Rapport intermédiaire

# Préambule

Ce rapport intermédiaire 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 au contexte énoncé, 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-entrainée adaptée aux mobiles.

Ensuite, nous nous concentrerons sur la recherche de solutions existantes ou non permettant la réalisation de l’intégration d’un modèle deep learning entrainé 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.

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

# 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[[1]](#footnote-1). Dit autrement, le terme définit une technique utilisée par un ensemble de modèles comme l’extraction de caractéristique dans une image ou dans un son. 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 (CNN[[2]](#footnote-2)) 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[[3]](#footnote-3). Nous allons développer ces trois dernières tâches car elles représentent des solutions pertinentes à notre situation.

Une image contenant texte, capture d’écran, diagramme

Description générée automatiquement

Figure - Exemples de type de problèmes de vision par ordinateur[[4]](#footnote-4)

## 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, présentée dans la Figure 1, 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).

### 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 Figure 1, cette technique applique une seule classe par entrée.

#### 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.

#### 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.

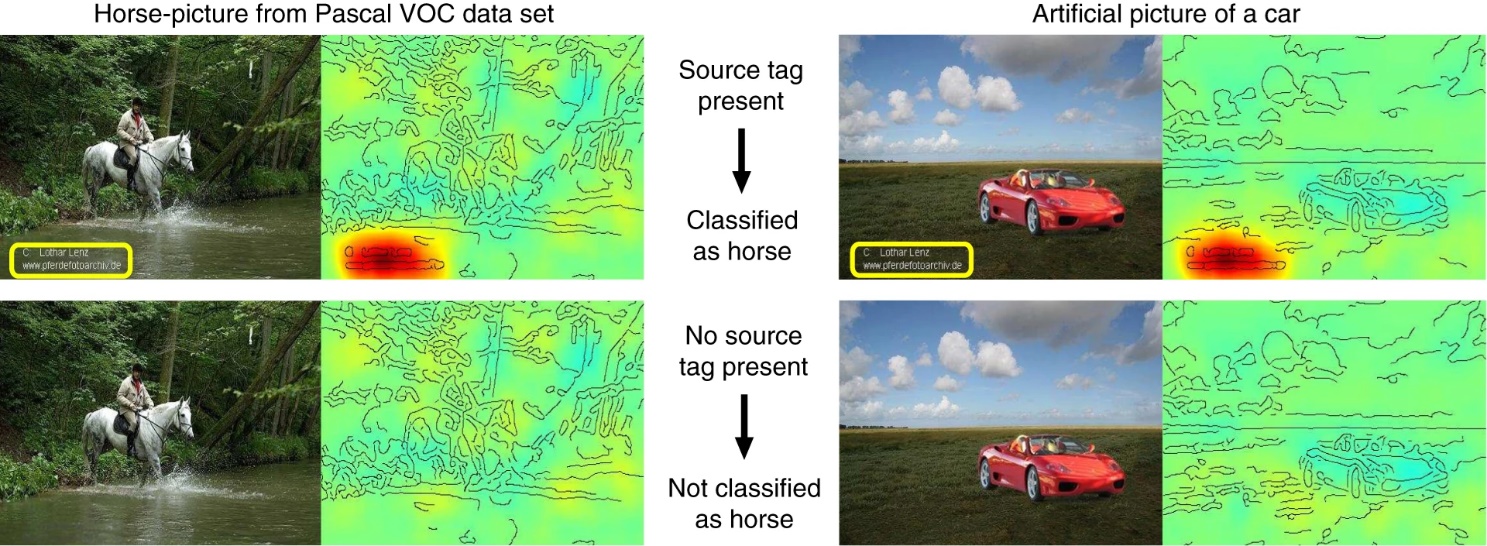


Figure - La présence d'un watermark activant fortement le modèle pour déterminer la classe "cheval" [[5]](#footnote-5)

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 prise 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.

