les valeurs des hyperparamètres. Cette approche est certes plus limitée dans des cadres de recherche avancée en machine learning, mais se prête très bien à notre contexte où le modèle en soit n'est pas l'objectif principal.

Notons également le fait que puisque que *Keras* et *TensorFlow* ont été développé par Google, ces deux librairies disposent d'une intégration commune. Ainsi, il est possible de disposer facilement des fonctionnalités de *TensorFlow* au travers de l'API de *Keras*. Par ce procédé, ces deux librairies deviennent un choix souvent préféré à *PyTorch*.

En plus de celles susmentionnées, *TensorFlow* offre d'autres fonctionnalités que nous détaillerons plus bas dans ce rapport. La suite de cette section détaillera aussi notre utilisation de cette librairie et pourquoi cette dernière nous a convaincus.

2.4.2 Export du modèle

Rappelons que nous souhaitons procéder à l'inférence du modèle directement sur la plateforme mobile. De ce fait, une solution développée au moyen d'un script python ne sera pas utilisable en tant que tel. Heureusement, il existe plusieurs formats de fichier et de méthode d'export permettant l'utilisation de modèles entraînés sur d'autres appareils.

2.4.2.1 TensorFlow Lite

L'approche la plus instinctive consisterait à utiliser directement les outils fournis par *TensorFlow* et plus particulièrement *TensorFlow Lite*. Il s'agit d'une librairie réalisée spécialement pour porter des modèles entraînés via *TensorFlow* ou *Keras* sur des plateformes mobiles.

En plus de fournir un format de fichier exportable, la librairie met également à disposition un ensemble d'outils pour optimiser le modèle lors de son exportation. Nous pouvons par exemple souligner la possibilité de quantifier le modèle afin que celui-ci utilise des entiers sur 8 bits plutôt que des nombres à virgules flottantes sur 32 bits. Ce procédé permet une simplification des calculs à effectuer sur la machine hôte, en particulier si celle-ci dispose de ressources matérielles limitées.

Le seul réel inconvénient de cette méthode et qu'il n'est, à l'heure actuelle, pas possible de transposer les libellés des classes dans le fichier exportable. Ainsi, il est nécessaire de fournir en annexe du modèle un fichier textuel contenant les libellés. Cela peut poser plusieurs inconvénients dans un contexte où nous disposons de beaucoup de classes. Dans notre cas, puisque le modèle sera binaire (2 classes), il ne sera probablement même pas nécessaire de fournir de libellés, puisque la sortie unique du modèle pourra être directement interprétée.

2.4.2.2 Open Neural Network Exchange (ONNX)

Il s'agit d'un écosystème d'intelligence artificielle open source ayant pour but de standardiser les opérations et les fichiers d'export afin d'assurer une interopérabilité entre les différents *frameworks* de machine learning. Cette initiative a été lancée par *Meta* et *Microsoft* en 2017 et a été rapidement soutenue par d'autres grandes industries comme *AMD*, *Intel* et *ARM* pour ne citer qu'eux.

Du fait du soutien d'industriels importants, c'est naturellement que nous retrouvons la possibilité d'exporter un modèle *Keras* ou *TensorFlow* au format ONNX.

Toutefois, *ONNX* définit un format de modélisation. Si nous souhaitons réutiliser un modèle exporté via ce standard, nous devons employer un autre outil : *ONNX Runtime*. Or, au moment de la rédaction de ce rapport, cet outil est limité à certains langages. En ce qui concerne le cross-platform, seul *React Native* dispose d'une solution *ONNX Runtime* officielle. En ce qui concerne d'autres langages *cross-platform*, comme *Flutter*, seules des librairies non-officielles publiées par des utilisateurs non-vérifiés sont disponibles. La **Figure 8** illustre le résultat de recherche sur le site de package de *Flutter* pub.dev. On y retrouve un ensemble de produit aux popularités variables. En comparaison, l'intégration avec *React Native* est directement documentée sur le site de *ONNX Runtime*⁶. L'utilisation de ces librairies tierces nous semble donc une mauvaise piste et plus risqué qu'avec celle de *TensorFlow Lite*.

⁶ Lien vers la doc pour l'implémentation de ONNX sur une application React Native : <https://onnxruntime.ai/docs/get-started/with-javascript/react-native.html>

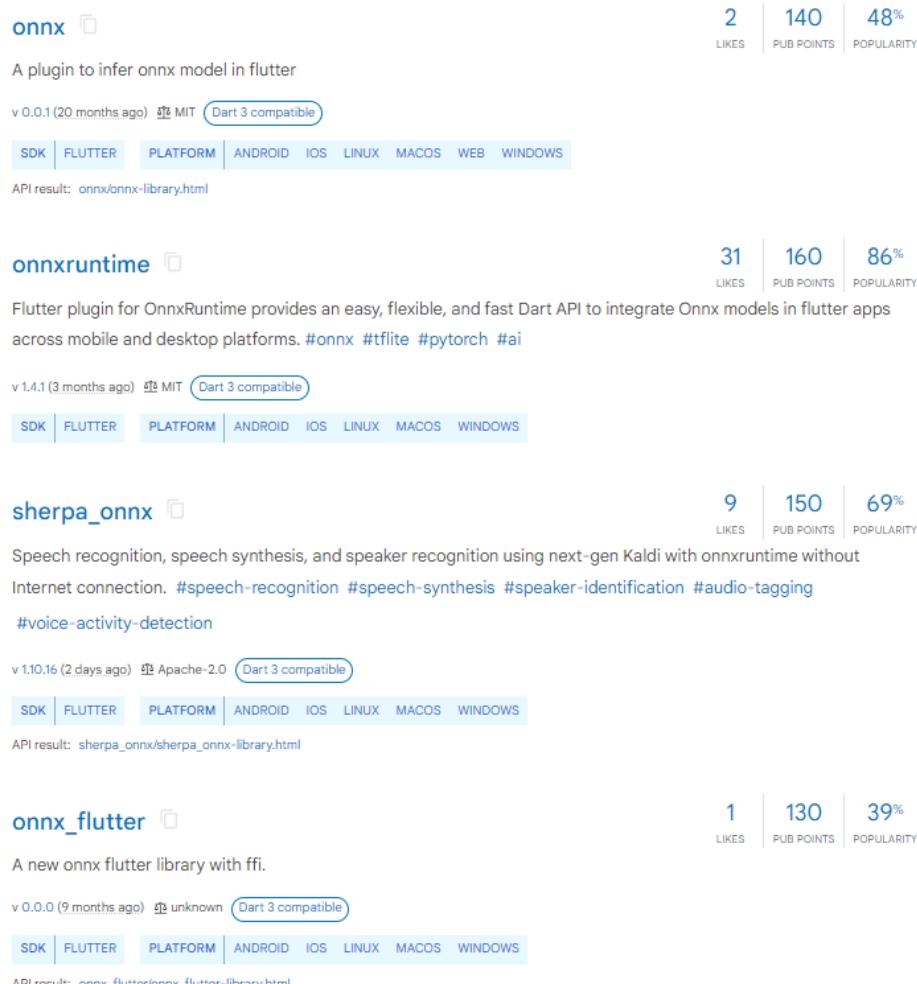


Figure 8 – Librairies ONNX de flutter⁷

Liste des librairies retournées lors d'une recherche sur le site pub.dev. On y voit des popularités variables et des nombres de "likes" faibles, indiquant des utilisations faibles ou une communauté restreinte

Toujours concernant Flutter, il serait possible de contourner cette limitation en passant par l'intégration C++ du ONNX Runtime via le Foreign Function Interface (FFI) de Dart. Cela consiste à employer les outils fournis par Flutter afin de générer du code Android et iOS dans leur version native en C++.

2.4.2.3 Solution choisie

Comme mentionné, la solution la plus instinctive serait d'utiliser *TensorFlow Lite*. Son intégration avec la librairie *Keras* facilite grandement l'export du modèle tout en mettant à disposition des outils simples pour optimiser son inférence sur un smartphone. Ces outils nous permettront de réduire grandement le temps nécessaire à la gestion de l'export du modèle sur d'autres appareils, c'est pour cela que nous porterons notre choix sur cette technologie.

À contrario, le standard *ONNX* impose des contraintes et n'est pas supporté de façon officielle sur d'autres langages *cross-platform* que *React Native*. Le passage par le code natif en C++ nous semblant trop fastidieux et complexe pour être une solution fonctionnelle à court terme.

2.4.3 Solution cross-platform

Dans un premier temps, et pour mener à bien l'objectif principal de ce projet, nous devons disposer d'une solution déployable à la fois sur Android et iOS. Ainsi, le langage utilisé importe peu tant que

⁷ Résultats obtenus après une recherche du termes « ONNX » sur : <https://pub.dev/>

celui-ci dispose d'un écosystème développé et d'une communauté active nous permettant d'implémenter les solutions présentées jusqu'ici.

Idéalement, et si le temps le permet, il serait intéressant d'explorer une solution faisant abstraction du langage utilisé. Une piste à explorer serait d'implémenter une solution en code natif C++ exposant une API qui pourra être utilisée soit en *React Native* soit en *Flutter*, ou en tout autre langage *cross-platform* disposant d'une solution permettant l'intégration de librairie C++. Notons également que *TensorFlow Lite* est disponible en C++.

Cette solution « universelle » repose sur un défi de taille. Toutefois, *React Native* et *Flutter* disposent tous deux d'outils permettant la réalisation de code natif C++ transposable par la suite à la fois sur iOS et Android. Par exemple, *React Native* met à disposition ce qu'ils appellent le « *Native Module*⁸ » permettant l'appel de code C++ depuis le langage *JavaScript*. De même *flutter* permet le même procédé via le « *Foreign Function Interface* » mis à disposition par le langage *Dart*. Cependant, la documentation concernant ces deux techniques reste relativement légère. À l'heure actuelle, et sans expérimentation, il est difficile d'estimer la complexité ainsi que la durée de travail d'une telle implémentation.

À prendre en compte également qu'à la rédaction de ce rapport, *React Native* est en phase de terminer le déploiement de sa « Nouvelle architecture⁹ », prévue selon eux « à la fin de 2024 ». Bien qu'il soit mentionné explicitement que cette version est en phase expérimentale et qu'il est donc préférable d'éviter de l'utiliser immédiatement, il est, de surcroît, indiqué qu'elle affectera l'utilisation des *Natives Modules* qui seront dépréciés lors du déploiement de la nouvelle architecture, laissant la place à d'autres implémentation que sont *Turbo Native Module*¹⁰ et *Fabric Native Components*¹¹. Par conséquent, il est important de souligner qu'une solution développée en *React Native* lors de ce projet risque de devenir obsolète dans les mois à venir.

En conséquence, et dans le cadre d'une solution cross-platform Android et iOS uniquement, nous préférerons l'utilisation du *framework Flutter* qui semble disposer d'une version plus stable dans la fenêtre de temps imparti pour la réalisation de ce projet. Nous n'excluons pas complètement *React Native*, et nous reviendrons dessus si nous venions à réaliser une solution « universelle » adaptable à plusieurs langages et frameworks *cross-platform*.

⁸ Pour en savoir plus sur les *Native Module* : <https://reactnative.dev/docs/native-modules-intro>

⁹ Pour en savoir plus sur la nouvelle architecture : <https://reactnative.dev/docs/the-new-architecture/landing-page>

¹⁰ Pour en savoir plus sur les *Tubor Modules* : <https://github.com/reactwg/react-native-new-architecture/blob/main/docs/turbo-modules.md>

¹¹ Pour en savoir plus sur le *Fabric Native Components* : <https://github.com/reactwg/react-native-new-architecture/blob/main/docs/fabric-native-components.md>

2.4.4 Inférence du modèle sur mobile

Récapitulons l'ensemble des technologies choisies jusqu'ici :

- Nous disposerons d'un modèle dont l'implémentation aurait été réalisée à l'aide de *Keras* et *TensorFlow*.
- Le modèle sera ensuite exporté dans un fichier unique auquel nous pourrons, si nécessaire, ajouter un fichier textuel supplémentaire définissant les libellés de nos classes.
- L'inférence du modèle sera développée grâce au framework *Flutter*.

À présent, nous devons sélectionner les technologies nous permettant d'importer et d'exécuter le modèle au format *TensorFlow Lite* via *Flutter*. Par chance, il s'avère que toutes les technologies utilisées ici sont des créations de Google. Ainsi, il nous a été relativement simple de trouver une solution officielle maintenue par *TensorFlow* eux-mêmes. Il s'agit d'un plugin nommé *tflite_flutter*¹² qu'il suffit d'installer via le gestionnaire de dépendance du framework. Nous avons pu explorer cette solution au travers d'un prototype que nous explorerons dans la section suivante.

Il existe d'autres solutions facilitant l'intégration de modèle de deep learning. Google mettant à disposition au développeur mobile *ML Kit*. Toutefois, cette boîte à outil gargantuesque a été initialement prévue pour développer des solutions directement sur la plateforme mobile dédiée, donc soit Android, soit iOS. En ce qui concerne le cross-platform, Google opte plutôt sur une stratégie d'inférence sur un serveur cloud au travers d'une service *Firebase* dédié : *Firebase ML*.

Néanmoins, la communauté *Flutter* met à disposition un plugin¹³ permettant son utilisation dans ce framework, rendant ainsi *ML Kit cross-platform*. Nous n'avons malheureusement pas exploré cette solution car, bien qu'elle soit maintenue par un publieur vérifié, il ne s'agit pas de Google comme c'est le cas pour le plugin mentionné précédemment, on pourrait donc s'attendre à ce qu'une solution officielle soit supportée plus tard. Le répertoire *Github* du projet communautaire¹³ dispose toutefois d'un exemple de classification d'image fonctionnel, et consiste donc en une alternative envisageable en cas de problème rencontré avec l'autre plugin.

2.5 Modélisation et architecture d'un prototype

Nous avons tenté de réaliser une première ébauche comprenant l'ensemble des technologies préférées qui vous ont été présentées jusqu'à maintenant. L'objectif final de notre prototype était de tester la faisabilité de notre approche en réalisant la classification d'une image quelconque à l'aide d'un modèle deep learning depuis une application mobile. Le prototype s'est axé en priorité sur une application fonctionnelle faisant fi des optimisations possibles aux différents niveaux et de l'interface utilisateur. Nous allons détailler les différentes parties dans les sous-sections suivantes ainsi que les problèmes rencontrés et comment ils ont été résolus.

2.5.1 Réalisation du modèle

Nous avons donc réalisé cette partie au moyen d'un script Python en utilisant les librairies *TensorFlow* et *Keras*. Plus précisément, nous avons importé un modèle utilisant l'architecture MobileNet V2 que nous avons initialisé avec les poids du dataset ImageNet. Ce faisant, nous n'avons donc pas eu besoin de procéder à un quelconque entraînement.

Nous avons rapidement testé le modèle en fournissant des images pour vérifier que celles-ci étaient correctement prédites. Une contrainte que fourni *Keras* est qu'il est nécessaire de redimensionner

¹² Lien vers le package *tflite_flutter* : https://pub.dev/packages/tflite_flutter

¹³ Lien vers le repository du plugin communautaire pour *ML Kit* : https://github.com/flutter-mi/google_ml_kit_flutter

l'image fournie pour qu'elle puisse être processée par le modèle. En l'occurrence, l'architecture que nous avons utilisée oblige un format de 224 pixels par 224 avec 3 canaux pour les couleurs RGB en entrée.

Le modèle fournit en sortie une probabilité d'appartenance pour les 1000 classes disponibles dans *ImageNet*. Keras fournit les libellés de ces classes. L'exécution du script python nous donne donc en sortie le nom des classes ainsi que le score de probabilité d'appartenance à ces dernières.

Aucun problème n'a été rencontré pendant cette phase. Les bibliothèques sont très faciles d'accès et il est possible d'obtenir un modèle avec lequel expérimenté rapidement en quelques lignes de codes.

2.5.2 Exportation du modèle

Puisque nous importons le modèle et les poids directement depuis Keras nous avons dû le sauvegarder en local au format « *SavedModel* ». Ce format enregistre un dossier contenant diverses caractéristiques sur le modèle. À noter qu'il n'est pas possible de le sauvegarder directement au format *TensorFlow Lite*, il faut au préalable le sauvegarder en *SavedModel* ou *.keras*, la documentation officielle recommandant le premier format.

Une fois le modèle sauvegardé, il faut le convertir au format *.tflite*, qui pourra ensuite être exporté sur la plateforme de notre choix. La conversion du fichier est également triviale, mais il est important de noter que les libellés des classes sont perdus lors de ce processus.

Nous avons pu confirmer que la conversion n'altérait pas les résultats du modèle puisque Keras met également à disposition l'exécution de modèles importés via de tels fichiers. Les résultats obtenus avec nos images de tests étaient identiques une fois la conversion effectuée.

2.5.3 Réalisation de l'application

Nous sommes partis d'un *template* d'application Flutter vierge. Ce dernier contient simplement un bouton en bas à droite de l'écran incrémentant un compteur qui est affiché au centre. Nous avons simplement retiré l'incrémant et modifier le comportement du bouton. Celui-ci demande maintenant à l'utilisateur de sélectionner une image de sa galerie. Ceci est réalisé au moyen du package *image_picker*.

