

4. Modèle de reconnaissance gestuelle

Dans cette partie nous présentons la manière dont nous avons utilisé un jeu de données basé sur le langage des signes américains (ASL) pour entraîner un modèle de réseau neuronal dans le but de détecter la gestuelle de nos mains. Pour ce faire, nous avons modélisé un réseau neuronal de convolution (CNN), nous l'avons entraîné et implémenté sur un ESP32. Après avoir testé notre modèle en conditions réelles, nous avons apportés des optimisations qui nous ont permis de conclure sur des résultats finaux satisfaisants au vu de notre hardware et du jeu de données à partir duquel le modèle a été entraîné.

4.1. Objectif

L'objectif est d'asservir notre drone à l'aide de la gestuelle de notre main. Pour ce faire, il nous faut en premier lieu disposer de capteurs permettant l'acquisition de données à partir desquelles nous pourrons travailler sur la reconnaissance gestuelle. De plus, l'objectif de ce projet est de démontrer la faisabilité de la solution proposée de A à Z à partir de composants électroniques, qui seuls ne représentent qu'un hardware assez modeste, mais qui ensemble et avec l'implémentation de notre firmware, sont théoriquement capable de grandes choses. Il était donc inenvisageable pour nous de déporter notre modèle de reconnaissance sur une machine extérieure qui communiquerait avec le drone puisque cela ne s'inscrirait pas dans notre démarche.



Figure 26 - Objectif du modèle de reconnaissance gestuelle

Ainsi, nous avions à la base dégager deux solutions. La première étant l'utilisation des données du capteur ToF pour entraîner et inférer avec notre modèle. A première vue, cette solution a l'avantage de dégager plus distinctement les formes de la main notamment en filtrant en fonction de la distance, elle possède également l'avantage de ne pas dépendre des conditions d'éclairage. Cependant malgré qu'il soit possible d'augmenter artificiellement la qualité de nos données (matrice 8x8) à partir d'un algorithme d'interpolation, nous n'avons pas réussi à trouver de data set labélisé à partir de gestes de la main pour un ToF. Ce pourquoi nous nous sommes dirigés vers une solution plus classique qu'est la caméra d'un AI Thinker ESP32-CAM.

4.2. Data Set

Une fois que nous nous étions mis d'accord sur le type de capteur utilisé, il nous fallait obtenir un modèle de réseau neuronal fonctionnel, ce dernier reposant principalement sur le data set à partir duquel il sera entraîné. Nous avons trouvé plusieurs solutions à des problématiques de reconnaissance gestuelle, dont nombreuses d'entre-elles utilisent MediaPipe développé par Google. C'est une bibliothèque open-source largement utilisée dans le milieu mais qui n'est utilisée exclusivement que pour des IA déportées sur une machine portable (PC, Mac...). Cela s'explique notamment par le fait que les modèles de machine Learning développés avec sont particulièrement sophistiqués puisqu'ils sont capables de déterminer en temps réel la position des doigts et des phalanges.

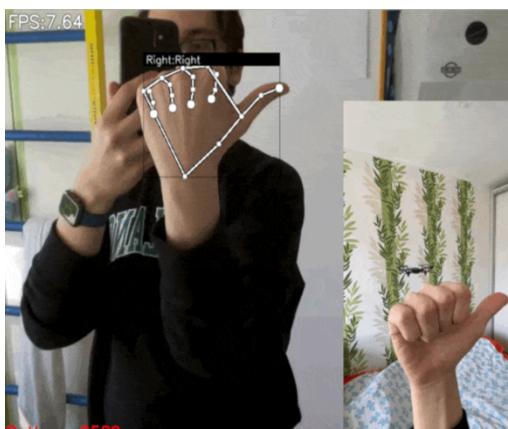


Figure 27 – Drone contrôlé par reconnaissance gestuelle à partir d'un modèle de Machine Learning MediaPipe déporté sur une machine secondaire (Mac)

Comme le montre la figure ci-dessous, créer notre propre jeu de données présente des avantages comme la labellisation des données propre à notre application et la réduction de biais. Cependant, nous n'avons pas opté pour cette solution, principalement en raison de contraintes de ressources. En effet, un jeu de données efficace nécessite un volume important de données, ce qui était hors de notre portée.

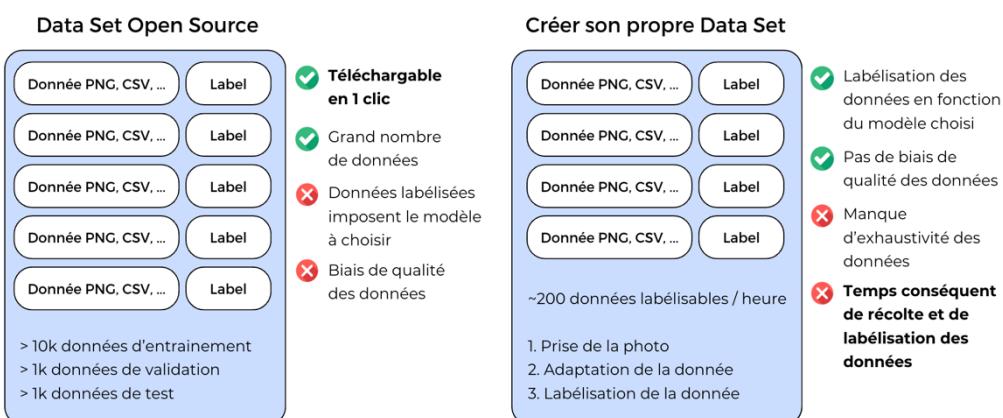


Figure 28 - Comparatif entre la création de son propre Data Set et l'utilisation d'un Data Set open-source

Aussi, un modèle de type MediaPipe n'aurait pas pu être envisageable au vu de la puissance de l'ESP32. Il nous fallait un data set bien plus simple, constitué de différents gestes précis, déjà labélisés. C'est ainsi que nous avons opté pour un data set tiré de la base de données MNIST (Modified National Institute of Standards and Technology) qui représente les différents gestes de main possible dans le langage des signes américains (ASL), comme l'illustre la figure ci-dessous

Data Set Open Source

Donnée PNG, CSV, ...	Label

> 10k données d'entraînement
> 1k données de validation
> 1k données de test

ASL Data Set
ASL = American Sign Language



Figure 29 - Visuels des différents gestes labélisés sur le Data Set ASL

Les lettres J et Z ne sont néanmoins pas présentes car elles ne relèvent pas d'un geste statique mais d'un mouvement particulier, toutefois nous possédons un nombre de gestes identifiables largement suffisant au vu de l'application souhaitée. L'enjeu derrière un modèle d'IA est de le rendre le plus efficace possible, l'objectif est ainsi de réduire au maximum la taille des données d'entrée à traiter durant le processus d'inférence. C'est la raison pour laquelle, les données des data set, à l'image des données que l'on mettra en entrée de notre modèle durant l'inférence, sont constamment redimensionné à une résolution très faible mais suffisante pour identifier les gestes de la main. Aussi, comme nous pouvons le voir sur la figure ci-dessous, la couleur n'apporte généralement que très peu d'information si ce n'est que de tripler le temps d'inférence car l'encodage est en RGB, les images d'entraînement sont donc fournies en gris.