### 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 Figure 1, 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’autre architectures ont vu le jour permettant d’accélérer ce processus et permettant la sélection de régions et la classification en une seule étape.

#### 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 identifier, 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’avèrerait fortement pratique 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.

#### Limitations

Les architectures pré-entrainées 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[[6]](#footnote-6).

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 faible 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.

## 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 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.

## 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, l’application enverrait dans tous les cas le cliché à un serveur ce qui permettra à un œil humain d’identifier plusieurs individus sur l’image si besoin.

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 au final 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.

# Dataset

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 entrainer 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 *Robotflow* [[7]](#footnote-7).

À la date de la mise en ligne du sujet de ce travail, un premier jeu de donnée a été suggéré[[8]](#footnote-8). 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 |

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 annotées[[9]](#footnote-9) 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 recherchée. L’autre solution consisterait à procéder à de l'apprentissage par transfert en sélectionnant un modèle pré-entraîné sur un ensemble de classe. On pourrait, par exemple, sélectionner un modèle entrainé sur un dataset particulier et le renforcer en y ajoutant notre nouvelle classe.

Une image contenant texte, capture d’écran, diagramme, Police

Description générée automatiquement

Figure - Ajout d'une classe "chat" sur un modèle entrainé à détecter la classe "chien" par transfert[[10]](#footnote-10)

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.

Malheureusement, il sera soit nécessaire d’agrémenter ce jeu avec d’autres données et de disposer d’une annotation pour cette nouvelle classe à introduire, soit de rechercher un modèle pré-entrainé sur lequel nous pourrons aisément appliquer de l’apprentissage par transfert.

# 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 faible 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 cités 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[[11]](#footnote-11) visant à mesurer les performances de différentes architectures CNN sur mobiles dans un contexte de détection de mouvement d’yeux (*eye tracking*).

## 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[[12]](#footnote-12). Nous vous proposons ci-dessous une rapide présentation de ces dernières.

### 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.

### 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.

### 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.

### 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 des features des différents canaux. Contrairement à MobileNet, le nombre de canaux en entrée et sortie reste identique.

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

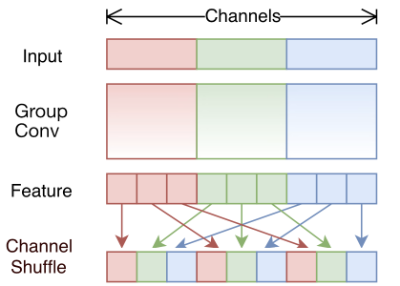


Figure - Dans sa première version, l'architecture effectuait également un regroupement des couche 1x1. Ceci a été abandonné dans la V2[[13]](#footnote-13)

## 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.

Une image contenant capture d’écran, texte, Tracé, ligne

Description générée automatiquement

Figure - Espace mémoire et consommation d'énergie lors d'analyse de 1'000 images

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 nombre de paramètres des différents modèles est le suivant

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

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.

Une image contenant texte, capture d’écran, diagramme, Tracé

Description générée automatiquement

Figure - Écarts quadratiques moyen par rapport au temps d'inférence

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.

Finalement, l’architecture MobileNet-V3 a beau disposer d’un temps d’inférence plus élevé que ShuffleNet-V2, sa précision est plus élevée.

## 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.

# 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.

## Création 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. Par conséquent, nous faisons le choix de développer le modèle à l’aide de *TensorFlow* et de *Keras*.

## 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.

### 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 entrainés 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.

### Open Neural Network Exchange (ONNX)

Il s’agit d’un écosystème d’intelligence artificielle open source ayant pour but de standardisé les opérations et les fichiers d’export afin d’assurer une interopérabilité entre les différents framework 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.

De part le soutient 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 exporter 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, seul des librairies non-officiel sont disponibles.

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 fournit par Flutter afin de générer du code Android et iOS dans leur version native en C++.

### 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 permettrons 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 disponible 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 envisageable.