Après inférence du modèle, celui-ci nous retournera une collection sous la forme d'une *Map* ayant en clé des chaînes de caractères, les libellés de nos classes, et comme valeur des nombres à virgules flottantes, les probabilités d'appartenance. Nous procédons ensuite à une mise en forme des données de telle façon à ce que nous obtenions une liste des données triées de façon descendante sur les valeurs des probabilités, puis nous limitons le nombre de résultats dans la collection à 3. Dis autrement, nous sélectionnons les 3 meilleurs résultats de classification retourné par le modèle.

Hormis quelques méconnaissances du framework nécessitant quelques ajustements ça et là ne méritant pas de mention spéciale, nous avons pu obtenir une interface sommaire et minimaliste nous permettant de manipuler des images en entrée, de les transmettre à un modèle et d'en afficher les résultats.

2.5.4 Import et inférence du modèle

Pour utiliser un modèle ainsi que ses libellés via le plugin *tflite_flutter*, nous devons au préalable les définir comme assets dans notre projet Flutter. Cela consiste à créer un dossier éponyme à la racine du projet, puis d'en indiquer le chemin depuis le fichier *pubsec.yaml* du projet.

Nous devons également fournir en asset un fichier textuel contenant l'ensemble des libellés de chacune de nos classes. Nous avons donc récupéré ce fichier en ligne, chaque classe étant séparée par un retour à la ligne. Le contenu de ce fichier a ensuite été chargé dans une variable sous forme d'une liste de chaînes de caractères.

En ce qui concerne le code pour l'inférence du modèle, nous nous sommes inspirés du code fourni en exemple sur le répertoire Github¹⁴ des développeurs du plugin eux-mêmes. Cet exemple est doté d'une implémentation plus complexe que nécessaire pour notre prototype pour la bonne raison que leur implémentation permet à la fois la classification d'image issue de la galerie de l'utilisateur, mais également de classifier des images live reçue par la caméra de l'utilisateur.

Cette deuxième fonctionnalité oblige l'utilisation de la structure *Isolate*¹⁵ de Flutter. En effet, sans l'usage de telles structures, l'application subirait des latences importantes puisque la caméra serait figée le temps que l'image soit traitée par le modèle. L'expérience utilisateur s'en verrait affectée.

Pour notre prototype, nous n'avons pas besoin d'utiliser ces structures. Nous avons donc pris soin d'extraire les parties du codes importantes, à savoir celles permettant de récupérer les informations depuis le fichier importé et celles permettant de redimensionner l'images aux bonnes dimensions d'entrée du modèle.

La librairie *tflite_flutter* permet une grande simplification de l'import du fichier. Il suffit en effet d'instancier un interpréteur en lui fournissant le chemin relatif de l'assets de notre fichier *.tflite*. Ensuite, cet interpréteur dispose de 3 méthodes importantes. Une première permet de récupérer les dimensions du tenseur en entrées qui dans notre cas est [1, 224, 224, 3]¹⁶, une deuxième permet de récupérer le tenseur de sortie, à savoir [1, 1000]¹⁷. Et finalement une troisième qui exécutera le modèle et qui prends en paramètre une entrée (notre image) et écrit les résultats dans le second, la sortie.

L'étape suivante consiste maintenant à transformer notre image en valeurs numériques et au format du tenseur attendu en entrée. Ici, le package *image* nous a été utile. Il permet de modéliser une image comme étant une collection itérable de pixels et réalise ainsi pour nous la conversion des pixels en valeurs numérique. Avec ce package, nous avons créé en premier temps une copie de l'image d'entrée aux dimensions 224x224. Nous avons ensuite créé une matrice de l'image en parcourant chaque pixel de celle-ci. Nous obtenons ainsi le tenseur d'entrée de notre modèle.

Pour le tenseur de sortie, il suffit de créer une matrice d'une entrée à 1000 valeurs initiées à 0.

Après l'inférence, nous devons encore associer chacune des sorties au bon libellé. La difficulté qui peut résider ici et qu'il est nécessaire de savoir exactement à quoi corresponde chacune des sorties du modèles, mais également de leur ordre, auquel cas les résultats ne seront pas compréhensibles. Une fois l'association entre le libellé et sa valeur de probabilité, il suffit de retourner la collection et de l'afficher dans l'interface.

Ce prototype a été relativement simple à mettre en place, les parties les plus complexes résidant dans le pré-processing de l'image et l'association des résultats à la bonne classe. Nous avons d'ailleurs rencontré un problème d'encodage lors du pré-processing de l'image entre Flutter et Python. En effet, dans Python, la valeur des pixels de l'image en entrée avait été normalisé entre -1 et 1, au contraire du package *image* de Flutter qui a préféré une représentation non normalisée entre 0 et 255. Cette différence nous a causé des confusions, puisque nos premiers tests retournaient des valeurs étonnamment basses (moins de 0.001% de probabilité) et que systématique le top 3 des classes attribuées étaient hors sujet. Par exemple, un chien était alors prédit comme étant un rideau de douche.

Après correction et en normalisant également les données dans le code Flutter, nous obtenions des résultats similaires au script python pour les mêmes images en entrée.

¹⁴ https://github.com/tensorflow/flutter-tflite/tree/main/example/image_classification_mobilenet

¹⁵ Sorte de mini-thread équivalent au coroutine dans Android

¹⁶ Respectivement : [Taille du batch, Largeur, Hauteur, canaux de couleurs (RGB)]

¹⁷ Respectivement : [Taille du batch, Nombre de sorties]

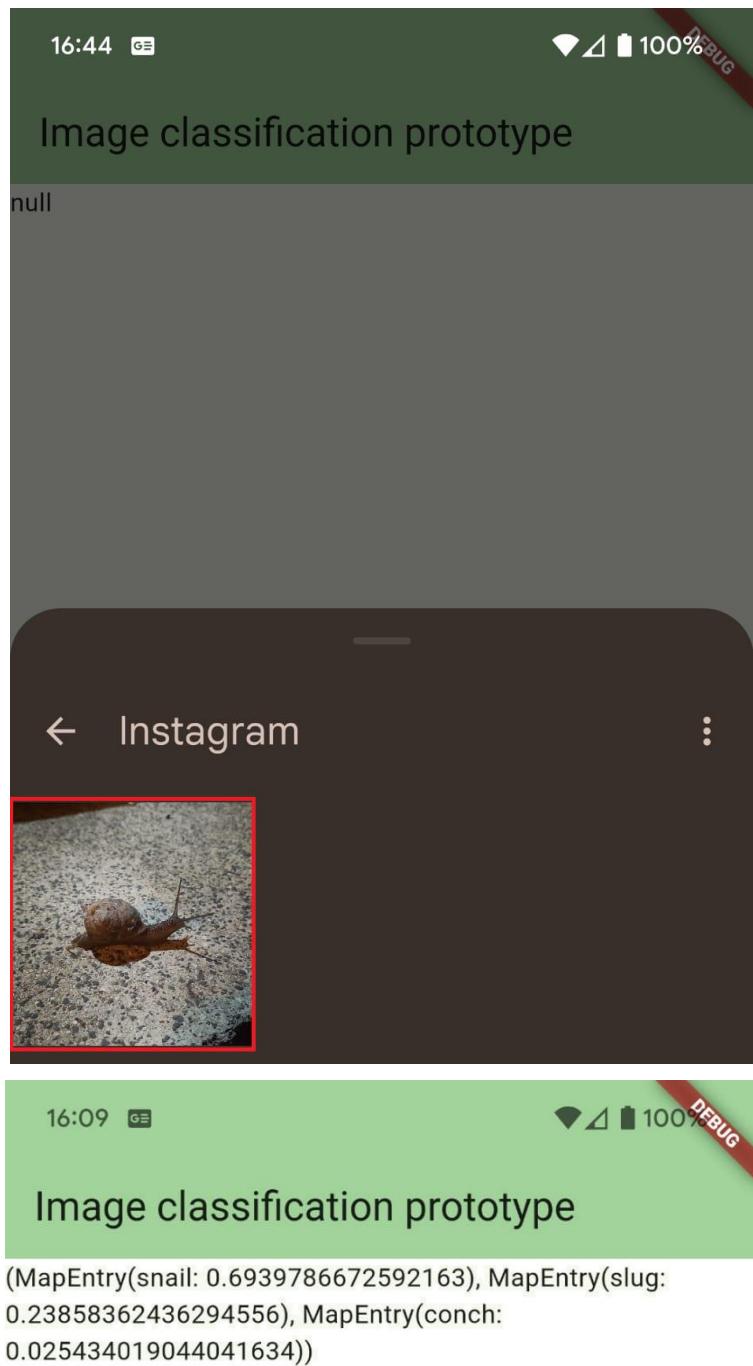


Figure 9 – Résultat du prototype Android après inférence d'une image

En sélectionnant une image d'escargot, le prototype donne les 3 classes les plus probables. Ici, on constate que le modèle attribue la classe « escargot » avec une probabilité de 69%.

Chapitre 3

Implémentation

Arriver à réalisation pratique de ce travail, nous avons convenu avec M. Dutoit la révision du cahier des charges initial afin de concentrer l'effort sur le point central : l'implémentation d'une solution *cross-platform* mobile permettant de réaliser une classification d'image avec une inférence réalisée directement sur le téléphone. En comparaison avec notre prototype, cela inclue l'ajout d'une interface plus claire permettant de définir diverses options supplémentaires d'inférence (utilisation du GPU, nombre de threads, etc..) ainsi que l'intégration fonctionnelle de modèle à virgule flottante ou quantifié.

Avec son accord, nous avons donc retiré les fonctionnalités du cahier des charges initial liées à l'insertion de coordonnées géographiques, à l'envoi de données à un serveur distant et à leur persistance, ainsi qu'au renseignement des trajectoires de vols du frelon asiatique.

En effet, nous avons conclu ensemble que ces fonctionnalités, bien que nécessaires dans la réalisation d'une app fonctionnelle avec un cas d'utilisation précis, constituaient un intérêt plutôt limité étant donné leur nature triviale. Nous sous-entendons par-là que les technologies impliquées lors de leurs implémentations n'imposent pas de difficultés ni de limitations particulières, et donc qu'il est préférable de concentrer l'effort sur le modèle *deep-learning cross-platform*.

De même, les fonctionnalités optionnelles du cahier des charges n'ont pas été implémentées.

3.1 Architecture

Nous inclurons dans l'architecture l'ensemble des éléments qui composent ce projet. Nous identifions ainsi trois éléments principaux reliés entre eux. La **Figure 10** schématisé ces trois blocs.

En premier, le script permettant d'obtenir un modèle de classification pour le frelon asiatique, visible par le bloc du haut de la **Figure 10**. Ce script réalisera le pré-traitement des données comme le recadrage, ou l'augmentation de donnée. Puis, il se chargera de la sauvegarde du modèle ainsi que de son export ou format *tflite*.

Afin d'assurer une utilisation large de notre solution à diverses application *Flutter*, nous avons conçu notre solution au sein d'un *Dart package* créé pour l'occasion. En effet, cette méthode permet l'import de notre implémentation au sein de plusieurs applications *Flutter* en ne changeant que les paramètres d'entrées. Précisons que l'appellation *Dart package* fait référence au nom officiel utilisé dans la documentation de *Flutter*. Ce dernier peut inclure ou non des dépendances avec le *framework Flutter*, mais il conservera cette appellation *Dart Package*. Dans notre cas, nous avons typiquement une dépendance avec *Flutter* puisque notre package utilise *tflite_flutter*.

De ce fait, le deuxième élément de notre architecture réside en une application de démonstration réalisée avec l'aide de *Flutter*, visible dans le bloc du milieu dans la **Figure 10**. Il s'agit ici d'une simple application graphique permettant l'utilisation de notre *package*. Ici, il s'agira de mettre en évidence les fonctionnalités offertes par notre *package*.

Finalement, le troisième élément sera donc un *Dart package* contenant l'ensemble du code nécessaire permettant l'inférence d'une image en lui fournissant en entrée le chemin vers celle-ci, le modèle à utiliser au format *TensorFlow Lite*, et les labels associés aux classes. Le bloc final de la **Figure 10** représente ce package.

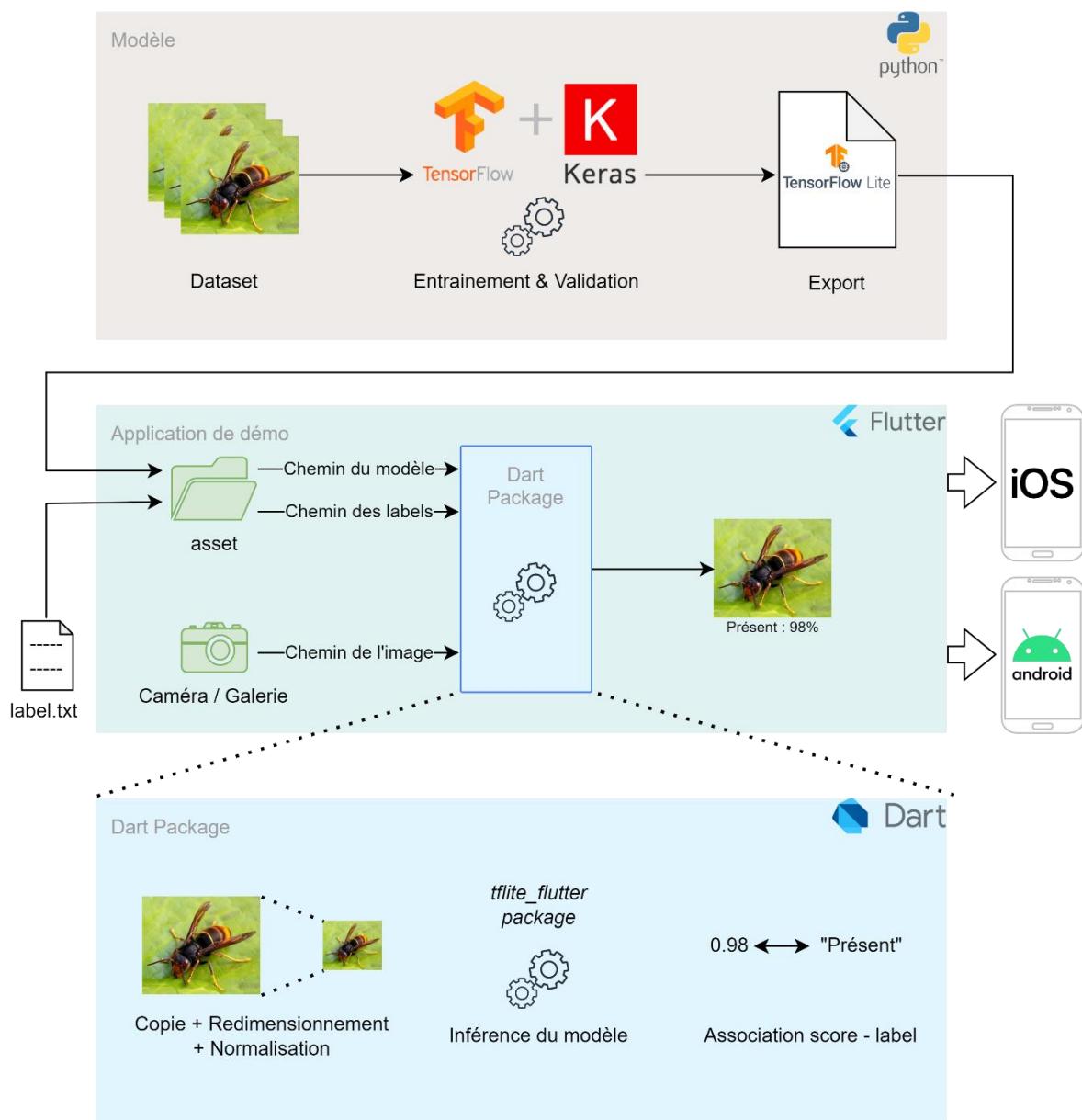


Figure 10 - Architecture globale

Chaque rectangle constitue un bloc du projet (de haut en bas : le modèle, l'app démo, le Dart package). L'app démo prendra un fichier `tflite` contenant le modèle et un fichier texte contenant les labels. Ces derniers en plus de l'image seront fournis au package pour réaliser l'inférence

3.2 Entraînement du modèle

Comme mentionné à plusieurs reprises dans ce rapport, la clé de voûte de ce projet ne réside pas dans la réalisation d'un modèle de classification d'image performant. Ce modèle sert de prétexte afin de d'explorer et de développer une solution mobile *cross-platform* prenant en paramètre un modèle *deep-learning* dont l'inférence serait exécutée directement sur le téléphone et non pas déléguée à un serveur distant.

Par conséquent, nous n'avons pas la prétention d'avoir fourni un modèle d'excellente qualité, ni d'avoir tenté de façon exhaustive les différentes pistes d'amélioration afin de maximiser les performances dudit modèle. Malgré tout, nous pouvons quand même analyser le travail fourni et porter un regard critique sur ce dernier tout en fournissant différents axes d'améliorations dont le travail pourrait bénéficier.

3.2.1 Dataset complémentaire

Dans le chapitre précédent, nous avions fait état de la situation en expliquant la nécessité d'agrémenter le *dataset* du frelon asiatique avec un autre jeu de données, tout ceci dans le but de procéder à une méthode de *transfert learning* (c.f. la **Figure 4**). Nous avons donc commencé par chercher un *dataset* supplémentaire pertinent ne contenant pas le *Vespa Velutina* afin d'affiner le modèle pré-entraîné pour la réalisation d'une tâche plus spécifique.