- 27 455 données d'entraînement
- 7 172 données de test et de validation
- 28 x 28 pixels
- Grayscale 0 - 255
- 24 labels (J et Z exclus)

Figure 30 - Exemples de données fournis par le Data Set ASL

4.3. Modèle CNN

Le plus dur qui relève du fait de trouver le data set le plus adapté à notre application est désormais fait. Ainsi, à partir des données déjà labélisés dans le data set nous savons quelle sera la forme de sortie de notre réseau. Et à partir du format des données labélisées, nous connaissons le format des données à mettre en entrée pour l'inférence et donc la forme d'entrée de notre réseau. Nous en déduisons ainsi la forme externe du réseau neuronal comme nous l'illustrons sur la figure ci-dessous. Logiquement, nos données d'entrée et de sortie seront alors respectivement interprétées sous forme de int et de float, où un neurone en entrée représente 1 pixel {0, ..., 255}, et un neurone en sortie représente la probabilité que l'image inférée corresponde au label [0, 1].

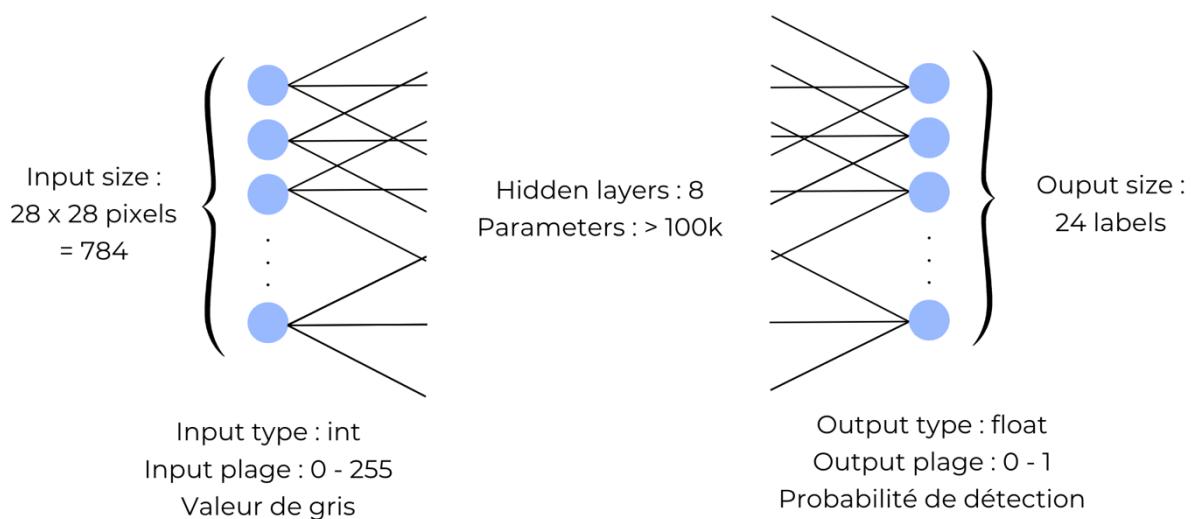


Figure 31 - Forme externe et format d'entrée-sortie du réseau de neurones (schéma de style FCNN)

A présent, nous connaissons la forme externe du réseau de neurones. Il nous faut désormais créer son architecture interne pour qu'il puisse être en mesure d'identifier le plus efficacement possible les images qu'il devra traiter. Les modèles de Machine Learning sont particulièrement efficaces pour ça, grâce à leur nombreuse couche cachée successives, ils permettent d'obtenir de très bons résultats. Dans le cadre de notre projet, nous ne pouvions pas inclure un modèle dit de Deep Learning, puisque qu'ils sont particulièrement profond et donc trop complexe pour la puissance d'un ESP32.

Nous nous sommes donc basés sur l'architecture la plus largement utilisée aujourd'hui dans le domaine de la Computer Vision, il s'agit des réseaux de neurones à convolution (CNN). Suivant les recommandations et en fonction de notre ESP32, nous avons choisi de modéliser un réseau composé de 3 couches successives de convolution et de pooling qui sont détaillées sur la page suivante.

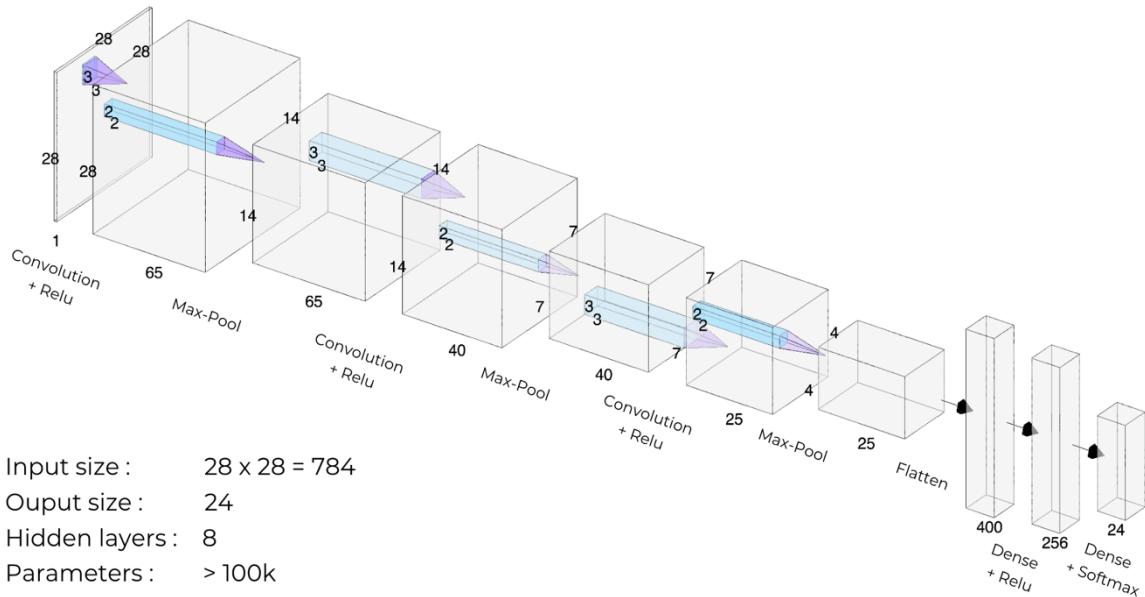


Figure 32 - Architecture du réseau neuronal à convolution (schéma de style AlexNet)

Un modèle CNN se décompose principalement en deux parties. La première est l'étape de convolution, le CNN applique plusieurs filtres (aussi appelés "kernels") de petite taille sur l'image d'entrée. Chaque filtre parcourt l'image pixel par pixel en effectuant une multiplication élémentaire entre ses valeurs et les valeurs correspondantes de la région de l'image où il se trouve. Cette opération génère des "cartes de caractéristiques" en mettant en évidence des traits spécifiques tels que les bords, les textures, ou d'autres motifs saillants. Après la convolution, le CNN utilise souvent des couches de pooling pour réduire la dimension spatiale des cartes de caractéristiques. Le pooling regroupe des régions de la carte de caractéristiques et ne garde que les valeurs maximales ou moyennes, ce qui permet de réduire la taille spatiale tout en conservant les caractéristiques les plus importantes. Cela rend le réseau moins sensible aux variations mineures dans l'image tout en conservant les informations cruciales.



