



IMT Nord Europe
École Mines-Télécom
IMT-Université de Lille

Rapport technique de l'application mobile et du serveur

Noah Châtelain
Étudiant année M1
FISA-TI 23
Année 2021-2022

IMT Nord Europe
Rue Guglielmo Marconi
59650 Villeneuve-d'Ascq

Domaine Numérique
UV Mobile et Vision
Encadrants universitaire
Alexis GUILLOTEAU
Didier JUGE-HUBERT



Projet mobile réalisé en 2^{ème} année
Le 31/01/2022

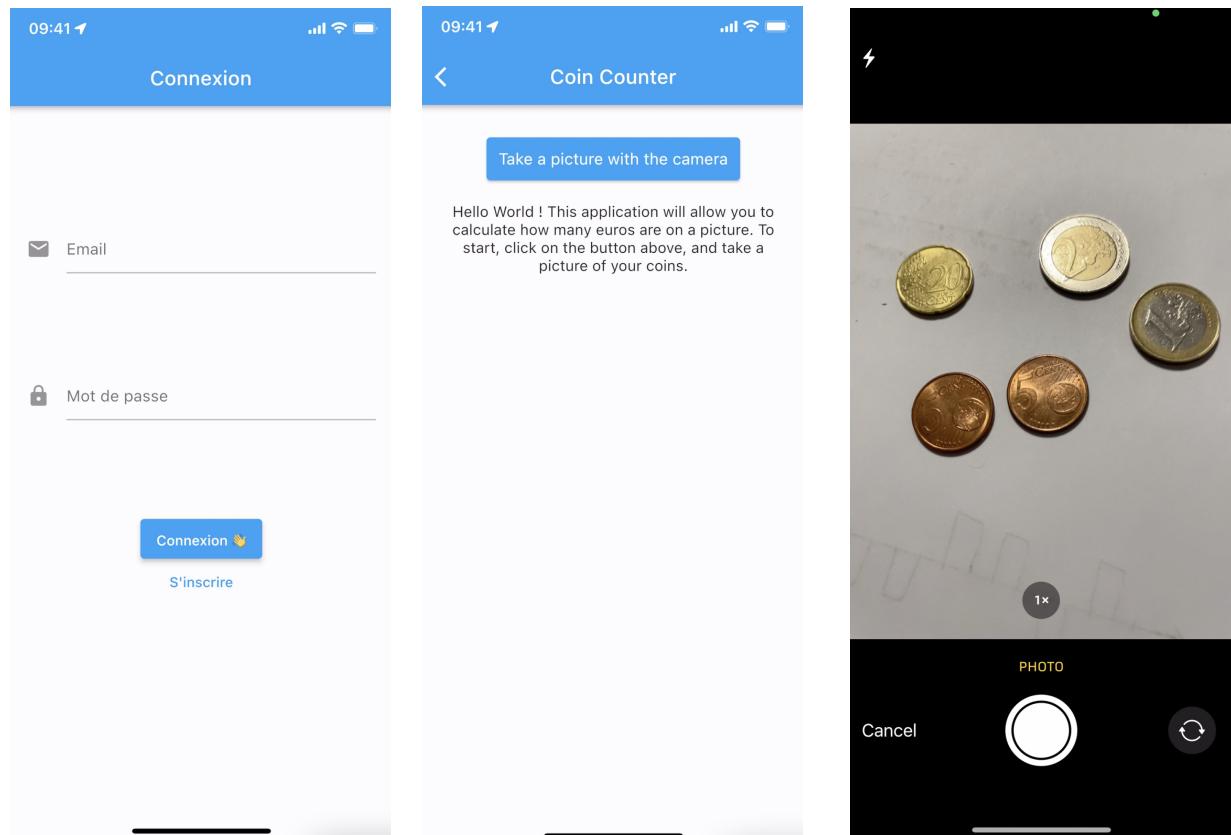
Table des matières

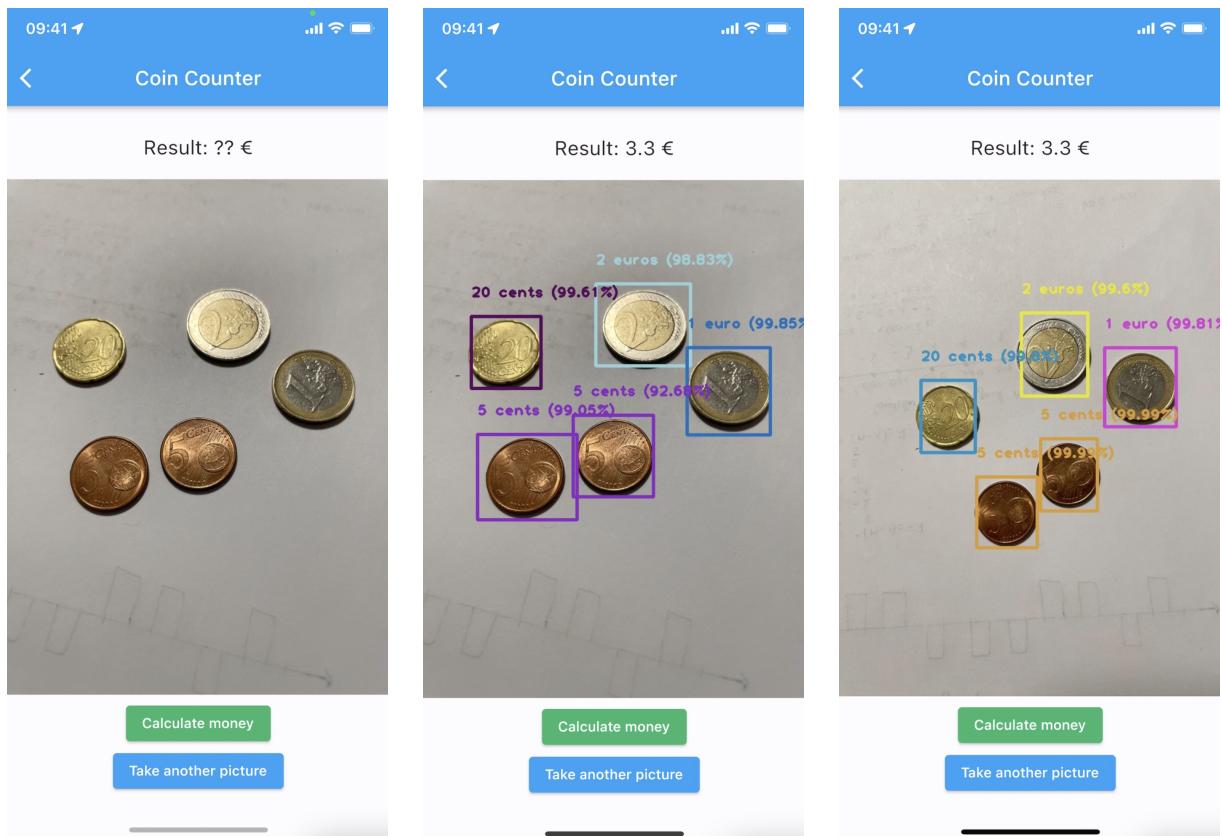
Introduction du projet à réaliser	2
Développement de la partie Web Server (Python)	3
2.1 Choix de l'algorithme YOLO	3
2.2 Réalisation du dataset	5
2.3 Entraînement du modèle	5
2.4 Traitement des requêtes reçues	9
2.4.1 Requête de traitement d'image et recherche de pièces	9
Développement de la partie Mobile (Flutter & Dart)	10
3.1 Les autorisations	10
3.2 Firebase, authentification et inscription	11
3.2 Schéma des communications entre l'application et le serveur	13

1. Introduction du projet à réaliser

Grâce à nos acquis obtenus durant les différents TP Mobile et Vision que nous avons eu à réaliser depuis le début de cette année scolaire, et grâce à des recherches personnelles, nous avons eu à réaliser une application complète accompagnée de son serveur web, qui permettrait à nos utilisateurs de compter le nombre de pièces de monnaie sur une photo prise avec leur smartphone. Nous avons reçu au début du projet un cahier des charges complet, indiquant les fonctionnalités globales que devait respecter notre application.

Voici ci-dessous, le résultat final de l'application, que j'ai développé en essayant de suivre au maximum le cahier des charges. Avec une gestion des utilisateurs avec une authentification et inscription à l'application grâce à un mot de passe chiffré. Une page d'accueil avec une explication du principe et de l'utilisation de cette application, ainsi qu'un bouton pour prendre instantanément une photo grâce au téléphone, qui ouvrira l'interface de la caméra du téléphone.





Une fois la photo prise, l'utilisateur arrive sur une nouvelle page avec la photo qu'il vient de prendre affichée. 2 boutons se trouvent sur cette page, un pour calculer la valeur monétaire en euro présente sur cette photo, et un autre qui permet de prendre une nouvelle photo.