Par « pertinent », nous sous-entendons un jeu de données permettant de maximiser la capacité de notre réseau de neurone à extraire les caractéristiques utiles de l'image lui permettant de reconnaître la présence ou non de l'insecte recherché. Ainsi, cette « pertinence » est étroitement liée au cas d'utilisation du modèle. À priori, entraîner le modèle à différencier les images contenant un frelon parmi un mélange entre des images de ce dernier et celles de camions de pompier ne présente pas de grand intérêt, car nous pouvons aisément partir du principe que l'être humain moyen sache différencier les deux.

Plus sérieusement, entraîner un modèle de cette façon ne pourra pas nous garantir que ce dernier identifie les bonnes caractéristiques lui permettant de réaliser une prédiction correcte. Dans notre exemple, il pourrait très bien associer la couleur rouge à la prédiction « Pas de frelon asiatique », la présence ou non de cette caractéristique n'est clairement pas déterminante ni suffisante pour obtenir des résultats corrects.

L'idéal aurait été de disposer d'un *dataset* supplémentaire contenant l'ensemble des variétés de guêpes et frelons européens. Cet ensemble d'espèces dispose de caractéristiques physiques très semblable, ce qui aurait obligé le modèle à identifier les détails de chacune afin d'identifier au mieux la bonne espèce.

Malheureusement, nous n'avons pas trouvé un tel *dataset*, nous avons néanmoins trouver une équivalence qui nous avons jugé acceptable compte tenu de notre volonté de ne pas trop approfondir la création du modèle. Cette solution réside en un jeu de données de différents insectes¹⁸ (chenilles papillons, coléoptères, etc...).

Nous avons fusionné les deux datasets et modifié les libellés de ceux-ci pour que chaque image soit annotée « *present* » si elle contient un frelon asiatique et « *absent* » dans le cas contraire.

3.2.2 Adaptations des tailles des datasets

Il est primordial que les *datasets* soient équilibrés lors de l'entraînement d'un réseau de neurones. En effet et dans notre cas, une classe surreprésentée par rapport à une autre aurait pour conséquence

¹⁸ Disponible ici : <https://universe.roboflow.com/pests-data/pests-dataset102>

que notre modèle serait d'avantage tenté de deviner la classe à laquelle appartient l'image en privilégiant la classe surreprésentée, plutôt que d'analyser les caractéristiques contenues dans l'image.

Nous disposons d'un *dataset* bien fourni pour le frelon asiatique, ce qui n'est pas le cas pour le jeu de données complémentaire. Le **Tableau 2** présente les différentes tailles respectives en nombre d'images des *datasets*.

Répartitions des jeux d'images	<i>Dataset Vespa Velutina</i>	<i>Dataset insectes nuisibles</i>
Entraînement	16'125	6'718
Validation	1'535	1'923
Test	765	954
Total	18'425	9'595

Tableau 3 - Nombre d'images par ensemble des datasets utilisés

Les valeurs sont réparties selon leur utilisation pendant la conception du modèle (entraînement, validation, test)

Nous disposons de trop de données d'entraînement sur le frelon tout en ayant moins d'images de validation et de test par rapport au dataset d'insectes nuisibles. Sachant que les images qui composent le jeu d'entraînement du *Vespa Velutina* ont subies diverses transformations. Comme le montre la **Figure 11**, pour chaque image originale, nous disposons de deux variantes altérées selon certains critères¹⁹.



Figure 11 - Échantillons du dataset du frelon asiatique

À g. l'image originale, les deux suivantes ont subi des modifications de rotations et de rognage

Nous avons donc procédé à deux ajustements. Le premier consiste à réduire le jeu de données d'entraînement du frelon asiatique en supprimant les images issues dans l'augmentation de données. Il sera toujours possible de procéder à de l'augmentation d'image via diverses méthodes de *Keras*.

En deuxième temps, nous allons rééquilibrer les autres ensembles d'images de façon que chaque classe dispose du même nombre d'image pour les 3 ensembles (entraînement, validation, test). Nous avons rédigé un script *PowerShell* afin de retirer toutes les images dont les pixels d'au moins 1 des 4 coins disposent d'une valeur dans une nuance de gris allant de 0 à 127, c'est-à-dire les pixels dont les trois valeurs R, G et B sont identiques et inférieures ou égales à 127.

Procéder de la sorte a considérablement réduit le nombre d'image du dataset initial, mais nous permettra d'obtenir un équilibre dans les deux classes. De plus, les images altérées de *Vespa Velutina*

¹⁹ Rotation, luminosité, rognage, étirement, flou, etc...

auraient pu corrompre l'apprentissage en indiquant au modèle que les photos aux bords gris sont indicatrices de la présence de l'insecte. Pour l'anecdote, ce *dataset* était initialement prévu pour entraîner un modèle à réaliser de la détection d'objet. Ainsi, ces bordures foncées ne constituent pas une problématique dans ce scénario puisque le sujet est généralement délimité en dehors de ces zones « mortes ».

Finalement, nous avons supprimée des données du *dataset* complémentaire afin d'obtenir un équilibre parfait entre les deux classes. Le résultat final est représenté dans le **Tableau 4**.

Répartitions des jeux d'images		Nb images
Entraînement		5'350
Validation		1'535
Test		765
Total		7'650

Tableau 4 - Nombre d'image de chaque dataset après filtrage

3.2.3 Visualisation des données

Nous avons procédé à un étiquetage également via un script *Powershell*. En effet, cette étape c'est avéré nécessaire puisque le filtrage opéré précédemment a rendu inutilisable le fichier de label fournit avec le téléchargement du dataset. Nous avons ensuite visualisé nos données via un *dataframe* dans un *Notebook Python*, comme le montre la **Figure 12** afin de s'assurer du bon chargement de nos données.

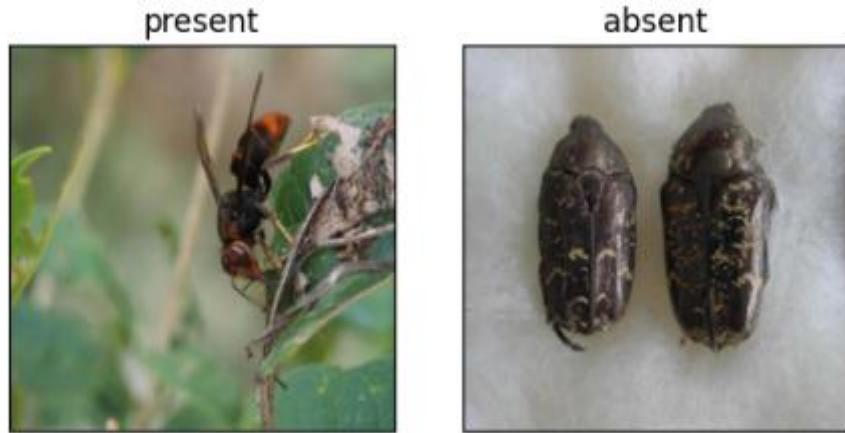


Figure 12 - Visualition des données dans le dataframe du notebook python

À g. une image de *Vespa Velutina* avec le label « present ». À d. une image de nuisible avec le label « absent »

3.2.4 Pré-processing

Pour entraîner le modèle dans les meilleures conditions, nous avons appliqué quelques transformations sur les données en entrées. L'ensemble de nos images sont sur des dimensions de 640x640 pixels. Nous avons donc, dans un premier temps, redimensionner ces images en 224x224 pour les adapter à l'entrée du modèle *MobileNetV3* qui, pour rappel, a été le modèle retenu lors du chapitre précédent.

Puisque nous avons éliminé les images altérées de notre dataset original, nous avons réappliqué ces procédés avec les outils que la librairie *Keras* met à disposition. Vous retrouverez les différentes modifications appliquées dans le **Code 1**.

```
1 data_augmentation = keras.Sequential([
2     preprocessing.RandomFlip("horizontal_and_vertical"),
3     preprocessing.RandomRotation(0.2),
4     preprocessing.RandomZoom(0.2),
5     preprocessing.RandomHeight(0.2),
6     preprocessing.RandomWidth(0.2),
7     preprocessing.RandomContrast(0.2)
8 ], name="data_augmentation")
```

Code 1 - Paramètres appliqués pour l'augmentation de données

Chaque altération du **Code 1** est une couche par laquelle une image va passer. Elle aura donc une chance d'être modifié sur sa symétrie axiale, sa rotation, sur son zoom, sa hauteur, sa largeur et son contraste. La valeur chiffrée « 0.2 » correspond aux bornes positives et négatives en pourcent de modification. Par exemple, concernant la hauteur, celle-ci sera augmentée ou diminuée de 20%.

3.2.5 Entrainement du modèle

Comme mentionné dans le chapitre précédent, nous avons procédé à de l'apprentissage par transfert afin de simplifier l'entraînement, d'en réduire son temps tout en augmentant sa précision globale. Le **Code 2** présente notre initialisation du modèle *MobileNetV3*.

```
1 pretrained_model = MobileNetV3Large(
2     weights='imagenet',
3     include_top=False,
4     input_shape=(224, 224, 3),
5     pooling='avg',
6 )
7
8 pretrained_model.trainable = False
```

Code 2 - Instanciation du modèle pré-entraîné

Les différents paramètres nommés ont les effets suivants :

- **weights** : configure les poids du modèle pour leur affecter une valeur initiale. Ici, on assigne les valeurs des poids obtenus après entraînement sur le dataset *ImageNet*
- **include_top** : définit si oui ou non nous incluons la couche de sortie du modèle (les couches de classification). Puisque nous utilisons ce modèle pour lui attribuer une nouvelle tâche, nous devons retirer cette couche.
- **input_shape** : définit les dimensions d'entrée du modèle. Ici, l'image doit être de dimensions 224x224 et chaque pixel doit être encodé sur 3 valeurs numériques distinctes (canal RGB).
- **pooling** : indique quelle méthode est utilisée pour la mise en commun des valeurs des pixels après la dernière couche de convolution. Ici, la moyenne est appliquée.

La ligne 8, quant à elle, indique que nous ne souhaitons pas entraîner ce modèle. Par cette opération, nous gelons les poids de cette architecture et ils ne pourront plus être modifiés.

À cette architecture, nous avons rajouté en entrée les différentes couches d'augmentation de données décrites dans le **Code 1**, puis nous avons rajoutée deux couches entièrement connectées de 128 neurones chacune. À la sortie de ces deux couches, nous avons ajouté deux *Dropout* de 20%. Cela signifie que chaque valeur des neurones en sortie de ces couches a 20% de chance d'être ignorée. Ce procédé permet d'éviter le surapprentissage notamment dans un contexte où nous disposons de peu de données. Finalement, en fin d'architecture, nous avons ajouté notre unique couche de sortie à laquelle il est important de définir une fonction d'activation sigmoïde afin de disposer de valeurs interprétables dans un ensemble entre 0 et 1.

Les derniers paramètres du modèle que nous avons appliqué sont le taux d'apprentissage défini à 0.00001 car nous disposons d'une architecture largement entraînée et qui aura juste besoin d'être affinée, ce qui ne sera pas possible avec un taux d'apprentissage trop élevé car ce dernier risquerait de manquer le minimum de la fonction de coût. Nous avons défini un nombre d'*epochs* max à 100, mais nous avons toutefois pris soin de définir une fonction de rappel après chaque *epoch* qui vérifie si la valeur de la fonction de coût continue de diminuer. Si cette dernière n'a pas diminué depuis 3 *epochs*, alors cela peut indiquer que notre modèle commence à effectuer du surapprentissage sur les données d'entraînements. Le dernier paramètre que nous avons attribué est celui de la fonction de coût qui, dans le cas d'une classification binaire, sera de type *binary cross-entropy*.

Nous détaillerons dans le chapitre suivant les résultats obtenus par suite de l'entraînement de notre modèle.

3.3 Implémentation de l'application de démonstration

Avec la revue de notre cahier des charges, nous avons également revu l'application de démonstration pour que cette dernière mette en avant les différentes options que mettrons à disposition le *Dart package* réalisé.

Notre idée est donc de présenter une interface visuelle simple permettant de sélectionner à la volée entre plusieurs types de modèles *deep-learning*, dont celui sur *Vespa Velutina*, et les différentes options, comme l'utilisation du GPU ou le nombre de threads à utiliser.

Une fois les options sélectionnées, nous pouvons sélectionner une image à envoyer au modèle soit depuis la galerie du téléphone, soit en prenant directement une photo. Une fois l'image sélectionnée/prise, elle sera transmise à notre *package* où le modèle se chargera de réaliser la prédiction et d'y associer le label correspondant. Notre application affichera ainsi l'image sélectionnée en overlay avec la prédiction et le label qui lui ont été attribués.

La **Figure 13** présente ainsi les écrans de notre application. Notez que les différents types de paramètres et ce sur quoi ils influent sont détaillés dans la section traitant de l'implémentation de notre *package*.

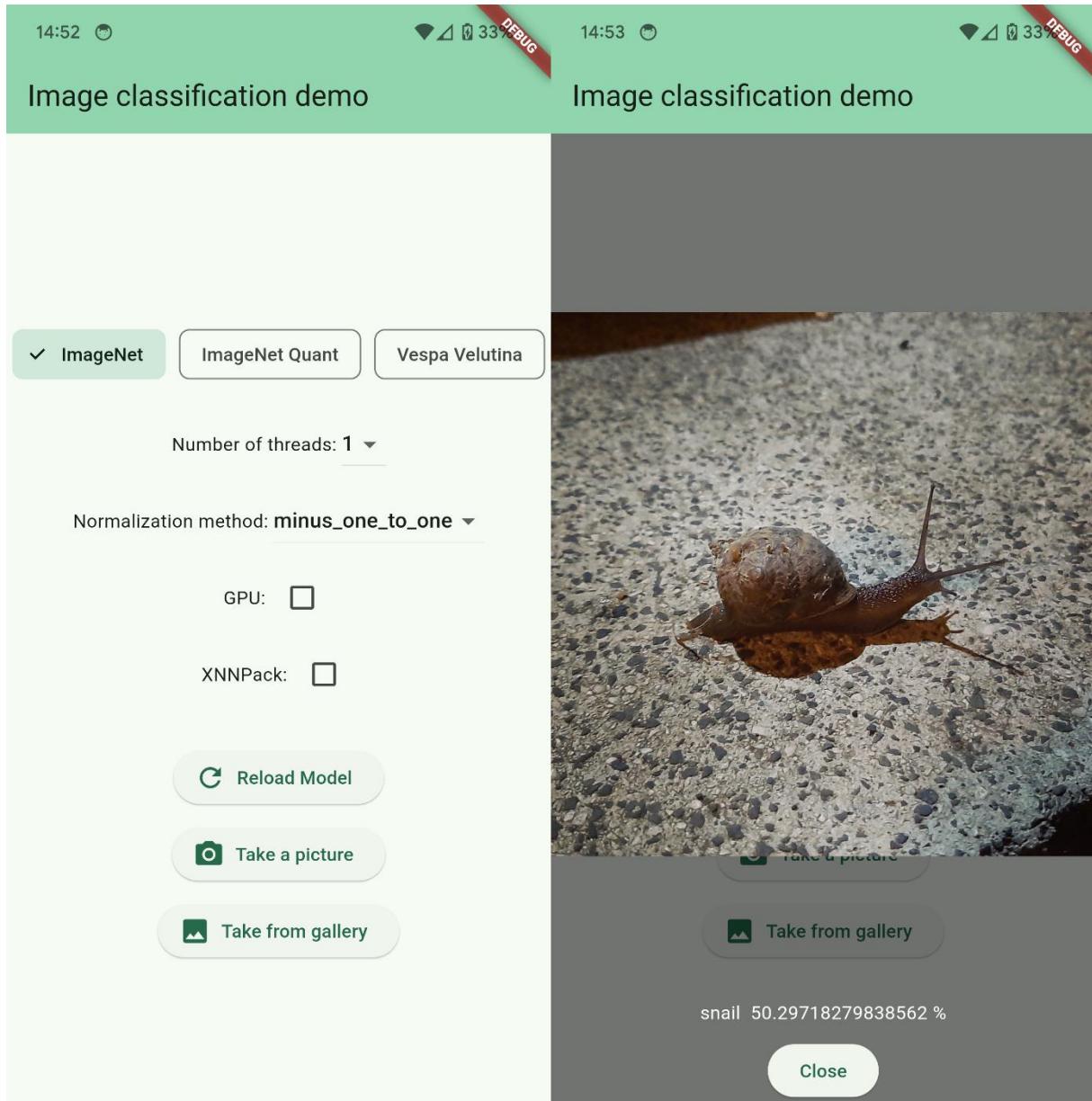


Figure 13 - Application de démonstration sous Android

À g., l'écran d'accueil avec les différents paramètres. À d. l'overlay du résultat après inférence d'une image depuis la galerie. Notez qu'ici, l'application intègre déjà notre package, d'où la prédiction et le label présent sur l'image.

Pour cette application, nous avons chargé trois modèles différents que vous retrouvez en première ligne de l'application sur la **Figure 13**. Le premier modèle est une architecture *MobileNetV3* entraîné sur le *dataset ImageNet*. Il s'agit du même modèle utilisé lors du prototypage. Le deuxième modèle est également entraîné sur *ImageNet*, mais il s'agit cette fois-ci d'une architecture *MobileNetV1* où les poids ont été quantifié afin d'ôter les calculs à base de virgule flottante. Nous avons choisi ce modèle afin de mettre en avant la flexibilité de notre *package* à traiter aussi bien des modèles avec des poids entiers ou à virgule. Le dernier modèle n'est autre que celui que nous avons entraîné pour détecter le *Vespa Velutina*. Ces trois modèles sont tous contenu dans le dossier *assets* du projet *Flutter*.