Figure 33 - Exemple d'une des 3 couches Convolution + ReLu + Max-Pooling (la première) dont les caractéristiques correspondent à la première couche de notre CNN à l'exception de la taille de l'image d'entrée

Sur l'exemple de la figure ci-dessus de Convolution + ReLu + Max-Pooling, seul la taille de l'image d'entrée n'est pas représentative pour faciliter l'illustration. De même, le filtre de Kernel affiché a pour effet de décaler les données mais n'est qu'un exemple choisi arbitrairement et qui n'a pas de réel sens. Le filtre de Kernel fait la somme de la zone sélectionnée pour la convolution pondérée aux poids du filtre, une moyenne est ensuite faite, d'où le 9 dans l'exemple. La couche Relu puis Max-Pooling ont ensuite pour effet respectif de supprimer les valeurs négatives et de réduire la taille de la donnée en ne gardant que le maximum d'une zone définie.

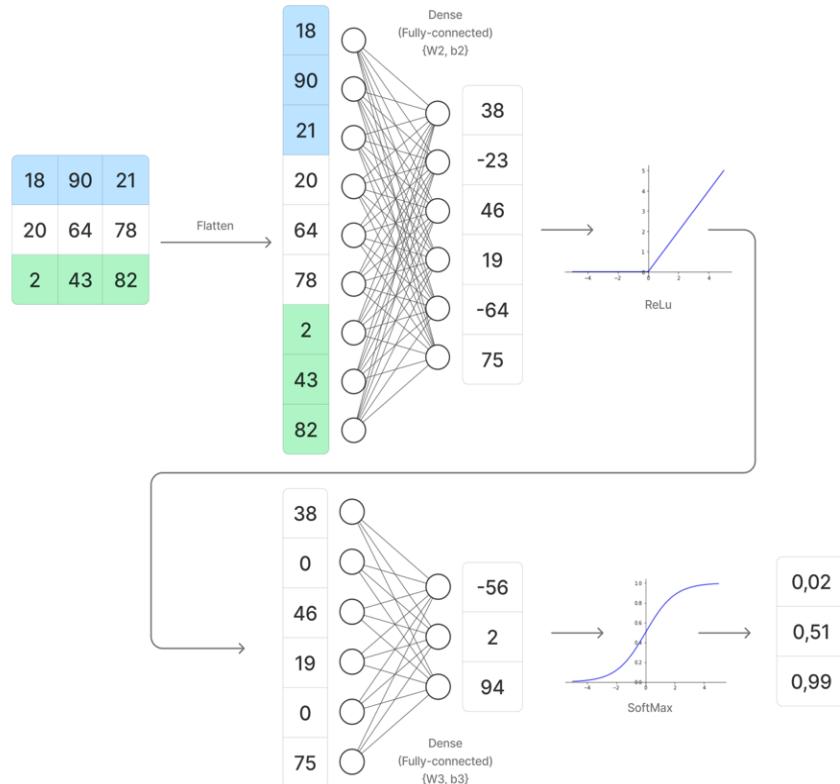


Figure 34 - Exemple d'une couche Flatten + Dense + ReLu + Dense + SoftMax dont les caractéristiques correspondent à la dernière couche de notre CNN à l'exception de la taille des données traitées

La seconde étape, après avoir appliqué des couches de convolution et de pooling, consiste à aplatisir (flattened) les informations résultantes. Cela signifie que les cartes de caractéristiques bidimensionnelles sont converties en un vecteur unidimensionnel. Cette étape est essentielle pour connecter ces données à des couches de neurones entièrement connectées (fully-connected layers) pour la classification finale. Au fur et à mesure que l'on converge vers la dernière couche de neurones que sont les labels, on réduit la taille des couches intermédiaires pour que les caractéristiques correspondantes convergent vers les labels associés pondérés aux poids du réseau. Sur l'exemple de la figure ci-dessus, seul la taille des données traitées ne correspond pas aux tailles réelles de notre modèle encore une fois.

Maintenant que nous avons le modèle complet de notre CNN, s'il on est initié aux réseaux de neurones de base dans le style FCNN où l'on peut voir différentes couches de neurones se succéder, alors nous pouvons considérer que dans le cas présent nous avons déjà défini les couches, le nombre de neurones

de chacune, les liaisons entre ces différentes couches et les fonctions d'activations associées (ReLU, SoftMax). Désormais il nous reste à entraîner le modèle pour déterminer les poids W et biais b de chaque neurone. En réalité l'architecture CNN induit que certains poids et biais sont déjà déterminés en raison de l'architecture de convolution elle-même.

Il y a trois zones dans notre CNN où des poids et biais nécessitent d'être entraîné via notre data set. La première zone correspond aux filtres de Kernel utilisés sur les étapes de convolution qui sont nommés W_1 et b_1 sur le schéma d'une étape de convolution, ils sont au nombre de $65 + 40 + 25$ soit 130 filtres au total et de tailles différentes en fonction de la couche. En effet, nous ne pouvons pas à l'avance prédire quels filtres seront efficaces pour représenter les caractéristiques qui permettront de prédire les labels correspondants.

Les deux autres zones de poids et biais se situent au niveau de l'étape final après aplatissement des données, $\{W_2, b_2\}$ et $\{W_3, b_3\}$ se situent tout deux sur les deux couches de « fully-connected ». De même, une fois les cartes de caractéristiques finales sur une même et unique dimension, l'entraînement déterminera la pondération à accorder de manière à converger vers les bons labels.