Nous allons désormais voir comment fonctionne techniquement l'application et son serveur web associé. Comment j'ai réalisé l'application ainsi que ses différentes fonctionnalités.

2. Développement de la partie Web Server (Python)

2.1 Choix de l'algorithme YOLO

Nous allons commencer par la partie Web Server et Intelligence Artificielle. Tout d'abord, je voulais une application qui soit rapide à utiliser et qui affiche un résultat quasi instantané de la valeur monétaire présent sur la photo. J'ai donc fait quelques recherches sur internet, et après avoir

découvert de nombreux algorithmes différents, je suis tombé sur l'algorithme YOLO.

YOLO - *You Only Look Once* est un algorithme proposé par Redmond Et. Al dans un article de recherche publié à la conférence IEEE/CVF sur la vision par ordinateur et la reconnaissance de formes en tant que document de conférence, remportant le prix OpenCV People's Choice Award.

Par rapport à l'approche adoptée par les algorithmes de détection d'objets avant YOLO, qui réutilisent les classificateurs pour effectuer la détection, YOLO propose l'utilisation d'un réseau neuronal de bout en bout qui fait des prédictions de boîtes englobantes et de probabilités de classe en même temps.

Suivant une approche fondamentalement différente de la détection d'objets, YOLO obtient des résultats à la pointe de la technologie, surpassant largement les autres algorithmes de détection d'objets en temps réel.

Alors que des algorithmes comme Faster RCNN fonctionnent en détectant les régions d'intérêt possibles à l'aide du réseau de proposition de région, puis en effectuant une reconnaissance sur ces régions séparément, YOLO effectue toutes ses prédictions à l'aide d'une seule couche entièrement connectée.

Les méthodes qui utilisent les réseaux de proposition de région finissent donc par effectuer plusieurs itérations pour la même image, tandis que YOLO s'en tire avec une seule itération.

YOLO est un algorithme beaucoup plus rapide que ses homologues, il possède tous les avantages dont j'avais besoin pour mon application, et son atout principal est qu'il fonctionne jusqu'à 45 FPS. Donc s'il est capable de reconnaître des objets sur des vidéos de 45 FPS, je me suis dis qu'il serait capable de reconnaître très rapidement un objet sur une simple photo.

Selon différentes analyses, YOLO serait légèrement moins précis que les algorithmes Fast RCNN, mais pour des pièces photographiées en gros plan, avec un algorithme bien entraîné, il ne devrait pas y avoir de problèmes.

J'ai donc décidé d'utiliser cet algorithme, dont j'avais déjà entendu parler auparavant et que j'avais envie de tester depuis longtemps, car j'avais déjà vu de nombreuses vidéos impressionnantes sur la détection d'objets avec cet algorithme.

2.2 Réalisation du dataset

Pour réaliser mon dataset, j'ai commencé uniquement par la pièce de 1 euro, afin de tester si j'arrivais tout d'abord à faire comprendre à mon algorithme qu'il y avait une pièce de 1 euro présente sur la photo avant de commencer avec un gros dataset.

Puis une fois que mon algorithme détectait cette pièce, j'ai fait un nouveau dataset avec cette fois-ci une pièce de 5 centimes, une pièce totalement différente de la première pour voir si je pouvais détecter 2 objets de classes différentes sur une seule photo.

Enfin, une fois que cela fonctionnait avec mes 2 pièces, j'ai ensuite rajouté une autre pièce, de 2 euros, qui était ressemblante avec la première de 1 euro. Et mon algorithme comprenait bien qu'il y avait 3 pièces différentes et me les encadrerait sur la photo avec une prédiction assez forte.

L'algorithme YOLO fonctionnait donc bien ici dans mon cas avec des pièces de monnaie différentes et ressemblantes, donc j'ai continué avec celui-ci.

2.3 Entraînement du modèle

L'entraînement du modèle est une partie vraiment très intéressante, mais qui m'a pris beaucoup de temps. Notamment car je n'avais pas beaucoup de connaissances sur le sujet et encore moins sur l'algorithme YOLO. J'ai donc dû m'informer, et apprendre grâce à des explications en lignes et des formations sur l'algorithme YOLO.

Tout d'abord, j'ai dû réaliser un fichier .txt pour chaque image de mon dataset comprenant le numéro de la classe (1 classe par type de pièce) ainsi que les coordonnées x1, x2, x3 et x4 afin de former un carré autour de nos pièces. Afin que l'algorithme YOLO comprenne où se trouvait la pièce sur notre photo et qu'il puisse apprendre pour les détecter dans le futur.