Un modèle est chargé à la pression d'un des trois choix présentés en haut à droite de la **Figure 13**. Afin de ne pas alourdir l'application en rechargeant le modèle à chaque modification de paramètre, nous avons introduit un bouton « *Reload Model* » dont le but est de libérer l'ancienne instance du modèle choisi pour en recréer une avec les paramètres sélectionnés.

3.4 Implémentation du *Dart Package*

Puisque nous disposons déjà d'un *package*²⁰ permettant d'inférer une image dans un modèle *TensorFlow Lite*, nous avons orienté le développement de notre *package* comme une surcouche offrant une API haut niveau pour permettre la classification d'image. Par conséquent, cela spécialise le package officiel de *TensorFlow* à une seule tâche, mais facilite son utilisation en y introduisant une abstraction supplémentaire.

3.4.1 Contexte

Afin de mieux comprendre notre approche, il est nécessaire de revenir sur le fonctionnement même du *package* initial : *tflite_flutter*. Son approche, certes fonctionnelle, est assez directe et peut nécessiter un temps de développement additionnel non négligeable. Pour comprendre pourquoi, il suffit de regarder le code d'exemple fourni par la documentation pour l'utilisation de cet outil.

```
1 final interpreter = await Interpreter.fromAsset('assets/your_model.tflite');
2
3 // For ex: if input tensor shape [1,5] and type is float32
4 var input = [[1.23, 6.54, 7.81, 3.21, 2.22]];
5
6 // if output tensor shape [1,2] and type is float32
7 var output = List.filled(1 * 2, 0).reshape([1, 2]);
8
9 // inference
10 interpreter.run(input, output);
11
12 // print the output
13 print(output);
```

Code 3 - Snippet d'utilisation du package *tflite_flutter*

Le défi principal réside à la ligne 4 du **Code 3**. En effet, *tflite_flutter* part du principe que nous fournissons d'ores et déjà un input formatté correctement, c'est-à-dire une image représentée sous la forme d'une matrice de valeur numérique. Le *package* ne fournit aucune méthode de pré-traitement des données, ce qui implique que le code réalisant ses tâches doit se trouver dans l'application installant cette dépendance, ou dans un autre *package*.

Du reste, il existait déjà un *package* mettant à disposition divers utilitaire permettant de traiter des images afin de pouvoir les fournir à *tflite_flutter*. Ce dernier se nommait *tflite_flutter_helper*, mais est malheureusement annoté « *Discontinued*²¹ », le rendant obsolète sur les dernières versions de *Flutter*. Quant au *Github*²² du projet, ce dernier n'est plus maintenu depuis plus de 2 ans.

Afin de palier à ce manque, la documentation de *tflite_flutter* mentionne l'utilisation d'un autre *package* offrant des fonctionnalités similaires se basant sur le *MediaPipe Solutions* de *Google*. Ce produit est un ensemble de librairies et d'outils simplifiant l'utilisation de l'intelligence artificielle sur diverses plateformes, aussi bien mobiles que sur le web. La promesse étant de permettre l'utilisation de modèle *deep learning* en écrivant un minimum de code.

²⁰ Lien pub.dev de *tflite_flutter* : https://pub.dev/packages/tflite_flutter

²¹ Lien pub.dev du package : https://pub.dev/packages/tflite_flutter_helper

²² Lien du Github de *tflite_flutter_helper* : https://github.com/am15h/tflite_flutter_helper

Bien que cette solution soit documentée sur le site officiel de *MediaPipe*²³ pour *Android* et *iOS*, force est de constater qu'il n'est nullement fait mention d'une solution pour *Flutter* ni même un autre *framework cross-platform*. Pire encore, un simple coup d'œil sur le répertoire *Github*²⁴ dédié à l'implémentation *Flutter* de *MediaPipe*, nous indique que très peu de tâches sont actuellement supportée. La **Figure 14** indique les tâches disponibles au moment de la rédaction de ce travail.

Supported Tasks

Task	Android	iOS	Web	Windows	macOS	Linux
Text						
Classification	✓	✓	-	-	✓	-
Embedding	✓	✓	-	-	✓	-
Language Detection	✓	✓	-	-	✓	-
GenAI						
Inference	✓	✓	-	-	✓	-
Audio						
Vision						

Figure 14 - Liste des tâches supportées par *MediaPipe Flutter*²⁴

Consulté pour la dernière fois en septembre 2024. Nous pouvons constater que seuls les tâches en lien avec le traitement des textes et la génération par IA sont disponibles.

C'est dans ce contexte-ci que nous avons fait le choix de développer nous-même notre *package* utilitaire et, ce faisant, de redéfinir les priorités de notre projet.

Le développement de ce *package* repose également sur certaines limitations rencontrées lors du développement de notre prototype. Puisque *tflite_flutter* s'attend à une entrée formatée correctement, cela implique que chaque application voulant intégrer un modèle *deep-learning* doit réimplémenter sa logique de redimensionnement d'image, de normalisation des pixels et d'association des labels avec chaque prédiction en sortie. Même si cette solution est faisable, elle peut alourdir le code de l'application et ne correspond pas à notre volonté de proposer une solution générale *cross-platform*.

Avec ce *package*, nous simplifions l'utilisation de modèles de classification d'image permettant ainsi aux applications consommatrices de se focaliser sur la logique qui les concerne directement.

3.4.2 Implémentation des interfaces

Pour commencer, l'implémentation d'un *Dart Package* se doit respecter quelques conventions définies par *Dart* dans l'arborescence de fichier. C'est cette dernière qui permet de déterminer à quelles classes et fonctions le consommateur aura accès. Seuls les fichiers se trouvant dans le répertoire *lib* seront exportés, comme le montre la **Figure 15**.

²³ Lien vers *MediaPipe Solutions* : <https://ai.google.dev/edge/mediapipe/solutions/guide>

²⁴ Lien du *Github* de *flutter_mediapipe* : <https://github.com/google/flutter-mediapipe>

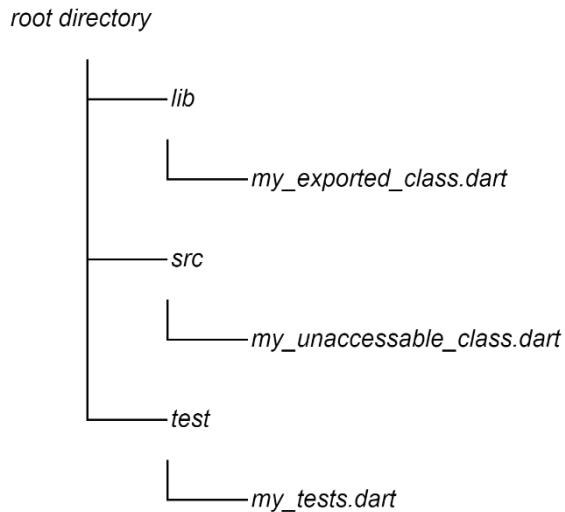


Figure 15 - Convention d'arborescence pour un Dart package

Dans notre cas, nous avons identifié 3 structures essentielles devant être exposées à nos consommateurs. Tout d'abord, une classe utilitaire disposant de deux méthodes principales, la première permettant l'initialisation des propriétés de ladite classe (un constructeur), puis une seconde réalisant l'inférence du modèle sur une image donnée en paramètre. Cette classe est le point d'interaction entre le consommateur et notre *package*, et devra être initialisé avec le modèle de *deep learning*, le fichier de labels, et les options d'inférence.

La seconde structure sera une classe façade qui englobe plusieurs paramètres permettant de définir les préférences d'exécutions du modèle, comme le nombre de threads alloués ou encore si le modèle doit préférer l'utilisation d'un processeur graphique. Ces options sont celles fournies par le *package tflite_flutter*, nous devons donc permettre leur accès sans obliger nos consommateurs de dépendre directement de ce *package*-ci. La création d'une classe pour englober ces options nous a semblé pertinente puisque ces paramètres sont essentiellement un ensemble de type primitif, et les combiner dans un seul élément offre une meilleure lecture et utilisation lorsqu'ils doivent être passer en paramètre d'une autre fonction.

La troisième structure repose simplement sur une énumération de différentes méthodes de normalisation des valeurs des pixels d'une image. Les options que nous avons décidé de traiter sont en lien avec les fonctions d'activations courantes utilisées dans les modèles de classification d'image, comme la sigmoïde ou la tangente hyperbolique. Elles normalisent respectivement les valeurs entre 0 et 1 ou -1 et 1. Nous avons également permis de ne pas normaliser les données, ce qui peut être utile dans le cas de modèles quantifiés ou les calculs s'effectuent avec des nombres entiers. La création de cette énumération découle d'un obstacle rencontré lors du prototypage de notre solution dans le chapitre précédent. En effet, le modèle *MobileNet* pré-entraîné avec *ImageNet* disposait de valeurs normalisées entre 0 et 1, mais notre prototype ne procédait à aucune normalisation, si bien que le résultat en sortie du modèle n'était pas du tout interprétable par notre application. En normalisant les pixels de l'image en entrée, nous avions ainsi pu obtenir le résultat attendu.

À ce stade de nos explications, nous disposons donc d'une structure comme décrite par la **Figure 16**.

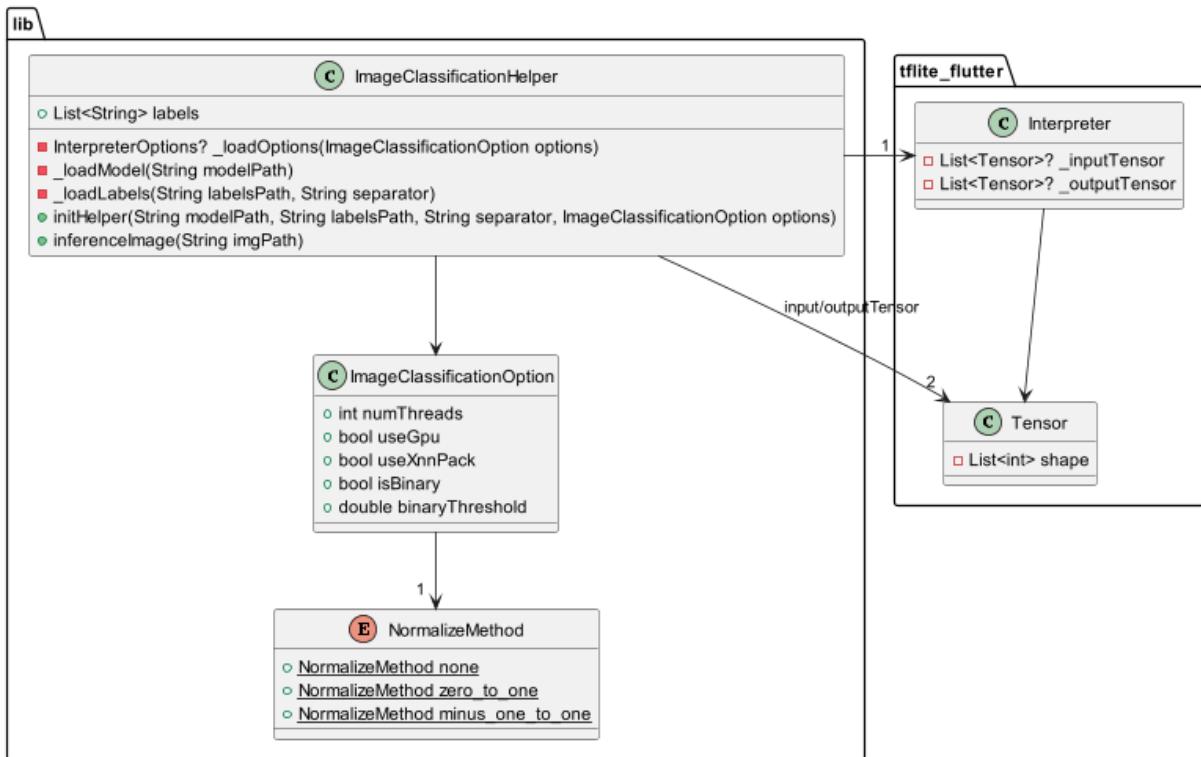


Figure 16 - Diagramme de classe de la partie exposée du Dart package

Les getters et setters ne sont pas représentés dans ce diagramme. Ci-dessus est figuré l'ensemble des structures dans le dossier `lib`. Ceci constitue l'ensemble des éléments accessibles depuis les apps consommatrices

La classe `ImageClassificationHelper` dans la **Figure 16** dispose de deux propriétés de type `Tensor` et une autre de type `Interpreter`. Les deux premières définissent une structure englobante permettant les interactions avec l'API de `TensorFlow`, la deuxième représente le modèle de réseau de neurones. Il s'agit de types fournis avec le package `tflite_flutter` représenté à droite sur la **Figure 16**. La méthode `_loadOptions` retourne un type `InterpreterOptions` qui est une structure également disponible dans `tflite_flutter`. Elle dispose de diverses options configurables paramétrant l'inférence du modèle, comme le nombre de thread et l'utilisation d'un processeur graphique. Ainsi, notre classe `ImageClassificationOption` est la façade de `InterpreterOptions`, comme nous l'avions évoqué plus tôt.

Pour faciliter l'initialisation de la classe `ImageClassificationHelper`, nous fournissons la méthode `initHelper` qui prendra en paramètre les chemins du modèle et du fichier de label. Pour ces derniers, il pourra être fourni un caractère de séparation indiquant au *package* comment chaque classe est séparée dans le fichier, la séparation par défaut étant le retour à la ligne (`\n`). Si le consommateur le désire, il pourra fournir les options souhaitées pour l'inférence.

Si nous nous attardons un peu plus sur la classe `ImageClassificationOption`, voici un descriptif des différents paramètres :

- **`numThreads`** : Définit le nombre de threads alloué pour l'inférence du modèle. Par défaut, seul un thread est alloué.
- **`useGpu`** : Ce booléen permet de déterminer à quelle unité de calcul l'inférence sera déléguée. Si ce dernier est défini à `true`, alors l'inférence sera exécutée sur le processeur graphique. Par défaut sa valeur est `false`.

- **`useXnnPack`** : Permet l'utilisation de *XNN Pack*, une solution elle aussi développée par *Google* permettant d'optimiser l'inférence des modèles utilisant des virgules flottantes sur diverses architecture de processeur²⁵. Par défaut, sa valeur est à *false*.
- **`normalizeMethod`** : Définit comment les valeurs numériques de chaque pixel de l'image seront normalisées en entrée du modèle. Par défaut, aucune normalisation n'est appliquée.
- **`isBinary`** : Définit si le modèle doit être interprété comme une classification binaire à une seule sortie. La valeur par défaut est *false*.
- **`binaryThreshold`** : Définit la valeur seuil à laquelle bascule la classification binaire sur la première ou deuxième classe. Si la prédiction est inférieure au seuil, la première classe lui sera attribuée. Dans le cas contraire, c'est la deuxième classe qui est attribuée. Par défaut, le seuil est fixé à 0.5. Cette valeur est ignorée si *isBinary* est *false*.

Les deux propriétés *isBinary* et *binaryThreshold* sont nées d'un besoin que nous avions pour notre modèle de classification du frelon asiatique. En effet, notre modèle ne dispose que d'une seule sortie indiquant la probabilité que l'image traitée contienne l'insecte ou non. Malgré le fait qu'il n'y ait qu'une seule sortie, nous souhaitions tout de même associer deux labels en sortie de modèle, et nous avions donc besoin d'informer notre *package* de cette configuration afin que ce dernier puisse associer le bon label en se basant sur le seuil (*binaryThreshold*) fourni. Nous avons ajouté le booléen *isBinary* pour distinguer facilement ce cas particulier. Les deux paramètres sont optionnels et disposent de valeur par défaut s'ils sont omis. Ces deux propriétés offrent ainsi une simplification d'intégration de modèles binaires à une sortie dans notre *package*, peu importe le modèle ou les labels associés.

Une fois le modèle, les labels et les options chargées, le consommateur pourra appeler la méthode *inferencelimage* à laquelle sera fourni un chemin vers l'image à traiter. Nous avons préféré l'envoi d'un chemin plutôt que d'un fichier pour déléguer la responsabilité d'interprétation du format de fichier au *package*. Cela simplifie l'utilisation pour le consommateur et l'allège de potentielle dépendance utilisée par notre *package*.

Avant d'être envoyé au modèle, l'image sera donc reconstruite depuis le chemin fourni. Ici, nous introduisons l'utilisation d'une autre dépendance au *package image*²⁶. Ce dernier permet d'ajouter un niveau d'abstraction supplémentaire à un fichier en facilitant l'accès aux propriétés des images tel que la hauteur et la largeur et aussi des valeurs RGB de chaque pixel. Cette abstraction simplifie la manipulation de l'image, notamment dans le processus de normalisation des valeurs que nous aborderons dans la prochaine sous-section.

3.4.3 Implémentation des fonctions internes

Les éléments mentionnés jusqu'à présent constituent donc les différentes interfaces qu'une application *Flutter* pourra utiliser pour interagir avec notre *package*. Nous allons maintenant approfondir l'implémentation du code qui n'est pas exposé aux consommateurs.