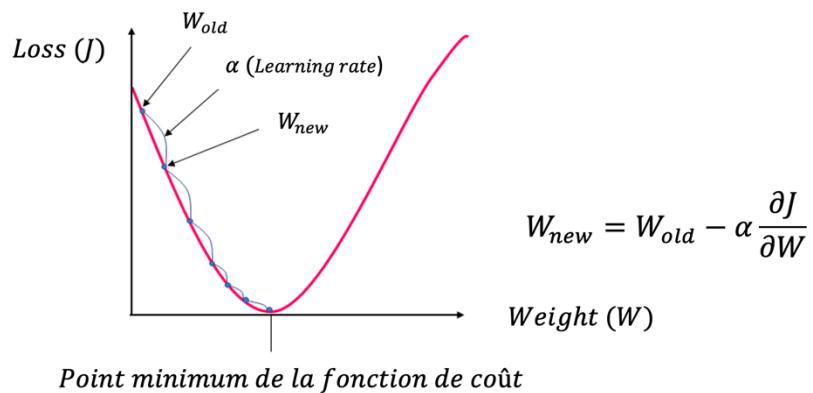


Figure 35 - Algorithme de descente du gradient utilisé pour entraîner le CNN

Sur la figure ci-dessus est illustré le fonctionnement de l'algorithme qui nous permettra de déterminer $\{W_1, b_1\}$, $\{W_2, b_2\}$ et $\{W_3, b_3\}$. Les données d'entraînement sont fournies au réseau durant l'entraînement et on cherche à déterminer le minimum de la fonction de coût, qui en pratique correspond aux meilleurs poids et biais qui permette de prédire le plus justement possible les labels des données d'entraînement. L'entraînement se fait en plusieurs périodes comme nous le verrons dans une partie suivante, le taux d'apprentissage α est automatiquement ajusté entre chaque période, pour cela l'algorithme regarde à partir des données de validation si notre modèle semble gagner en précision, le cas échéant alors le taux d'apprentissage sera diminué pour ne pas « sauter » le minimum local.

4.4. Technologies utilisées

Phase d'entraînement sur Jupyter (PC) :

Nous avons utilisé un notebook Jupyter sur nos machines pour entraîner le réseau de neurones, l'avantage étant que le script du notebook s'exécute de manière séquentielle, on peut donc procéder par étape, afficher des graphes et déboguer facilement. Pour la construction du modèle nous avons utilisé l'API Python de Keras, qui est sans doute l'une des plus populaires, notamment pour la construction de CNN. Enfin, après l'entraînement, le modèle est converti en utilisant TensorFlowLite Converter en un format binaire (model.cpp), ce qui le rend interprétable par l'API C++ de TensorFlowLite pour être déployé sur des appareils mobiles ou embarqués avec des ressources limitées comme c'est le cas pour notre ESP32.

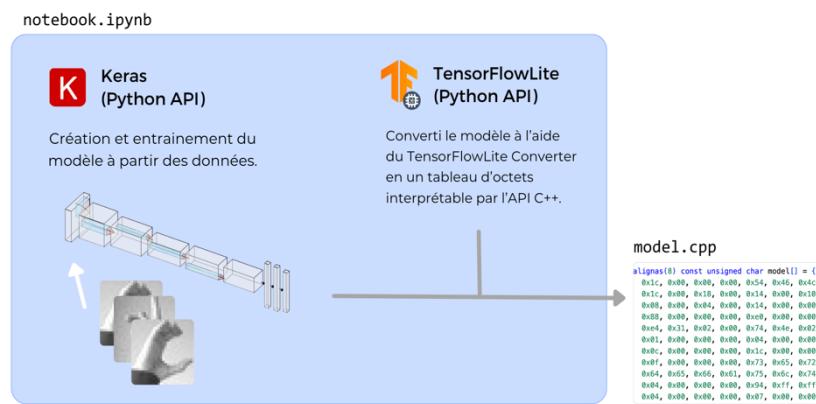


Figure 36 - Processus et technologies utilisées pour l'entraînement du modèle

Phase d'inférence sur ESP32 :

Le firmware codé en C++ de notre drone utilise l'ESP32 Software Development Kit (SDK) pour capturer une image avec la caméra périphérique de l'ESP32-CAM, adapter les données (réduction de taille, conversion en niveaux de gris), et ensuite mettre les données en entrée du modèle. L'API C++ de TensorFlowLite est ensuite utilisée pour interpréter le modèle converti (model.cpp), en définissant les différentes couches du réseau de neurones et en réalisant l'inférence avec les nouvelles données d'entrée pour obtenir la classification finale.

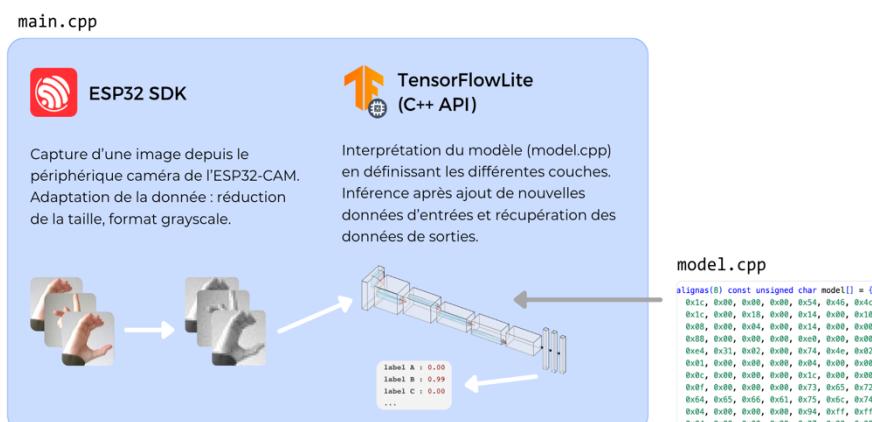


Figure 37 - Processus et technologies utilisées pour l'inférence du modèle

4.5. Résultats d'entraînement

Au niveau de la répartition des données d'entraînement, comme l'illustre la figure ci-dessous le nombre d'images pour chaque label est très similaire et assez conséquent, ce qui nous a permis d'obtenir un modèle suffisamment entraîné et dont aucun label ne sera surentraîné par rapport aux autres.