## 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 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 plateforme 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 dispose tout 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[[14]](#footnote-14) » permettant l’appel de code C++ depuis le langage javscript. De même flutter permet le même procédé via le « Foreign Function Interface » mit 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 architecture[[15]](#footnote-15) », 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[[16]](#footnote-16) et Fabric Native Components[[17]](#footnote-17). 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 Andoid et iOS uniquement, nous préfèrerons l’utilisation du framework Flutter qui semble disposer d’une version plus stable dans la fenêtre de temps à parti pour la réalisation de ce projet. Nous n’excluons pas complètement React Native, mais nous préférons garder son utilisation si nous venions à réaliser une solution « universelle » adaptable à plusieurs langages cross platform.

## Inférence du modèle sur mobile

Récapitulons l’ensemble des technologies choisie 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 *TensforFlow* eux-mêmes. Il s’agit d’un plugin nommé *tflit\_flutter*[[18]](#footnote-18) 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 plugin 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[[19]](#footnote-19) 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.

# 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 pouvoir réaliser 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.

## 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 entrainement.

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 l’image fournie pour qu’elle puisse être process 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 donnait 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 librairies sont très faciles d’accès et il est possible d’obtenir un modèle avec lequel expérimenté rapidement en quelques lignes de codes.

## Export du modèle

Puisque nous importions 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.

## 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’incrémentation 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 des 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 çà 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.

## 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 somme inspirés du code fourni en exemple sur le répertoire Github[[20]](#footnote-20) 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[[21]](#footnote-21) 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][[22]](#footnote-22), une deuxième permet de récupérer le tenseur de sortie, à savoir [1, 1000][[23]](#footnote-23). 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’étapes 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 au dimension 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.

# Point de situation

À ce stade du rapport, nous pouvons réaliser un point de situation sur les objectifs et souhaits à réaliser pour ce projet. Premièrement, concernant le modèle à réaliser, nous sommes confiants sur la réalisation de la tâche d’un point de vue théorique. C’est-à-dire que le jeu de donnée est suffisamment fourni pour entrainer correctement le modèle, et que les librairies python permettent d’entrainer un modèle de façon simple mais efficace.

Toutefois, nous émettons des doutes sur le temps nécessaire à l’entrainement qui pourrait prendre plusieurs heures au minimum, et, plus probablement, quelques jours. Nous ne disposons pas de matériel informatique permettant de rendre trivial l’entraînement, et la moindre erreur ou ajustement dans les hyperparamètre coutera cher en temps pour mener à bien ce travail. Il nous sera nécessaire d’expérimenter en réduisant le nombre de données si nous souhaitons tout de même disposer d’un modèle de classification d’image pour l’insecte étudié. Dans le pire des cas, il nous sera peut-être nécessaire d’abandonner cette classification précise et choisir un autre modèle pré-entrainé pour classifier d’autres objets ou animaux.

Le prototype réalisé nous a permis de nous rendre compte de la faisabilité de l’objectif principal de ce travail. Ainsi, l’import et l’utilisation d’un modèle deep learning sur un téléphone portable via un framework cross platform ne relève en rien d’une tâche insurmontable et s’avère même plus simple qu’initialement suspecté. Les dernières inconnues étant l’exécution sur un appareil iOS ainsi que l’utilisation de la caméra plutôt que de prendre une photo directement depuis la galerie du téléphone. À priori, ni l’une ni l’autre de ces deux inconnues ne sont bloquantes. En effet, la librairie *tflite\_flutter* confirme son fonctionnement sur iOS, et la capture d’image in-app est une fonctionnalité basique offerte par Flutter et ses plugins.