L'aspect principal que nous avons gardé en tête pendant la réalisation de ce package, est qu'un modèle de réseau de neurone nécessite un temps d'exécution qui est mesurable. Sans savoir à l'avance la durée exacte de l'inférence, nous devions proposer une solution ne bloquant pas le thread principal de l'application et évitant ainsi l'impression que l'app ne réponde plus. Typiquement, cet aspect avait complètement été ignoré lors de la réalisation de notre prototype.

3.4.3.1 Asynchronisme et threads

En *Dart*, il y a deux concepts clés à saisir pour comprendre l'ordre d'exécution des événements et la concurrence : l'***event loop*** et les ***Isolates***.

²⁵ Pour plus d'informations : <https://blog.tensorflow.org/2020/07/accelerating-tensorflow-lite-xnnpack-integration.html>

²⁶ Lien vers le *package image* : <https://pub.dev/packages/image>

Le premier se définit comme une file d'attente d'évènement devant être traitée par le programme, tels que le rafraîchissement de l'interface graphique, ou la gestion de pression d'un bouton. Chaque évènement est traité dans l'ordre qu'il a été inséré dans la file, et ils sont tous traité les uns après les autres, si bien qu'un évènement long à traiter donnera l'impression que notre application ne répond plus aux interactions. L'*event loop* introduit également la programmation asynchrone en Dart, la **Figure 17** permet d'illustrer au mieux ce mécanisme.

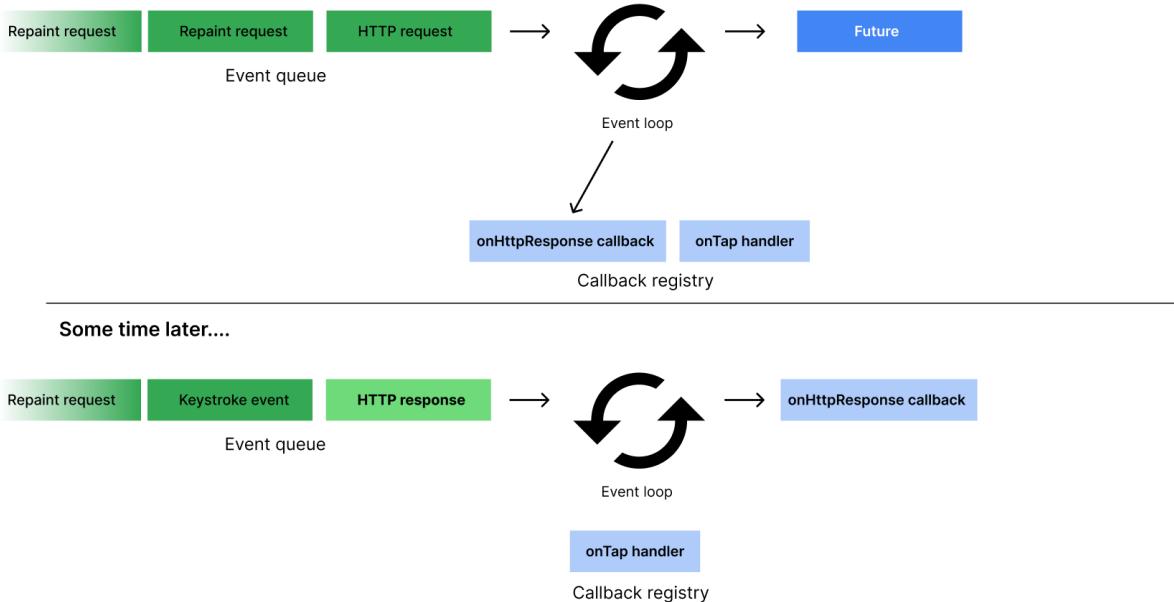


Figure 17 - Représentation de l'*event loop* de Dart^[5]

En vert, la file des évènements à traiter (de d. à g.). En bleu clair, les évènements asynchrones qui sont envoyé sur un registre de rappel pour ne pas bloquer la file principale. En bas, l'évènement `onHttpResponse` est remis dans la file une fois que l'exécution a été terminée

Le second concept, les *Isolates*, introduit la programmation concurrente. Un *Isolate* peut s'apparenter à un thread, hormis que les *Isolates* ne disposent pas de contexte global partagé, ce qui les rendent robustes aux *data race*²⁷. En contrepartie, le seul moyen qu'on les *Isolates* de communiquer entre eux est via les *Ports* (respectivement *ReceivePort* et *SendPort*). Ce sont deux canaux unidirectionnels dans lesquels des messages peuvent être envoyés à tout moment. Chaque *Isolate* dispose de sa propre *event loop*, ce qui permet aux *Isolates* créés d'effectuer des opérations bloquantes sans affecter l'exécution du programme principal, ce qui est montré sur la **Figure 18**.

²⁷ Évènement se produisant lorsque plusieurs threads tentent d'accéder à une ressource partagée mal ou pas protégée, menant à des comportements non prévisibles ou indéfinis.

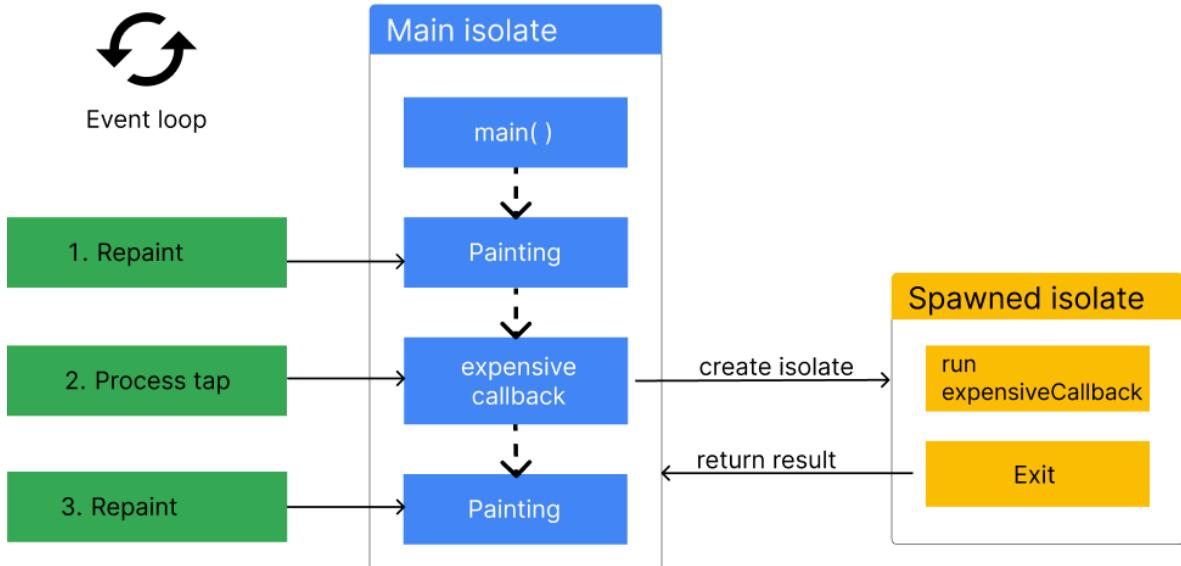


Figure 18 - Schéma des *Isolates* en Dart^[5]

En vert, les évènements de l'event loop. En bleu, le *Isolate* principal avec de haut en bas les différentes opérations effectuées.
En jaune le second *Isolate* créé par le principal.

Les deux concepts présentés ci-dessus nous offrent un ensemble d'outil pour répondre à notre besoin initial. L'inférence de modèles étant une opération pouvant être coûteuse pour les processeurs, nous avons décidé d'orienter une implémentation profitant au mieux des architectures multi cœurs en utilisant les *Isolates*.

3.4.3.2 Isolate d'inférence

⚠ À ce stade du projet, nous avons commis une erreur importante à souligner. Bien que nos intentions d'utiliser les *Isolates* étaient nobles, le package *tflite_flutter* met déjà à disposition une méthode permettant d'inférer un modèle au sein d'un *Isolate*. Les paragraphes suivants évoquent malgré tout cette implémentation superflue, mais elle a été par la suite abandonnée au profit de celle mise à disposition par *tflite_flutter*. Nous en reparlerons plus tard.

Dans un premier temps, nous devrons donc initialiser l'*Isolate* responsable de l'inférence. Une fois créé, les *Ports* d'un *Isolate* restent ouverts et utilisables tant que ce dernier n'a pas reçu de commande l'indiquant qu'ils devaient se fermer ou tant que l'*Isolate* n'a pas terminé d'exécuter la fonction qui lui était demandée.

Nous aurions pu utiliser un *Isolate* dans une *closure* (fermeture ou clôture en français), comme dans l'exemple suivant dans le **Code 4**. Cela nous aurait facilité la gestion de son instance puisqu'elle aurait été déléguée au *garbage collector*. L'*Isolate* aurait simplement exécuté le bloc de code nécessaire. Toutefois, procéder de la sorte n'est pas optimal si le code appelant la création à usage unique du *Isolate* est appelé souvent. En effet, la création d'un *Isolate* n'est pas gratuite, puisqu'elle implique une copie du contexte d'exécution du programme. Ainsi, créer une instance unique et réutilisable permet de couvrir le cas où l'inférence serait utilisé de nombreuses fois d'affilé.

```

1 const String filename = 'with_keys.json';
2
3 final jsonData = await Isolate.run(() async {
4     final fileData = await File(filename).readAsString();
5     final jsonData = jsonDecode(fileData) as Map<String, dynamic>;
6     return jsonData;
7 });

```

Code 4 - Exemple d'Isolate avec closure

Ici, la méthode `run` permet d'exécuter directement le code dans un `Isolate` séparé, elle prend un paramètre une fonction retournant un type `Future` (asynchronisme), d'où l'usage du mot clé « `async` » et « `await` ». L'`Isolate` est à usage unique, il est ensuite nettoyé par le `garbage collector`.

L'implémentation du `Worker` responsable d'exécuter l'inférence s'est majoritairement inspirée de l'exemple fournit dans la documentation de `Dart`²⁸ qui expose pas à pas un code robuste. Nous vous invitons à consulter cette documentation si vous souhaitez approfondir les différents détails. Ici, nous nous contenterons de synthétiser les points essentiels.

La **Figure 19** peut être consultée en parallèle pour mieux suivre les liens entre nos différentes classes. Dans notre `package`, dans le dossier `src`, nous allons créer une nouvelle classe `InferenceWorker` qui contiendra l'ensemble des méthodes gérant le `Isolate` responsable de l'inférence du modèle. Toute classe gérant un `Isolate` se doit de fournir un ensemble de méthodes.

En premier, le `Worker` offrira une méthode, souvent appelée `spawn`, permettant l'initialisation de l'`Isolate`, en prenant soin de stocker dans des attributs privés le port d'envoi et de réception pour communiquer avec ce dernier.

Pour faciliter la communication entre ce nouveau `Isolate` et le programme principal et ainsi éviter la manipulation directe avec les ports d'envoi et de réception, le `Worker` peut mettre à disposition une méthode publique qui se chargera d'exécuter la fonction principale attendue. Dans notre cas, cette méthode sera nommée `InferenceImage`.

La dernière méthode à exposer doit offrir à la classe utilisant le `Worker` la possibilité de fermer les ports du nouveau `Isolate` et terminer son processus. Cette méthode est généralement nommée `close`.

L'`InferenceWorker` que nous avons implémenté est toujours visible sur la **Figure 19** (dans la section `src`). Cependant, comme nous l'avions annoncé en encadré au début de cette sous-section, nous avons fini par retirer cette solution au profit de celle fournie directement par le `package tflite_flutter`. Le **Code 5** illustre comment s'utilise cette nouvelle solution.

```

1 final interpreter = await Interpreter.fromAsset('your_model.tflite');
2 final isolateInterpreter =
3     await IsolateInterpreter.create(address: interpreter.address);
4 await isolateInterpreter.run(input, output);

```

Code 5 - Utilisation des isolates de `tflite_flutter`

L'implémentation choisie par le package repose sur une classe englobant l'`Interpreter` créé à partir du fichier `tflite`.

Même si notre implémentation aura été superflue puis inutilisée, elle nous aura permis de comprendre le fonctionnement derrière la classe `IsolateInterpreter`. En effet, en inspectant le code de ce `package`, nous pouvons établir des équivalences avec notre implémentation. Nous y retrouvons le `Worker`

²⁸ Lien vers le code d'exemple : <https://dart.dev/language/isolates#robust-ports-example>

nommé ici *IsolateInterpreter*, avec les trois méthodes susmentionnées ; celle de l'initialisation de l'*Isolate* (*create*), celle exécutant le code fonctionnel principal (*run*), et une troisième de fermeture disponible mais non représentée dans le **Code 3** (*close*).

3.4.3.3 *InferenceModel*

Puisque les *Isolates* doivent communiquer entre eux au travers de canaux, nous avons décidé de créer une nouvelle classe *InferenceModel* regroupant l'ensemble des informations nécessaire à l'inférence du modèle, mais également au pré-traitement de l'image reçue par le *package* (notamment la méthode de normalisation) ainsi que les paramètres permettant d'associer les prédictions du modèle aux labels de chaque classe. C'est donc une instance de cette classe qui sera envoyé à l'*Isolate* se chargeant.

3.4.3.4 Classes utilitaires

Le dernier choix que nous avons effectué consiste en la façon dont nous traitons la normalisation des pixels de l'image d'entrée, et la façon dont nous allons associer les labels avec les prédictions obtenues. Pour de meilleures lisibilités et pour tester ces fonctions plus facilement, nous avons décidé de créer deux classes utilitaires.

Pour avoir une meilleure compréhension de notre architecture des fonctions internes et non exposées de notre *package*, nous avons réalisé un diagramme présenté dans la **Figure 19**.

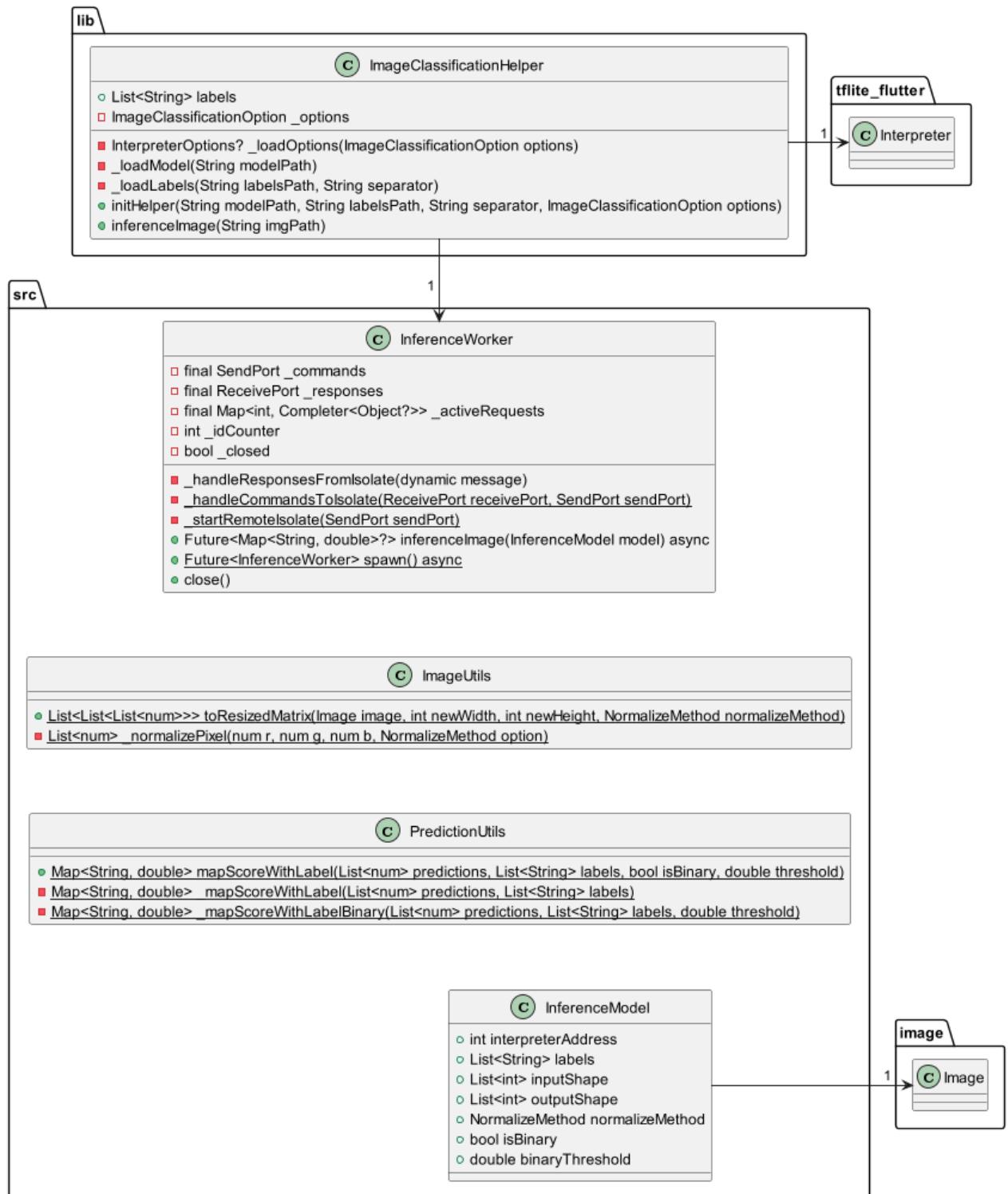


Figure 19 - Diagramme de classe des fonctions internes

Les classes du répertoire lib ont été en partie omises de ce schéma pour plus de lisibilité. Pour rappel, le répertoire src n'est pas exposé aux consommateurs. En bas à droite est visible la dépendance avec le package image

Passons en revue l'implémentation de nos classes utilitaires à commencer par *ImageUtils*. C'est dans cette classe que l'image passée au modèle sera redimensionnée au format attendu par celui-ci, puis elle sera transformée en liste de valeur numérique afin que nous puissions utiliser l'*Interpreter* de *tflite_flutter*. Lors de cette numérisation, nous appliquerons également la méthode de normalisation choisie par le consommateur.