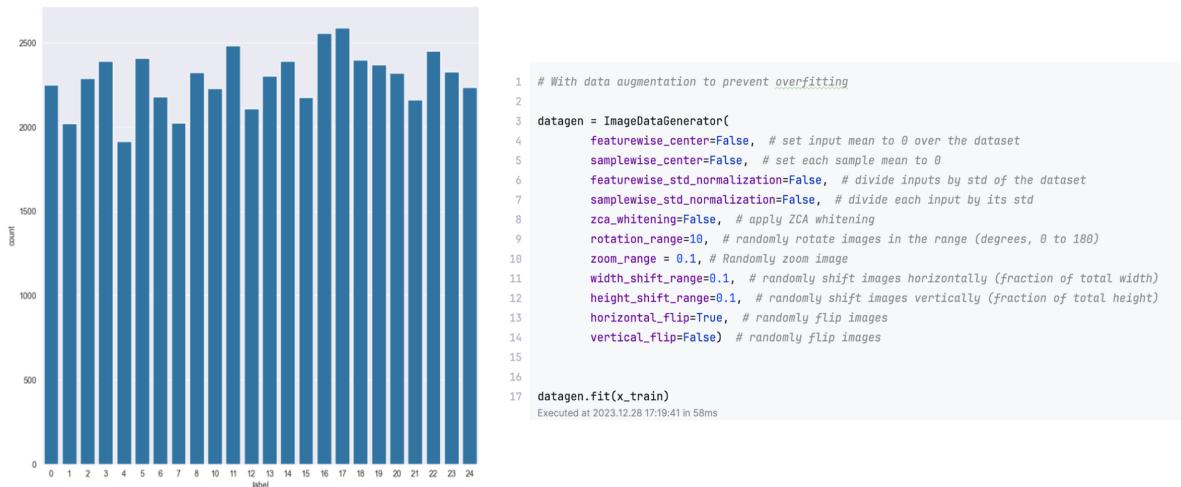


Figure 38 - Répartition des données d'entraînement et augmentation artificielle

L'entraînement par l'algorithme de descente s'effectue ensuite dans notre cas en 20 étapes (4 minutes au total). Entre ses 20 étapes, l'algorithme va réajuster son taux d'apprentissage à mesure qu'il gagne en précision, dont il a connaissance grâce aux données de validation. La précision de validation suit de près la précision d'entraînement, ce qui est bon signe qu'il n'y a pas de surajustement (overfitting) significatif. Aussi, la perte de test (données de validation) reste proche de la perte d'entraînement et se stabilise vers les dernières époques, ce qui suggère que le modèle généralise bien et n'est pas en surajustement. De même, la perte de test basse et stable en conjonction avec une haute précision de test indique que le modèle performe bien sur de nouvelles données non vues durant l'entraînement.

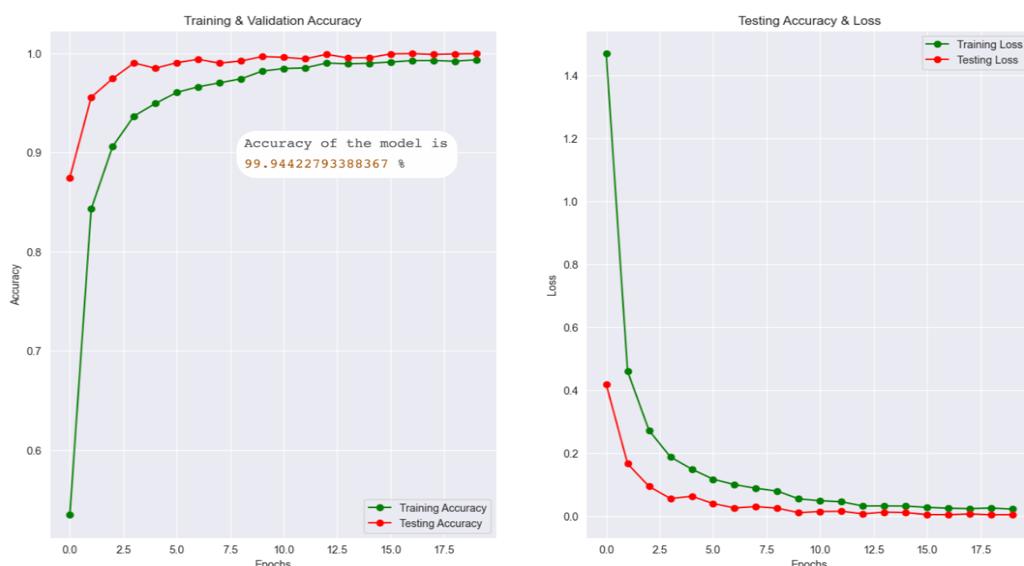


Figure 39 - Courbes d'apprentissage du modèle

L'entraînement fini nous pouvons alors tester la précision finale de notre modèle à partir des données de test. La matrice de confusion de la figure ci-dessous représente le nombre de fois qu'un label est identifié pour un autre label (lui compris). Si nous prenons la ligne 1 comme exemple, nous pouvons lire que sur le total des images de test comportant le label 1 (A = 0, B = 1...), la totalité soit 864 ont été classifiés comme étant du label 1, et aucune image n'a été classifiée sur un autre label. Ainsi, sur la base de cette matrice de confusion, le modèle semble avoir une tendance à bien classer les exemples, comme en témoignent les nombres relativement élevés sur la diagonale principale par rapport aux nombres hors diagonale.

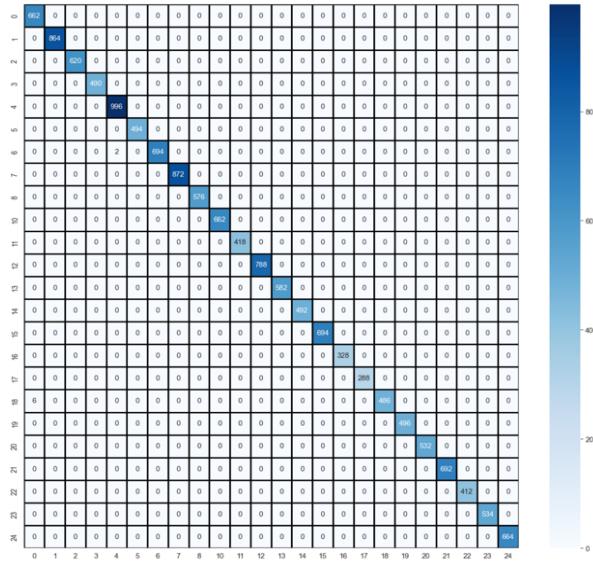


Figure 40 – Matrice de confusion post-entraînement

4.6. Résultats sur ESP32

Sur la figure de la page suivante, toujours sur Jupyter nous utilisons de l'API Python TensorFlow pour transformer notre modèle Keras en un modèle TFLite. Nous utilisons l'optimiseur par défaut pour convertir le modèle au format binaire et améliorer les performances. Les types d'entrées et de sorties sont définis comme uint8 et float32 pour être interprétés comme suit sur le firmware C++ du drone.