À des fins de recherche, il serait donc d’avantages intéressant d’explorer l’implémentation de la solution dite « universelle », c’est-à-dire d’une solution capable d’être utilisée par une majorité de framework cross platform. Une telle solution devra très certainement se réaliser en C++ et nécessitera d’approfondir des outils avancés des framework cross platform les plus connus pour s’assurer du bon fonctionnement de notre implémentation. Bien que cette voie relève d’un défi technique, elle serait un challenge intéressant à relever.

1. Apprentissage profond (2024, 2 juillet). In *Wikipedia.* https://fr.wikipedia.org/wiki/Apprentissage\_profond [↑](#footnote-ref-1)
2. Convolutional neural networks [↑](#footnote-ref-2)
3. Bhatt, D., Patel, C., Talsania, H., Patel, J., Vaghela, R., Pandya, S., Modi, K., Ghayvat, H., (2021). *CNN Variants for Computer Vision: History, Architecture, Application, Challenges and Future Scope.* MDPI. https://www.mdpi.com/2079-9292/10/20/2470 [↑](#footnote-ref-3)
4. Mrinal Walia. (Sep 28, 2022). *Object Detection vs. Image Classification vs. Keypoint Detection*. Roboflow Blog: https://blog.roboflow.com/object-detection-vs-image-classification-vs-keypoint-detection/ [↑](#footnote-ref-4)
5. Lapuschkin, S., Wäldchen, S., Binder, A., Montavon, G., Samek, W., Müller, K-R., (2019, 11 mars), Unmasking Clever Hans predictors and assessing what machines really learn, *Nature Communications,* *10*. https://www.nature.com/articles/s41467-019-08987-4 [↑](#footnote-ref-5)
6. 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. 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* [↑](#footnote-ref-6)
7. https://universe.roboflow.com/ [↑](#footnote-ref-7)
8. Accessible ici : <https://universe.roboflow.com/use-case-asian-hornet-detection/asian-hornet-detection-a6ael/dataset/2> [↑](#footnote-ref-8)
9. Accessible ici : https://universe.roboflow.com/cyp-puhyr/asian-hornet-2/dataset/1 [↑](#footnote-ref-9)
10. S. Gillis, A., Cole, B., (2023, septembre), *Transfer learning,* https://www.techtarget.com/searchcio/definition/transfer-learning [↑](#footnote-ref-10)
11. Bhatt, D., Patel, C., Talsania, H., Patel, J., Vaghela, R., Pandya, S., Modi, K., Ghayvat, H., (2021). *CNN Variants for Computer Vision: History, Architecture, Application, Challenges and Future Scope.* MDPI. https://www.mdpi.com/2079-9292/10/20/2470 [↑](#footnote-ref-11)
12. Dataset de 1000 classes d’objets différents, animaux, véhicules, outils, mobilier, etc… [↑](#footnote-ref-12)
13. https://machinethink.net/blog/mobile-architectures/ [↑](#footnote-ref-13)
14. https://reactnative.dev/docs/native-modules-intro [↑](#footnote-ref-14)
15. https://reactnative.dev/docs/the-new-architecture/landing-page [↑](#footnote-ref-15)
16. https://github.com/reactwg/react-native-new-architecture/blob/main/docs/turbo-modules.md [↑](#footnote-ref-16)
17. https://github.com/reactwg/react-native-new-architecture/blob/main/docs/fabric-native-components.md [↑](#footnote-ref-17)
18. https://pub.dev/packages/tflite\_flutter [↑](#footnote-ref-18)
19. https://github.com/flutter-ml/google\_ml\_kit\_flutter [↑](#footnote-ref-19)
20. https://github.com/tensorflow/flutter-tflite/tree/main/example/image\_classification\_mobilenet [↑](#footnote-ref-20)
21. Sorte de mini-thread équivalent au coroutine dans Android [↑](#footnote-ref-21)
22. Respectivement : [Taille du batch, Largeur, Hauteur, canaux de couleurs (RGB)] [↑](#footnote-ref-22)
23. Respectivement : [Taille du batch, Nombre de sorties] [↑](#footnote-ref-23)