C'est pour l'opération de redimensionnement que nous avons choisi d'utiliser le *package image*. Nous avions d'ores et déjà utilisé ce dernier lors de la réalisation de notre prototype. Ce package met à disposition une méthode *copyResize* à laquelle il suffit de fournir les nouvelles dimensions. Les dimensions attendues par le modèle sont stockées dans notre objet *InferenceModel* dans les propriétés *inputShape* et *outputShape*.

La différence avec notre prototype est qu'en plus de récupérer les valeurs numériques des pixels de l'image, nous allons appliquer une transformation de normalisation dépendant de l'option choisie. Le **Code 6** expose ces différents calculs. La finalité des méthodes de cette classe utilitaire et de transformer une image donnée en matrice redimensionnée (ou tenseur) de valeurs RGB pour chaque pixel de l'image.

```

1  /// Normalize pixel values based on the given option.
2  static List<num> _normalizePixel(
3      num r, num g, num b, NormalizeMethod option) {
4      switch (option) {
5          case NormalizeMethod.none:
6              return [r, g, b];
7          case NormalizeMethod.zero_to_one:
8              return [(r / 255), (g / 255), (b / 255)];
9          case NormalizeMethod.minus_one_to_one:
10             return [
11                 (r / 127.5) - 1,
12                 (g / 127.5) - 1,
13                 (b / 127.5) - 1,
14             ];
15     }
16 }
```

Code 6 - Méthodes de normalisation par pixel

La deuxième classe utilitaire est celle qui se chargera d'associer les prédictions aux labels donnés. Elle va simplement retourner cette association sous la forme d'une *Map<String, num>*. La légère particularité réside dans le traitement qui diffèrera si le mapping se fait sur un modèle binaire à sortie unique ou non. Le cas « binaire » va associer le premier label de la liste si la prédiction est inférieure au seuil donné et le deuxième label dans le cas contraire.

Dans le cas général (non binaire), l'association se fera en tenant compte de la collection la plus petite entre celle des prédictions et celles des labels. Si les collections diffèrent en tailles, certaines associations seront absentes de la *Map* résultante. Ce choix permet d'éviter de lever des exceptions inutiles. Attention toutefois, l'association entre labels et prédiction se fait dans l'ordre des sorties du modèles. Ainsi la première sortie du modèle est associée au premier label dans la liste et ainsi de suite.

Dans le cas d'un mapping pour modèle binaire à sortie unique, nous avons ajouté quelques vérifications sur les paramètres reçus dans le but de mieux informer le consommateur sur cette implémentation qui n'est pas forcément explicite à première vue. Le **Code 7** présente ces conditions particulières.

```

1     static Map<String, double> _mapScoreWithLabelBinary(
2         List<num> predictions, List<String> labels, double threshold) {
3             if (predictions.length != 1) {
4                 throw ArgumentError('Binary classification only supports one
5                     output.');
6             }
7             if (labels.length != 2) {
8                 throw ArgumentError('Binary classification requires exactly two
9                     labels.');
10            }
11            // Set classification map {label: points}
12            var classification = <String, double>{};
13            // Associate the prediction to the correct label
14            var labelIndex = predictions[0] > threshold ? 1 : 0;
15            classification[labels[labelIndex]] = predictions[0].toDouble();
16
17            return classification;
18        }

```

Code 7 - Association prédiction - label dans un cas binaire

Chapitre 4

Résultats obtenus

Les but de ce chapitre est de présenter les résultats obtenus lors de ce travail à l'aide de diverses métriques. D'une part il s'agira de mettre en avant les performances obtenues après l'entraînement de notre modèle, puis de présenter en quelques chiffres les performances de notre *Dart Package* en portant un regard critique sur ceux-ci.

4.1 Validation du modèle

Étant donné notre volonté de ne pas approfondir la réalisation du modèle, nous avons procédé à une évaluation sommaire de ce dernier afin essentiellement de confirmer que l'entraînement réalisé a été bénéfique à l'obtention d'un modèle fonctionnel.

L'entraînement du modèle a été réalisé dans un *Jupyter Notebook*, ce qui nous a permis de réaliser plusieurs graphiques permettant l'observation de l'apprentissage du modèle au fil des *epochs*. La **Figure 20** présente l'évolution de la réduction de la fonction de coût au fur et à mesure que l'apprentissage progresse.

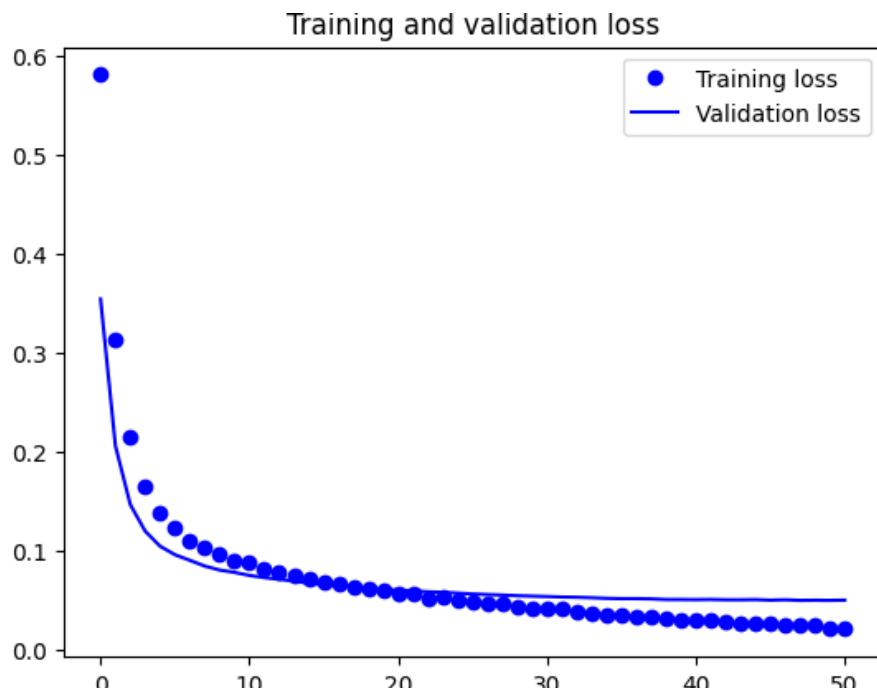


Figure 20 - valeur de la fonction de coût au fil des epochs de l'apprentissage

En abscisse : le nombre d'epochs. En ordonnée : la valeur de la fonction de coût

Cette courbe nous indique que l'apprentissage s'est bien déroulé avec une valeur de coût diminuant au fil des itérations. Le courbe d'entraînement et de validation sont relativement confondues. Cela implique que l'apprentissage des caractéristiques via le jeu d'entraînement ont été pertinente pour aider à mieux prédire le jeu de validation.

L'entraînement s'est arrêté après 51 epochs, car comme le montre la **Figure 20**, c'est à ce moment que la courbe de validation diverge de celle d'entraînement. Cela indique que le modèle commence à réaliser du surapprentissage sur les données d'entraînement et perd de plus en plus la capacité à prédire des données génériques.

Cet entraînement nous a permis d'obtenir des résultats au-delà de nos espérances, puisqu'après validation sur notre set de test, nous avons obtenu un score avoisinant le 99% de précision pour les deux classes. Le **Tableau 5** présente les différents résultats selon la classe, la **Figure 21** présente la matrice de confusion du modèle.

	Précision	Rappel	F-score
absent	0.984395	0.989542	0.986962
present	0.989488	0.984314	0.986894

Tableau 5 - Score du modèle sur le set de test par classe prédictive

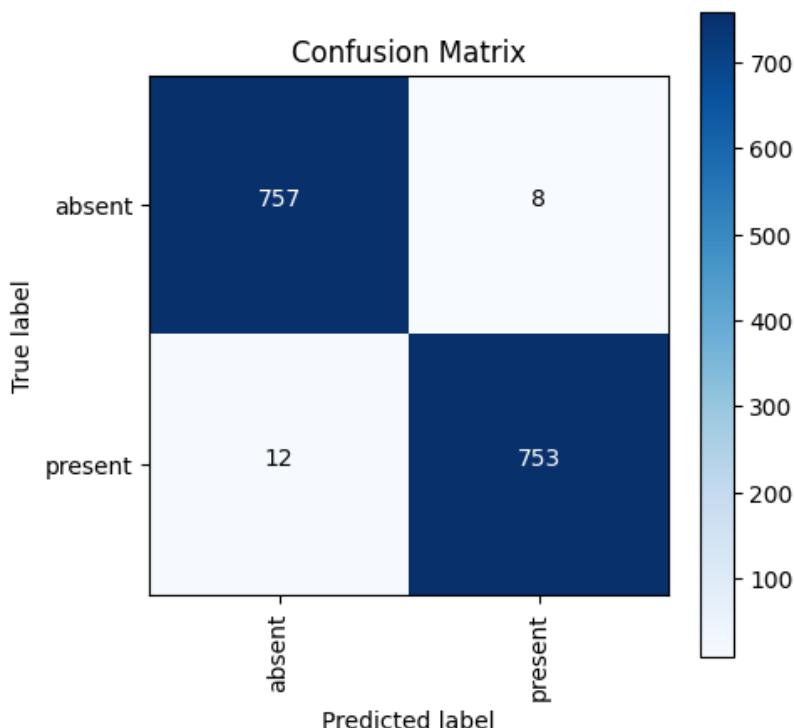


Figure 21 - Matrice de confusion du modèle sur le set de test

Cette matrice représente les prédictions réalisées par classe pour chaque image par rapport à la vérité. On constate donc ici une grande majorité de prédiction correcte.

Pour notre curiosité, nous avons également visualisé certaines images afin d'observer lesquels étaient correctement et lesquelles ne l'étaient pas. En plus de ceci, nous avons également réalisé une *heatmap* des images traitées afin d'identifier les zones d'intérêts de l'image pour le modèle. Les **Figures 22** et **23** présentent ces visualisations.

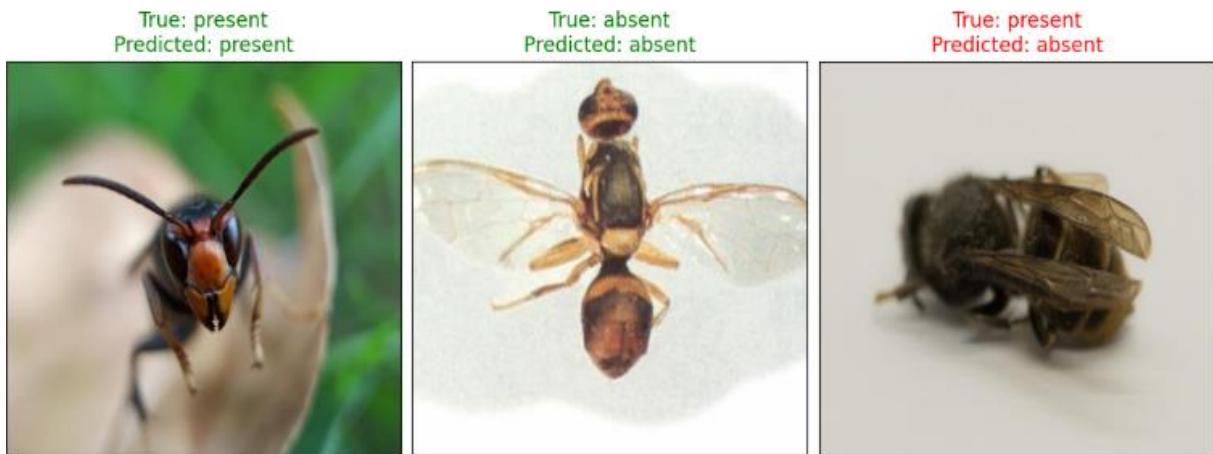


Figure 22 - Visualisation des données de tests avec leurs prédictions

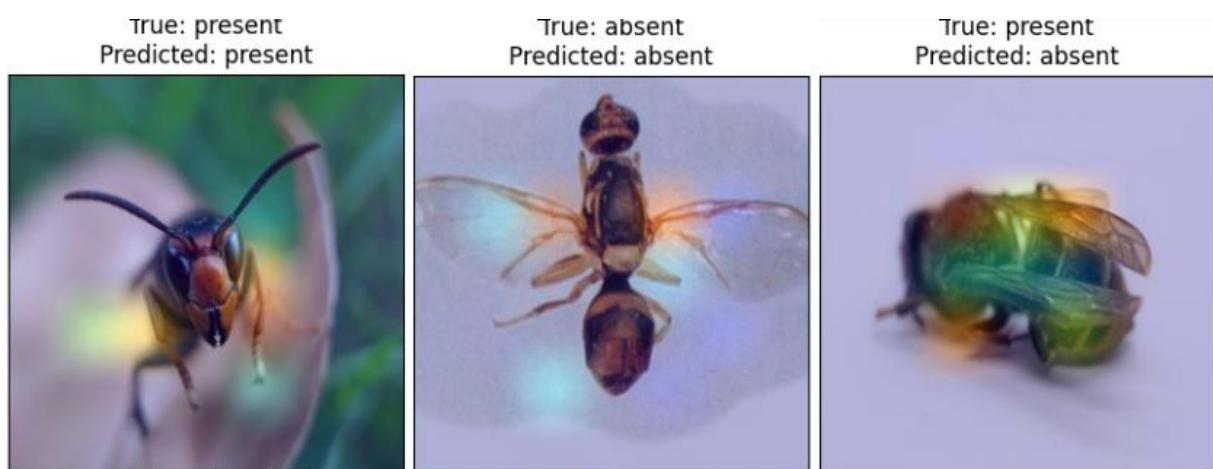


Figure 23 - Heatmap des données de tests avec leur prédiction

Plus le filtre sur l'image dispose d'une couleur chaude, plus cela indique une région d'intérêt pour le modèle

Pour l'anecdote, nous avions procédé à une première version du modèle sans filtrer les images altérées aux bords noir du jeu de données sur *Vespa Velutina*. La heatmap réalisée nous a été très utile, car c'est elle qui a mis en lumière le fait que le modèle semblait déterminer la présence du frelon en analysant les bords de l'image. Faisant ainsi l'amalgame : « Bords noirs = frelon asiatique ». Ce phénomène est visible sur la **Figure 24**.

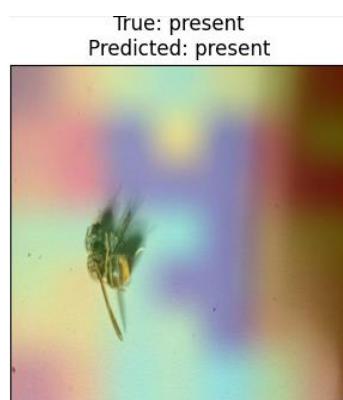


Figure 24 - Heatmap d'une version antérieure du modèle

Contrairement à la heatmap précédente, on constate ici un fort intérêt du modèle à consulter les bords de l'image.

D'apparence trop parfaite, le modèle réalisé ici nous a été suffisant pour des tests manuels. Toutefois, ce dernier présente quelques tares dont les origines sont détaillées dans le chapitre suivant. Puisque la réalisation de ce modèle ne fait pas partie de l'objectif principal de ce travail de bachelor, nous acceptons cette version comme fonctionnelle.

4.2 Mesure de performance du *Dart Package*

Flutter offre la possibilité de mesurer la performance en lançant l'application en *profile mode*. Ce mode permet notamment de mesure l'usage de la mémoire et la fréquence d'images d'une application afin de détecter des éventuels problèmes de performances. Toutefois, ce mode ne permet pas de mesurer la consommation en énergie directement.

Une autre mesure que nous allons réaliser concerne le temps d'inférence du modèle. Ceci est mesurable directement depuis le code via l'objet *Stopwatch* permettant de mesurer un temps écoulé depuis un début défini manuellement. Puisque nous réalisons trois opérations distinctes (pré-processing, inférence et labélisation), nous allons mettre en évidence ces trois temps dans le processus global, donc dès que la *package* reçoit l'image jusqu'au renvoi de la prédiction à l'application.

Ne disposant pas de réelle référence sur laquelle s'appuyer et comparer nos résultats, nous allons plutôt concentrer nos efforts sur les éventuelles améliorations qu'apportent les différentes options comment le nombre de threads ou l'utilisation du GPU.

4.2.1 Méthodologie de test

Par soucis de simplicité et de meilleure maîtrise de l'appareil, nous avons réalisé l'ensemble des tests sur un téléphone *Android*. Plus précisément sur un appareil *Google Pixel 6 Pro* disposant des spécificités suivantes :

- **Mémoire vive : 12 Go**
- **Nombre de cœur processeur : 8**

Les mesures du temps d'inférence ont été réalisé en sélectionnant 1'000 images du jeu de donnée de test de notre *dataset* utilisé pour la validation de notre modèle. Notez ici que les images utilisées importent peu. Nous ne nous intéressons pas ici à la précision du modèle, mais seulement à son temps d'exécution peu importe l'entrée.