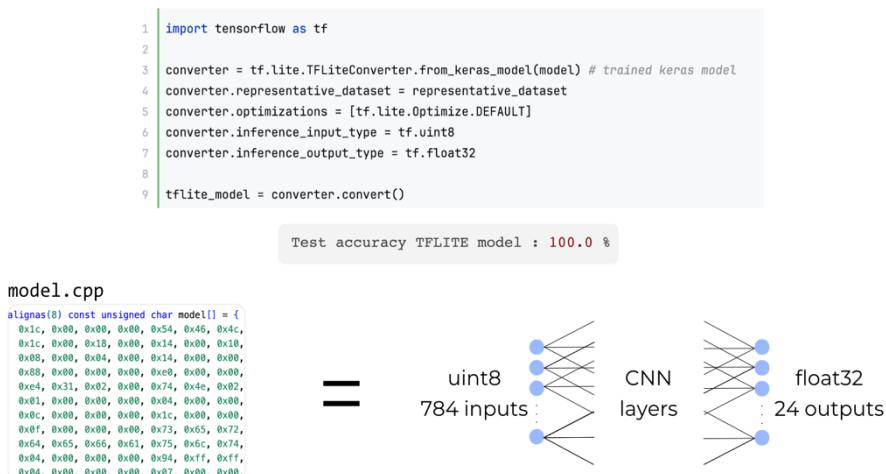


Figure 41 - Code de conversion en paquet d'octets et d'interprétation du modèle via TensorFlowLite

Une fois que nous avons développé la partie IA du firmware de l'ESP32-CAM à partir de l'API C++ de TensorFlowLite, nous testons le fonctionnement et la précision du modèle en faisant plusieurs inférences pour chaque label et lorsque rien ne se trouve à la caméra.

Lettre	Essai1	Essai2	Essai3	Essai4	Essai5	Essai6	Essai7	Essai8
Rien	H	G	G	G	H	G	H	H
A	G	X	P	E	G	G	X	G
B	X	X	X	X	X	X	B	X
C	C	C	C	C	H	C	C	C
D	T	T	T	T	D	T	D	D
E	H	X	X	X	H	H	X	X
F	F	F	F	F	F	F	F	G
G	H	H	H	H	H	H	H	H
H	H	H	H	H	H	H	H	H
I	D	T	D	T	T	D	T	T
K	T	T	T	T	T	T	K	T
L	L	L	L	H	L	L	L	L
M	H	X	T	Q	O	X	T	Q
N	Q	X	T	H	H	H	T	H
O	O	O	O	O	C	O	C	C
P	H	H	H	H	H	H	H	H
Q	P	P	P	P	P	Q	P	P
R	T	D	K	K	K	D	K	K
S	O	O	X	X	X	O	O	X
T	P	T	T	T	T	P	H	T
U	R	R	R	R	R	G	R	R
V	K	K	K	K	T	T	K	K
W	Y	Y	Y	K	W	W	Y	K
X	O	H	X	X	X	H	X	O
Y	T	Y	Y	Y	Y	Y	Y	Y

Figure 42 - Tests effectués avec l'ESP32-CAM dans un milieu clair

Ainsi, comme on peut l'observer sur la figure ci-dessus, dues à de nombreuses causes de pertes de précision expliquées plus en détails par la suite. Seul une poignée de labels que sont C, O, L et Y sont réellement performants, le C et le O se ressemblant fortement.

4.7. Optimisations apportées

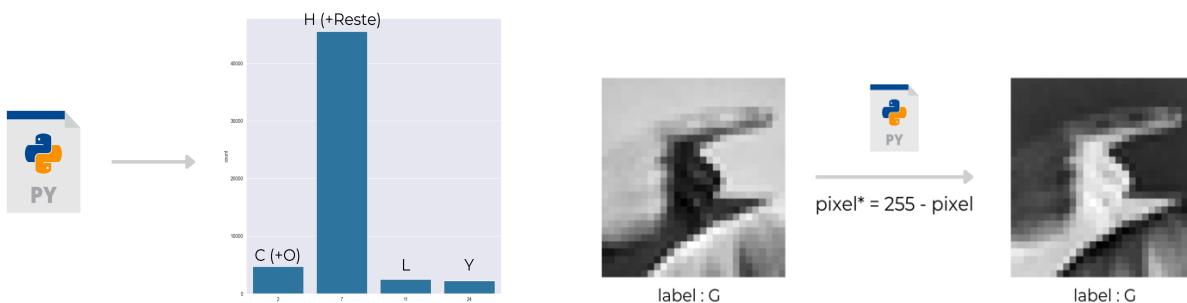


Figure 43 - Optimisations apportées (réajustement des labels + duplication négative des données)

À la suite des résultats pratiques, nous décidons de réajuster les labels en ne gardant que C, O, L et Y dues à leur exceptionnel performance, C et O étant sous le même label car leurs signes sont extrêmement proches. Pour le reste nous décidons de mettre tous les autres labels sur un label « poubelle » qui détectera tout autre signe ou lorsqu'il n'y aura rien. L'un des problèmes majeurs de notre data set est en effet le fait qu'il soit entraîné pour toujours reconnaître un signe, le cas où il n'y a rien n'étant pas prévu.

Le second soucis que nous avons constaté est le fait que le modèle est fortement sensible aux conditions d'éclairage et aux différents contrastes présents sur l'image. Ce biais vient essentiellement du fait que nos données d'entraînement sont beaucoup trop uniformes. L'ensemble des images sont toutes prises avec une main claire, dont la manche est sombre et dont le fond est gris et sans aucun détails gênants. Ainsi, dupliquer l'ensemble de nos données en créant une version négatives permet au moins de pallier en partie ce biais de manque d'exhaustivité, en addition de l'augmentation artificielle qui sera opérée en début de script comme présenté précédemment. Deux scripts Python nous permettent ainsi d'opérer très facilement les optimisations de la figure ci-dessus sur nos fichiers de données CSV.

4.8. Résultats finaux sur ESP32

Nous réentraînons ainsi une nouvelle fois notre modèle pour obtenir le modèle final, qui cette fois-ci ne contient plus que 4 sorties (C+O, H+Reste, L et Y), ce qui permettra à notre drone d'être capable de différencier 3 signes. La précision du nouveau modèle est encore une fois extrêmement satisfaisante après avoir testé le modèle avec les données de test. Les courbes ne semblent pas non plus présenter de problème de surajustement (overfitting, le terme n'a pas de lien avec le réajustement des labels opéré !).