Nous avons également choisi d'inférer les images sur le modèle pré-entraîné avec *ImageNet* plutôt que celui sur le frelon asiatique pour une raisons simple : nous disposons d'une version à virgule flottante et une version quantifiée de ce modèle. Ceci nous permettra donc d'observer les différences dans le cas où les poids d'un même modèle utilisent un type numéraire différent.

Nous utiliserons toujours le même point de référence pour mesurer les différences de performances selon les options sélectionnées. Il s'agira du modèle pré-entraîné sur *ImageNet* avec la configuration suivante :

- **Type numéraire : float32**
- **Processeur utilisé : CPU**
- **Nombre de threads CPU : 1**
- **Optimisation XNNPack : désactivée**

4.2.2 Espace mémoire

Le mode *profile* de *Dart* permet de monitorer en temps réel l'utilisation mémoire de l'appareil. Cette mesure est présentée sous la forme d'un graphique au cours du temps représentant la consommation de mémoire de l'app mais également la mémoire consommée par l'appareil. La **Figure 25** présente la consommation de la mémoire lors d'une inférence.

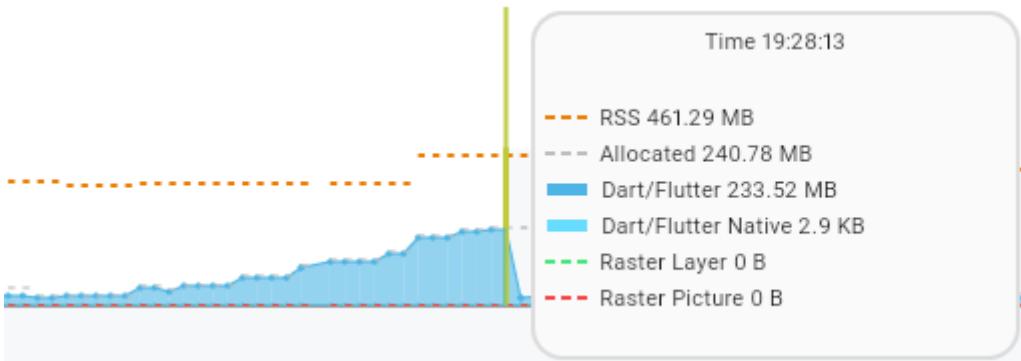


Figure 25 - Snapshot de consommation de mémoire de l'appareil lors d'une inférence

La légende représente les valeurs au niveau de la ligne verte. La consommation de l'app est représentée par la légende « Allocated ». La valeur RSS correspond à la consommation totale de la mémoire de l'appareil, hors app actuelle comprise.

4.2.3 Vitesse d'inférence

Nous désirions initialement découper la mesure d'inférence en trois, pour chaque étape du processus de notre *package*. Toutefois, après mesures, nous avons constaté que le temps nécessaire au *mapping* des prédictions avec les labels était inférieur à la milliseconde, le rendant complètement négligeable. Quant au temps nécessaire à la normalisation, il s'avère en moyenne inférieur à 10 ms. Nous voulions le différencier du temps d'inférence, mais il finissait par ne plus être visibles dans nos histogrammes. Nous l'avons donc incorporé au temps total. La **Figure 26** montre toutefois cette séparation des deux temps afin de visualiser la part de temps d'exécution entre les deux processus.

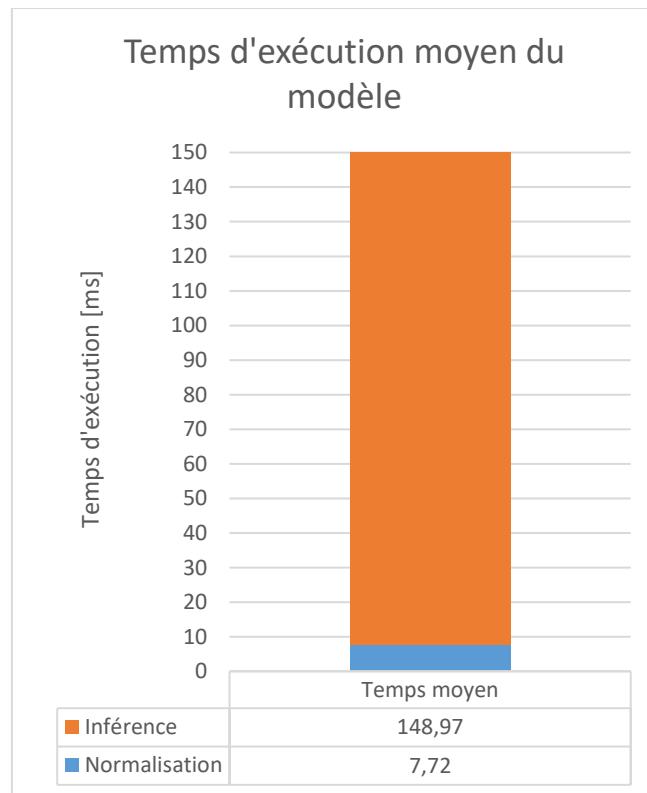


Figure 26 - Temps d'exécution moyen du modèle séparé par inférence et normalisation

Les figures suivantes ne feront donc plus la distinction entre ces deux temps. Nous présenterons directement la somme de ces deux temps.

Nous avons donc réalisé plusieurs mesures comparatives avec notre cas de référence, chaque fois en activant une option afin de voir son impact sur le temps d'inférence. Vous trouverez dans les figures suivantes les différents graphiques des mesures réalisées.

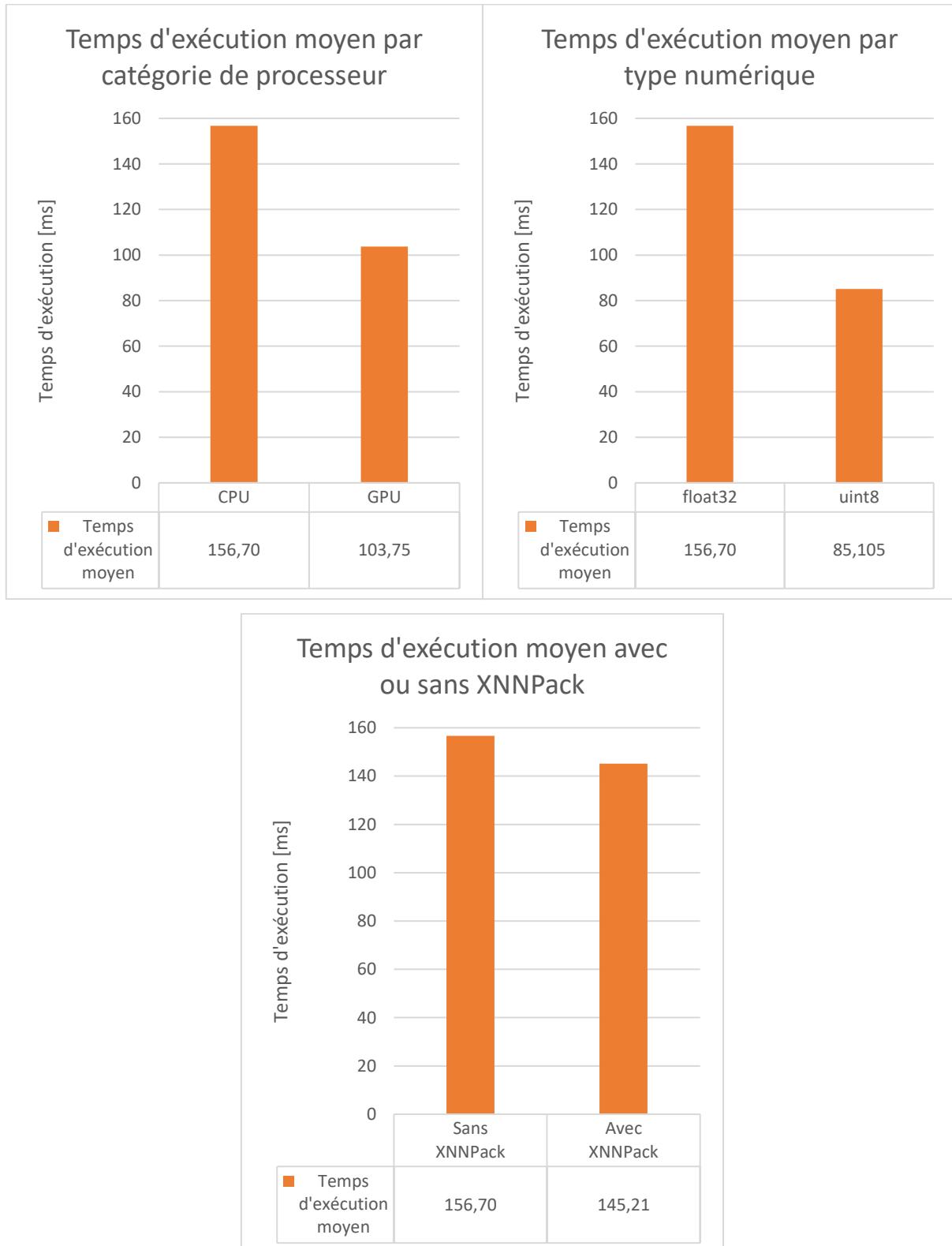


Figure 27 - Comparatifs des augmentations de performance en fonction des options sélectionnées par rapport au cas de référence

Quel que soit l'option choisie, on constate une amélioration plus ou moins significative sur le temps d'inférence

La **Figure 28** montre les différences de temps d'inférence selon le nombre de threads CPU sélectionné.

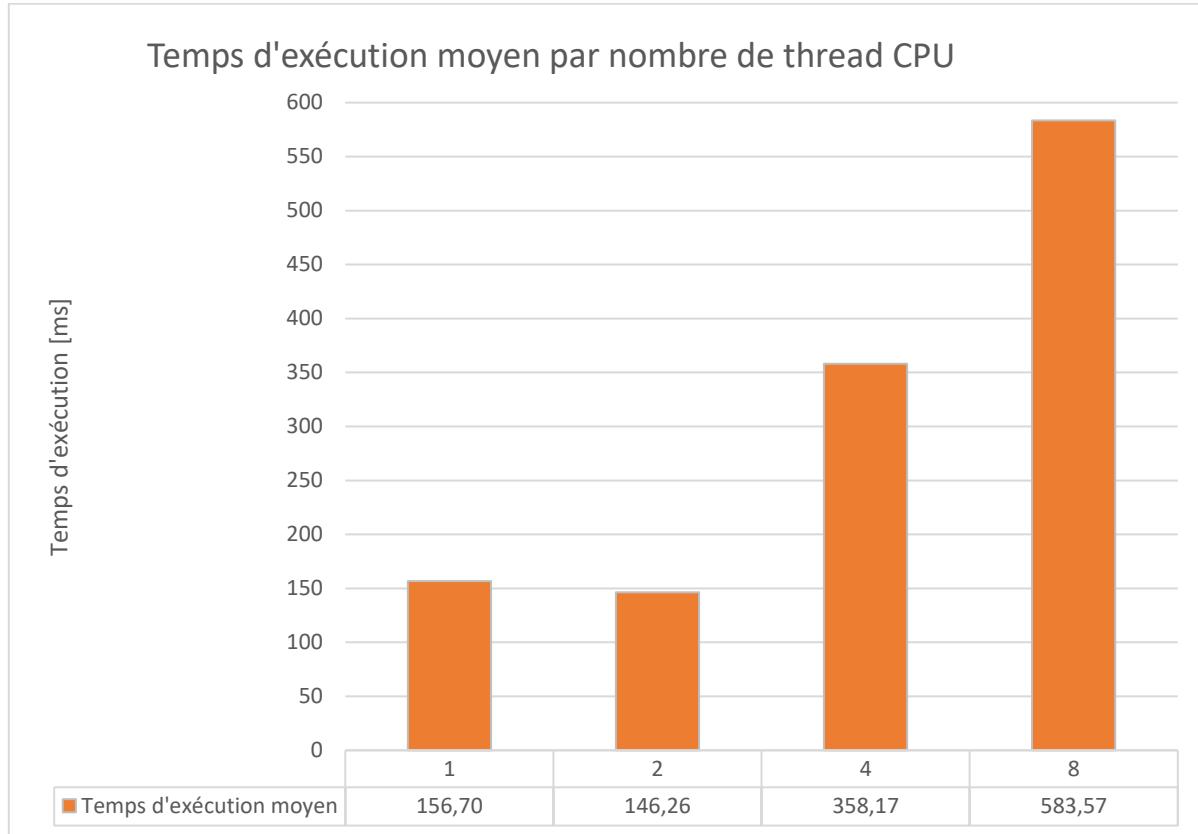


Figure 28 - Temps d'inférence moyen selon le nombre de threads utilisé

Nous pouvons étonnamment remarqué qu'à partir de 4 threads, le temps d'inférence augmente significativement. Ce phénomène est probablement dû au fait que le temps d'inférence est de manière générale relativement bas (moins de 200 ms). Avec l'augmentation du nombre de threads utilisés vient également des temps d'exécution additionnels comme les copies du contexte d'exécution et les synchronisations des différents threads afin de reconstituer les données. Dans le contexte d'une classification d'image, le nombre de thread n'est donc pas un paramètre pertinent à augmenter pour améliorer les performances. Il est préférable de ce tourner à d'autres technologies comme la quantification ou l'utilisation de processeurs graphiques.

4.2.4 Taille de l'application

Lors de la conception de notre prototype, nous avions eu une crainte en observant la taille de l'*apk* produit qui frôlait le 1 GB. Nous avons donc vérifié ici en produisant l'*apk* en mode *release*. Heureusement, notre première tentative devait être issu d'une mauvaise manipulation de notre part, car la taille du produit final n'est que de 102.7 MB.

Nous n'avons jamais réussi à reproduire l'ancienne taille exagérée obtenue précédemment.

4.2.5 Retours sur notre implémentation

Nous sommes très satisfaits des performances et de l'utilisation de notre *package*. Celui-ci répond à nos ambitions initiales, à savoir : offrir la possibilité d'utiliser un modèle de classification d'image simplement depuis n'importe quelle application *Flutter*. La conception de notre *package* rend son utilisation possible dans n'importe quelle autre app.

Bien qu'il serait possible de mettre notre solution à disposition de la communauté, son destin risque d'être semblable aux nombreuses autres tentatives du même genre, comme le *plugin tflite_flutter_helper*. D'une part, nous avons un couplage fort avec *tflite_flutter* qui dispose actuellement d'un suivi ralenti. Et pour terminer, celui-ci recommande l'utilisation de *MediaPipe* qui, nous imaginons, sera à termes plus adapté et pour des modèles *deep learning* plus étendu (texte, son, vidéo, etc...).

Chapitre 5

Axes d'amélioration

5.1 Modèle de classification du frelon asiatique

5.1.1 *Dataset du frelon asiatique*

Le modèle dont nous disposions était initialement prévu pour procéder à une détection d'objet. Ainsi, les images que contient le jeu de données sont parfois non pertinentes pour l'entraînement d'une classification car l'insecte apparaît de façon décentrée sur l'image, ou est parfois éloigné ou masqué derrière certains obstacles (par exemple capturé dans une bouteille). Un exemple est fourni dans la **Figure 29**.

Il est donc difficile d'être certain que notre modèle ait appris les caractéristiques physiques spécifiques du frelon. Ce dernier aurait très bien pu apprendre des informations sur l'environnement ce qui n'est pas forcément pertinent.

Nous disposons des coordonnées des boîtes encadrant l'individu dans chaque image. Un script aurait pu être réalisé afin de mieux détourer le sujet d'intérêt pour l'apprentissage. L'autre solution aurait été de proposer un système de détourage directement dans l'application mobile permettant de mieux centrer l'insecte dans l'image à analyser.



Figure 29 - Image du set d'entraînement

On distingue l'insecte présent sur la fleur, mais il n'occupe qu'une infime partie de l'image au complet

5.1.2 Dataset complémentaire

Puisque l'objectif principal du projet ne résidait pas dans l'obtention d'un modèle parfait, certains compromis ont été accepté notamment sur le jeu de donné supplémentaire utilisé pour l'entraînement. Nous avions opté pour un *dataset* contenant diverses images d'insectes nuisible, mais ce dernier dispose de certains élément « pollués » contenant du texte, voire carrément des dessins. Or comme présenté lors de notre analyse de l'état de l'art, nous avions mis en lumière l'exemple d'un *watermark* faussant la reconnaissance d'un cheval sur une image. Le même phénomène a donc pu se produire ici.

Nous avons accepté cette situation puisque nous ne disposions pas d'autres jeux de données suffisamment fourni. Ou alors, il aurait fallu encore réduire le nombre de donnée de notre jeu de *Vespa Velutina* pour un bon équilibrage, ce qui n'aurait pas été idéal.