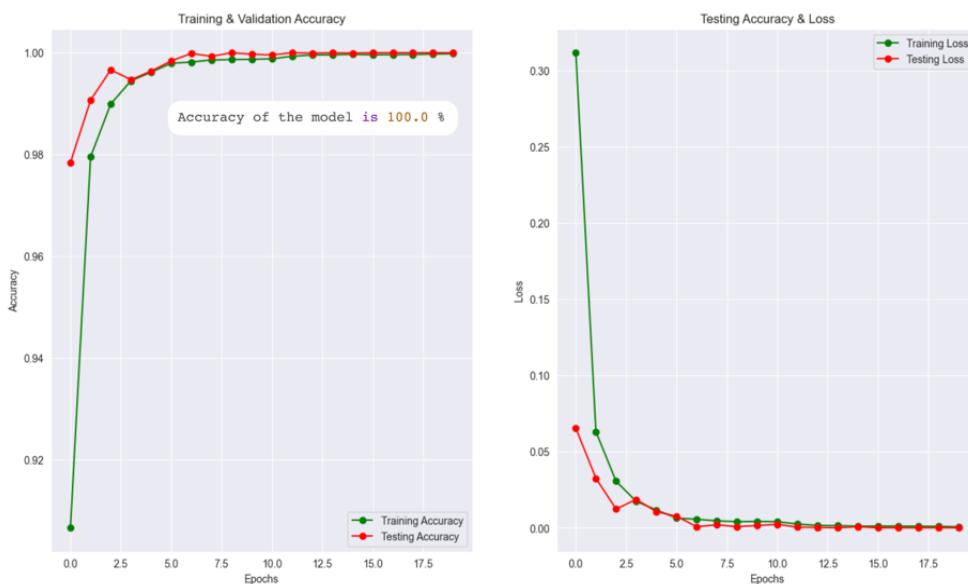


Figure 44 - Courbes d'apprentissage pour le modèle entraîné à partir des données post-optimisation

Enfin, sur les matrices de confusions ci-dessous, nous pouvons observer les résultats finaux après entraînement donc à partir des données de test, et à partir de test réel sur l'ESP32-CAM. Notre modèle a une précision finale proche de 100% durant l'entraînement, chaque label est correctement classifié.

Pour ce qui est des résultats réels, nous sommes satisfaits d'observer que nos optimisations ont portées leur fruit. En effet, lorsqu'aucun signe n'est réalisé devant la caméra de l'ESP32, nous ne sommes jamais parvenus à ce qu'un autre signe parmi C+O, L ou Y ne soit détecté, et de même lorsque nous décidons de réaliser un signe que nous avons écarté lors du réajustement des labels. Ce critère est très important car cela signifie qu'il n'y aura jamais de « faux-vrais », ce qui aurait été contraignant si notre drone se mettait à changer son comportement sans que personne ne lui en donne l'ordre. Concernant les 3 labels restants que nous avons laissé détectable, leur précision n'est pas de 100%, mais après plusieurs heures de test dans un milieu relativement éclairé, nous avons obtenu des précisions de l'ordre de 70% dans chaque cas. Au vu du data set à partir duquel le modèle est entraîné et au vu de la qualité de notre caméra, nous sommes particulièrement satisfaits de ses résultats.

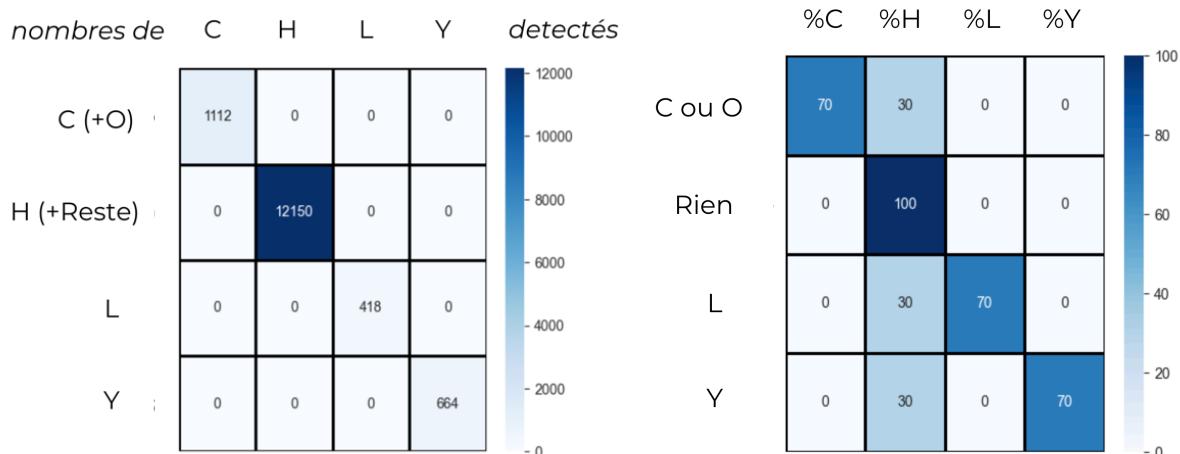


Figure 45 – Matrices de confusion finales, post-entraînement (gauche), et à partir de test réel sur l'ESP32 (droite)

4.9. Analyse de la perte de précision

Dans cette dernière partie concernant le modèle de reconnaissance gestuelle, nous nous proposons d'analyser les différentes étapes impactant la précision de la classification finale du modèle comme l'illustre la figure ci-dessous.

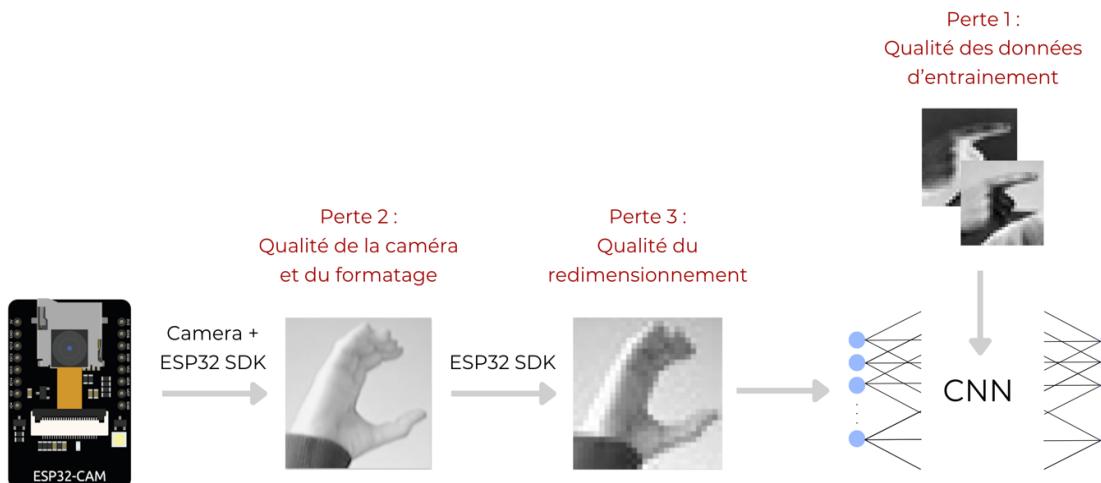


Figure 46 - Schéma bloc des différentes étapes impactant la précision de la classification des données

Premièrement, le modèle en lui-même n'est pas parfait comme nous avons pu le voir et comprend de nombreux biais dû à l'hyper-uniformité des données d'entraînement. Mais également sur les données de test et de validation, ce qui explique que durant l'entraînement et après, la précision est estimée proche de 100%. Malgré l'augmentation artificielle des données et des optimisations que nous avons opérées manuellement, le modèle aura pu être bien plus efficace. Dans l'idéal, constituer son propre data set permettrait d'obtenir les meilleurs résultats pour le modèle souhaité, mais comme détaillé précédemment, nous n'avons pas pu le faire par faute de moyen.

Ensuite, la qualité de notre caméra impacte beaucoup la classification prédictive pendant l'inférence. La caméra est de médiocre qualité, les détails sont peu lisibles et la caméra a tendance à être surexposée sous forte luminosité ou à l'inverse manquer de contraste en plus faible luminosité. Enfin la qualité supérieure des données d'entraînement, induit le modèle lors de la phase d'entraînement à chercher des caractéristiques qu'il peine parfois à retrouver sur les images capturées à l'aide de l'ESP32-CAM.

Enfin, c'est sans doute l'étape la moins impactante, mais nous ne pouvons pas écarter le fait que l'algorithme de redimensionnement de l'image puisse faire perdre des détails essentiels. Concrètement, nous utilisons le SDK de l'ESP32, donc un algorithme particulièrement efficace et qui utilise la conversion JPEG ; néanmoins il se peut que cela joue un rôle, bien que sans doute minime.

Bibliographie

GitHub repository et sites web :

espressif. esp-dl [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://github.com/espressif/esp-dl>

espressif. arduino-esp32 [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://github.com/espressif/arduino-esp32>

sparkfun. Sparkfun_VL53L5CX_Arduino_Library [en ligne]. [consulté le 12/01/2024]. Disponible sur :
https://github.com/sparkfun/SparkFun_VL53L5CX_Arduino_Library

johnathanrandall. American-Sign-Language [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://github.com/jonathanrandall/American-Sign-language>

stm32duino. VL53L5CX [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://github.com/stm32duino/VL53L5CX>

tanakamasayuki. Arduino_TensorFlowLiteESP32. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
https://github.com/tanakamasayuki/Arduino_TensorFlowLite_ESP32

Moddingear. esp32-esc. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://github.com/Moddingear/esp32-esc>

me-no-dev. ESPAsyncWebServer. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://github.com/me-no-dev/ESPAsyncWebServer>

CarbonAeronautics. Guide for manual quadcopter. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://github.com/CarbonAeronautics>

Arduino. Wire library documentation. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://www.arduino.cc/reference/en/language/functions/communication/wire/>

TensorFlow. TensorFlow Lite for Microcontrollers. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://www.tensorflow.org/lite/microcontrollers?hl=fr>

TECPERSON. Sign Language MNIST. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://www.kaggle.com/datasets/datamunge/sign-language-mnist>

Datasheet :

Bosch. BMP280 Digital Pressure Sensor. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://cdn-shop.adafruit.com/datasheets/BST-BMP280-DS001-11.pdf>

InvenSense. MPU-6000 and MPU-6050 Register Map and Descriptions. [en ligne]. [consulté le 12/01/2024]. Disponible sur : <https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf>

SpeedyBee. F405 Mini BLS 35A 20x20 Stack User Manual V1.0. [en ligne]. [consulté le 12/01/2024]. Disponible sur : https://store-fhxhuiq8q.mybigcommerce.com/product_images/img_SpeedyBee_F405_Mini_BLS_35A/SpeedyBee-f405-mini-stack-Manual-en.pdf

BetaFPV. 450mAh 4S 75C Lipo Battery. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://betafpv.com/products/450mah-4s-75c-lipo-battery?variant=14260465926188>

CultureFPV. Frame HIGHT CULT. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://culturefpv.fr/wp-content/uploads/2022/10/HIGHCULT-plan-de-montage.pdf>

STMicroelectronics. VL53L5CX Time of Flight documentation. [en ligne]. [consulté le 12/01/2024]. Disponible sur : <https://www.st.com/en/imaging-and-photonics-solutions/vl53l5cx.html#documentation>

MATEKSYS. MICRO BEC 6-30V TO 5V/9V-ADJ. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<http://www.mateksys.com/?portfolio=mbec6s>

Joy-It. KY-012 Active Piezo-Buzzer module. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://datasheetspdf.com/pdf-file/1402025/Joy-IT/KY-012/1>

Joy-It. KY-009 RGB LED SMD module. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
<https://datasheetspdf.com/pdf-file/1402022/Joy-IT/KY-009/1>

IFlight. 4 paires Nazgul F5/5140 5 pouces 3 lames/hélice Leic-blade Compatible XING-E moteur 2207 pour RC FPV Racing Drone Part. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
https://fr.aliexpress.com/item/1005006097396637.html?spm=a2g0o.productlist.main.7.4ab23de6ZwHb5Q&algo_pvid=6ed45175-d098-4699-a7f3-955d7a72cae5&algo_exp_id=6ed45175-d098-4699-a7f3-955d7a72cae5-3&pdp_npi=4%40dis%21EUR%2113.42%210.49%21%21%21102.74%21%21%402101efec17042795053514061eac22%2112000035726124215%21sea%21FR%210%21AB&curPageLogUid=IYcFT3ewYDbX#nav-description

AXISFLYING. 2207 AE-2207 1850KV 1960KV [en ligne]. [consulté le 12/01/2024]. Disponible sur :
https://fr.aliexpress.com/item/1005005783207749.html?srcSns=sns_Copy&spreadType=socialShare

[&bizType=ProductDetail&social_params=60445296717&aff_fcid=6c4df2c4189d4db7bfffab046591a89ff-1702824511770-08309-](#)
[EGbLdPT&tt=MG&fbclid=lwAR3JpV6F0fYFE9caQSBIbfshHm9G1FH3SCUalwvzzUg9sgQgg0V_Jh3FU](#)
[Y&aff_fsk= EGbLdPT&aff_platform=default&sk= EGbLdPT&aff_trace_key=6c4df2c4189d4db7bfffab046591a89ff-1702824511770-08309-](#)
[EGbLdPT&shareId=60445296717&businessType=ProductDetail&platform=AE&terminal_id=6d332ebd493f46269a8f3a69d669406e&afSmartRedirect=y#nav-specification](#)

Firmware :

Wikipédia. Serial Peripheral Interface. [en ligne]. [consulté le 12/01/2024]. Disponible sur :
https://fr.wikipedia.org/wiki/Serial_Peripheral_Interface

Wikipédia. I2C. [en ligne]. [consulté le 12/01/2024]. Disponible sur : <https://fr.wikipedia.org/wiki/I2C>

Modèle CNN :

Vancappel K. Deep Learning : Le réseau neuronal convolutif (CNN). Le Blog Business & Decision. [en ligne]. [consulté le 12/01/2024]. Disponible sur : <https://fr.blog.businessdecision.com/tutoriel-deep-learning-le-reseau-neuronal-convolutif-cnn/>

Asservissement :

Shi X. Cours d'AU2. INSA Lyon.

GE INSA Lyon. TP Drone AU1. INSA Lyon.