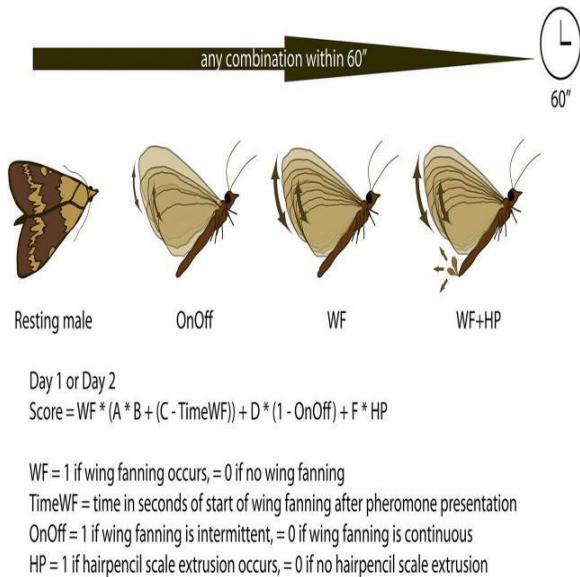


Figure 30 - Image issue du set d'entraînement

Cette image a donc été traitée et analysée par notre modèle pour s'entraîner, ce qui n'est absolument pas représentatif de notre cas d'utilisation ou, à priori, seul des photos réelles seront prises.

5.1.3 Précision relative du modèle

Malgré un score de précision excellent, nous avons très vite réalisé que ce dernier n'est pas vraiment représentatif et interprétable dans plusieurs contextes. Par exemple, notre modèle a tendance à retourner un résultat positif à la présence de *Vespa Velutina* dans des clichés ne contenant même pas d'insecte à l'image. Cette situation est probablement due éléments susmentionnés dans cette section.

En effet, si les images de *Vespa Velutina* sont régulièrement des images où le sujet n'est pas centré et est en tout petit dans un vaste décor, alors le modèle est probablement biaisé et part du principe qu'un décor vide a plus de chance de contenir un frelon asiatique.

5.2 Dart Package pour la classification d'images

5.2.1 Interprétation des headers du fichier tflite

Nous avons forcé les consommateurs de la lib à fournir un fichiers supplémentaire contenant les différents labels des classes de leurs modèles. Toutefois, *TensorFlow Lite* offre la possibilité de lier des fichiers directement au fichier *tflite*. Par la suite, nous aurions pu accéder à ce fichier et en extraire les informations souhaitées. La **Figure 31** schématisé l'encodage du format *tflite*.

Les fonctionnalités proposées ne s'arrêtent pas là, et il est possible de fournir un ensemble de métadonnées au fichier *tflite*, notamment des informations sur les entrées et sorties du modèles, les ensembles de valeurs supportées, les types numériques traités, le spectre d'encodage des couleurs de l'image, etc...

Nous n'avions pas eu besoin d'informations complémentaires autres que les formats des entrées et sorties du modèle. L'utilisation des labels en format séparé nous a permis d'expérimenté avec les librairies de *Flutter* sans avoir besoin de recompiler le modèle au format *tflite*. D'où notre choix de ne pas approfondir cette piste.

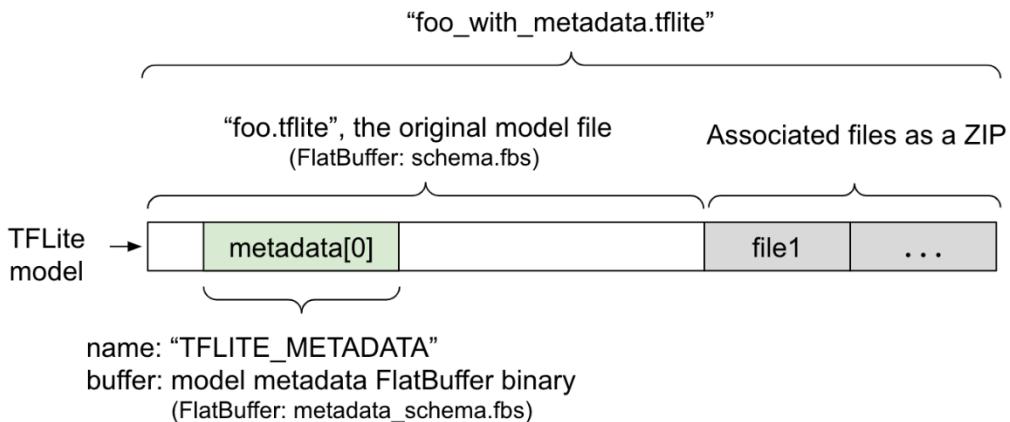


Figure 31 - Schéma des données d'un fichier .tflite^[11]

Les métadonnées en vert peuvent donner en information la présence de fichiers additionnels zippés (en gris foncé)

5.2.2 Traitement de différents canaux de couleurs

Nous avons généralisé l'implémentation de notre *package* à des images utilisant les canaux RGB²⁹. Ceci est limitant dans la mesure où la numérisation d'une image en noir et blanc sera effectuée selon ces trois canaux en utilisant notre *package*. Ceci alourdi inutilement le processus de normalisation qui devra effectuer trois calculs par pixel de l'image au lieu d'un seul.

La deuxième limitation plus contraignante, est que notre librairie aura un comportement indéterminé pour des images utilisant d'autres canaux moins conventionnels comme CIELAB qui utilise aussi 3 canaux, mais qui n'ont pas les mêmes significations que RGB.

5.2.3 Solution universelle

Nous l'avons mentionnée plusieurs fois dans ce rapport, mais par manque de temps, nous n'avons pas pu approfondir une solution universelle pouvant s'interfacer avec n'importe quel *framework* ou langage *cross-platform*. *iOS* et *Android* sont deux écosystèmes dont le code natif repose sur une base C++. Or ce langage dispose de ses propres librairies *TensorFlow Lite* que nous imaginons mieux maintenues étant donné la grande communauté et l'écosystème ultra développé qui l'accompagne.

Du reste, le package *tflite_flutter* est déjà une abstraction de son implémentation homologue en C++. En atteste le code source du *package* faisant appel à beaucoup de fonction native au travers du *Foreign Function Interface* de *Dart*.

Toutefois, l'existence même de cette solution repose sur nos hypothèses et n'est, peut-être, pas réalisable. Même si une base en C++ commune est réalisable, elle ne retirera en rien la nécessité de développer les interfaces propres à chacune des *frameworks* et langages *cross-platforms*.

²⁹ Rouge, Vert, Bleu

Chapitre 6

Conclusion

En conclusion, nous sommes satisfaits du travail réalisé, des résultats et de la performance, aussi bien concernant l'apprentissage du modèle que notre *package*. Dans un monde où l'intelligence artificielle prend de plus en plus d'ampleur, il est étonnant de constater que l'intégration de modèle *deep-learning* sur des langages *cross-platform* restent encore un sujet « de niche » avec assez peu d'outils complets, démocratisés et maintenus.

Force est de constater que *Flutter* souffre encore de son statut de jeune *framework* avec une communauté certes grandissante, mais bien inférieure en comparaisons avec d'autres. Cette situation a complexifié certains aspects de ce travail, notamment lorsque *Dart* a rendu obsolète certains types utilisés dans *tflite_flutter*, nous obligeant à modifier manuellement le code source de ce *package*, ou encore l'absence de *MediaPipe* pour la classification d'image.

Cela étant dit, ce même statut qu'a *Flutter* rend enrichissant la participation au développement de son écosystème. Même si à long terme, notre *package* risque de devenir obsolète en étant remplacé par d'autres solutions, nous avons su acquérir des compétences sur ce langage et sur le développement de bibliothèques tierces.

Ce travail justifie également l'approfondissement de ce sujet en tentant de développer une solution universelle permettant d'intégrer des modèles *deep-learning* sur mobile tout en faisant l'abstraction du langage utilisé.

Chollet Bastian

Bibliographie

1. Bhatt, D., Patel, C., Talsania, H., Patel, J., Vaghela, R., Pandya, S., . . . Ghayvat, H. (2021). *CNN Variants for Computer Vision: History, Architecture, Application, Challenges and Future Scope*. Retrieved from MDPI: <https://www.mdpi.com/2079-9292/10/20/2470>
2. Bhavesh, S. B. (2020, Novembre 3). *Types of Convolutional Neural Networks: LeNet, AlexNet, VGG-16 Net, ResNet and Inception Net*. Retrieved from Medium: <https://medium.com/analytics-vidhya/types-of-convolutional-neural-networks-lenet-alexnet-vgg-16-net-resnet-and-inception-net-759e5f197580>
3. Creus Castanyer, R., Martínez-Fernández, S., & Franch, X. (2021, Mars 11). *Integration of Convolutional Neural Networks in Mobile Applications*. Cornell University.
4. Cyp. (2023, Novembre). *Asian Hornet 2 Dataset*. Retrieved from Roboflow Universe: <https://universe.roboflow.com/cyp-puhyr/asian-hornet-2>
5. Dart. (2024, Juillet 29). *Concurrency in Dart*. Retrieved from Dart Dev.: <https://dart.dev/language/concurrency>
6. Dart. (2024, Août 24). *Isolates*. Retrieved from Dart Dev: <https://dart.dev/language/isolates>
7. Detection, U. C. (2022, Février). *Asian Hornet Detection Computer Vision Project*. Retrieved from Roboflow Universe: <https://universe.roboflow.com/use-case-asian-hornet-detection/asian-hornet-detection-a6ael>
8. Flutter. (2024, Juillet 23). *Developing packages & plugins*. Retrieved from Docs | Flutter: <https://docs.flutter.dev/packages-and-plugins/developing-packages>
9. Gillis, A. S. (2023, Septembre). *transfer learning*. Retrieved from TechTarget: <https://www.techtarget.com/searchcio/definition/transfer-learning>
10. Google. (2023). *flutter-tflite*. Retrieved from GitHub: <https://github.com/tensorflow/flutter-tflite>
11. Google AI. (2024, Septembre 5). *Ajouter des métadonnées aux modèles LiteRT*. Retrieved from Google AI for Developers: https://ai.google.dev/edge/liter/models/metadata?hl=fr#pack_the_associated_files
12. Harsh, B. (2020, Juin 5). *Run CNN model in Flutter*. Retrieved from Medium: <https://medium.com/analytics-vidhya/run-cnn-model-in-flutter-10c944cadcb>
13. Hollemans, M. (2020, Avril 8). *New mobile neural network architectures*. Retrieved from Machine, Think!: <https://machinethink.net/blog/mobile-architectures/>
14. *Image Classification vs. Object Detection: Everything You Need to Know*. (2024, Février 19). Retrieved from Mindy Support: <https://mindy-support.com/news-post/image-classification-vs-object-detection-everything-you-need-to-know/>
15. Kieran, M. (2022, Juin 18). *Deep Learning Project: Image Classifier deployed to Mobile from Scratch*. Retrieved from Medium: <https://kieran-mcc91.medium.com/deep-learning-project-image-classifier-deployed-to-flutter-app-from-scratch-f77c53014cb9>

16. Lapuschkin, S., Wäldchen, S., Binder, A., Montavon, G., Samek, W., & Müller, K.-R. (2019, Mars 11). Unmasking Clever Hans predictors and assessing what machines really learn. *Nature Communications*, p. 10.
17. Mrinal, W. (2022, Septembre 28). *Object Detection vs. Image Classification vs. Keypoint Detection*. Retrieved from Roboflow Blog: <https://blog.roboflow.com/object-detection-vs-image-classification-vs-keypoint-detection/>
18. Nishan, G., Jeewani Anupama, G., Bahman, J., & Gough, L. (2022). Performance Analysis of CNN Models for Mobile Device Eye Tracking with Edge Computing. *Procedia Computer Science*, pp. 2291-2300.
19. Rasyad, M. A., Dewanta, F., & Astuti, S. (2021). *All-in-one computation vs. computational-offloading approaches: a performance evaluation of object detection strategies on android mobile devices*. Retrieved from ResearchGate: https://www.researchgate.net/publication/358725019_All-in-one_computation_vs_computational-offloading_approaches_a_performance_evaluation_of_object_detection_strategies_on_android_mobile_devices
20. ruhela, I. (2023, Octobre 12). *IMAGE CLASSIFICATION VS OBJECT DETECTION AND IMAGE SEGMENTATION*. Retrieved from Medium: <https://ruhelalakshya.medium.com/image-classification-vs-object-detection-and-image-segmentation-e89af675e976>
21. TensorFlow. (2020, Juillet 24). *Accelerating TensorFlow Lite with XNNPACK Integration*. Retrieved from TensorFlow Blog: <https://blog.tensorflow.org/2020/07/accelerating-tensorflow-lite-xnnpack-integration.html>
22. vencerlanz09. (2022). *Insect Classification using CNN + MobileNetV3*. Retrieved from kaggle: <https://www.kaggle.com/code/vencerlanz09/insect-classification-using-cnn-mobilenetv3/notebook>
23. Wikipedia. (s.d.). *Wikipedia*. Consulté le Juillet 2, 2024, sur https://fr.wikipedia.org/wiki/Apprentissage_profond

Annexes

Annexe 1 - Journal de travail 58

Annexe 1 - Journal de travail

Analyse et état de l'art	86:29:00
Réalisation du modèle	12:00:00
Réalisation de l'application	53:19:00
Tests et validations	13:12:00
Gestion du projet, documentation et présentation	80:03:00
Total	245:03:00

<i>Date</i>	<i>Début</i>	<i>Fin</i>	<i>Durée</i>	<i>Quoi</i>
20.06.2024	10:00:00	14:00:00	04:00:00	Gestion du projet, documentation et présentation
23.06.2024	10:00:00	14:00:00	04:00:00	Gestion du projet, documentation et présentation
25.06.2024	18:00:00	19:00:00	01:00:00	Gestion du projet, documentation et présentation
01.07.2024	09:30:00	10:04:00	00:34:00	Analyse et état de l'art
01.07.2024	10:05:00	17:00:00	06:55:00	Analyse et état de l'art
02.07.2024	09:00:00	15:16:00	06:16:00	Analyse et état de l'art
02.07.2024	18:20:00	19:20:00	01:00:00	Analyse et état de l'art
03.07.2024	10:00:00	12:30:00	02:30:00	Analyse et état de l'art
05.07.2024	13:50:00	17:50:00	04:00:00	Analyse et état de l'art
06.07.2024	14:00:00	17:30:00	03:30:00	Analyse et état de l'art
09.07.2024	10:40:00	18:17:00	07:37:00	Analyse et état de l'art
09.07.2024	21:40:00	23:20:00	01:40:00	Analyse et état de l'art
10.07.2024	09:20:00	11:50:00	02:30:00	Analyse et état de l'art
10.07.2024	13:00:00	20:00:00	07:00:00	Analyse et état de l'art
11.07.2024	09:00:00	13:20:00	04:20:00	Analyse et état de l'art
12.07.2024	10:00:00	19:25:00	09:25:00	Analyse et état de l'art
13.07.2024	12:00:00	17:00:00	05:00:00	Analyse et état de l'art
14.07.2024	12:00:00	20:21:00	08:21:00	Analyse et état de l'art
15.07.2024	16:00:00	22:52:00	06:52:00	Analyse et état de l'art
17.07.2024	19:00:00	23:59:00	04:59:00	Analyse et état de l'art
18.07.2024	10:00:00	12:00:00	02:00:00	Analyse et état de l'art
19.07.2024	14:00:00	16:00:00	02:00:00	Analyse et état de l'art
27.07.2024	14:00:00	18:00:00	04:00:00	Réalisation du modèle
08.08.2024	13:47:00	17:25:00	03:38:00	Réalisation du modèle
09.08.2024	13:38:00	18:00:00	04:22:00	Réalisation du modèle
10.08.2024	10:00:00	16:50:00	06:50:00	Gestion du projet, documentation et présentation
16.08.2024	13:30:00	18:00:00	04:30:00	Réalisation de l'application
17.08.2024	09:00:00	15:40:00	06:40:00	Réalisation de l'application
19.08.2024	10:30:00	16:00:00	05:30:00	Réalisation de l'application

23.08.2024	08:30:00	14:11:00	05:41:00	Réalisation de l'application
24.08.2024	08:30:00	14:28:00	05:58:00	Réalisation de l'application
29.08.2024	13:00:00	18:00:00	05:00:00	Réalisation de l'application
30.08.2024	13:00:00	18:00:00	05:00:00	Réalisation de l'application
31.08.2024	13:00:00	18:00:00	05:00:00	Réalisation de l'application
02.09.2024	13:00:00	18:00:00	05:00:00	Réalisation de l'application
03.09.2024	13:00:00	18:00:00	05:00:00	Réalisation de l'application
04.09.2024	13:00:00	17:34:00	04:34:00	Gestion du projet, documentation et présentation
05.09.2024	11:16:00	16:18:00	05:02:00	Gestion du projet, documentation et présentation
06.09.2024	09:44:00	18:03:00	08:19:00	Gestion du projet, documentation et présentation
07.09.2024	09:24:00	18:14:00	08:50:00	Gestion du projet, documentation et présentation
12.08.2024	10:12:00	18:07:00	07:55:00	Gestion du projet, documentation et présentation
13.09.2024	11:36:00	17:23:00	05:47:00	Gestion du projet, documentation et présentation
16.09.2024	10:29:00	16:06:00	05:37:00	Gestion du projet, documentation et présentation
17.09.2024	18:04:00	21:35:00	03:31:00	Tests et validations
19.09.2024	10:42:00	20:23:00	09:41:00	Tests et validations
20.09.2024	10:37:00	17:26:00	06:49:00	Gestion du projet, documentation et présentation
21.09.2024	09:45:00	15:00:00	05:15:00	Gestion du projet, documentation et présentation
22.09.2024	11:13:00	17:18:00	06:05:00	Gestion du projet, documentation et présentation