Pour se faire, pour ne pas passer des heures à entrer les coordonnées des 4 coins un à un. Il existe un petit outil open-source développé en python appelé *labelImg* ([github : https://github.com/tzutalin/labelImg](https://github.com/tzutalin/labelImg)) permettant de détourer chaque objet de chaque images d'un dossier facilement et de lui attribuer une classe. La manœuvre reste toujours assez longue et fastidieuse, mais cet outil facilite grandement la tâche.

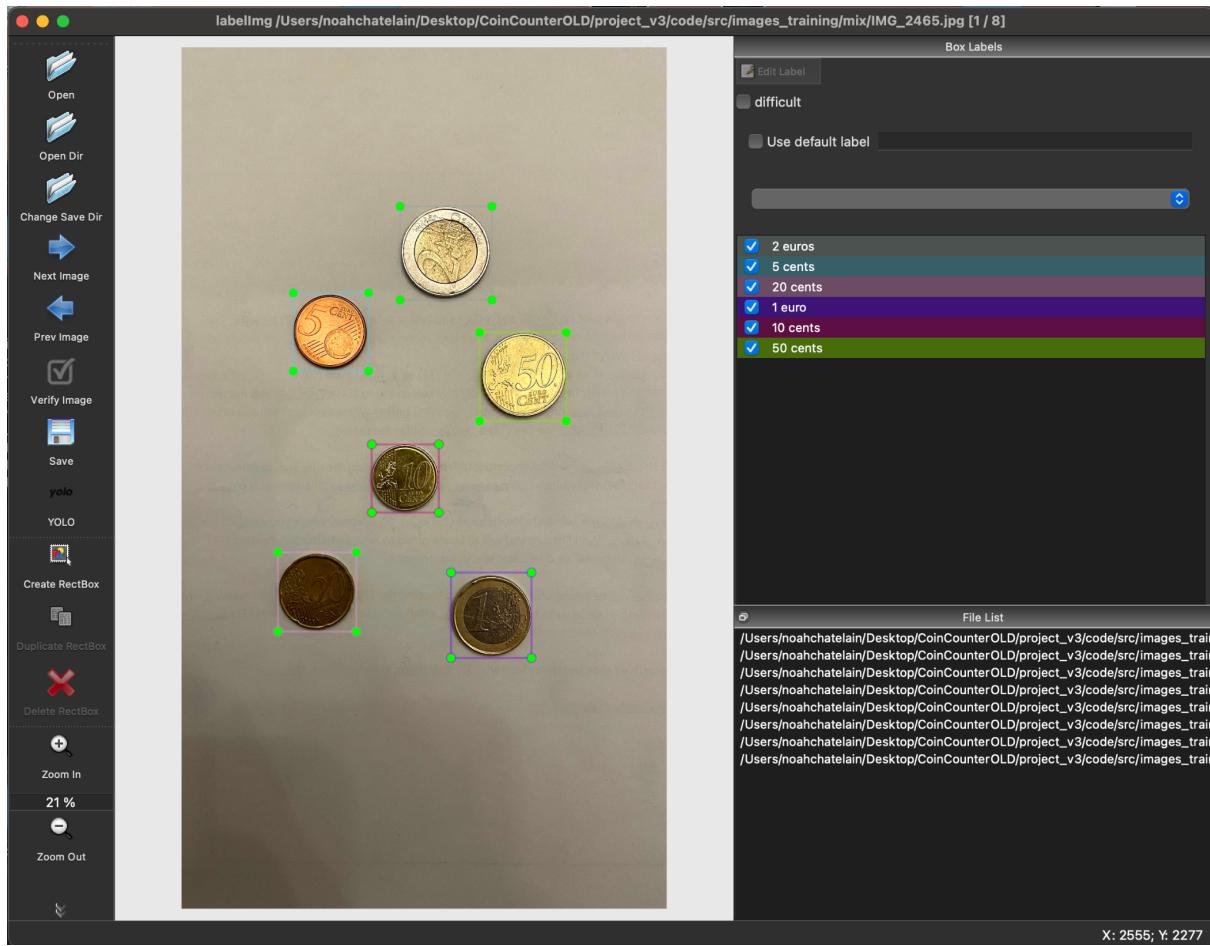


Illustration : Capture d'écran de l'outil open-source *labelImg*

Une fois toutes mes images traitées, j'obtenais pour chaque image son fichier .txt avec les informations nécessaires pour trouver l'image sur celle-ci. Ensuite je crée un fichier zip contenant toutes les images au format jpeg accompagné de tous les fichiers .txt associés. (appelé ici *images.zip*)

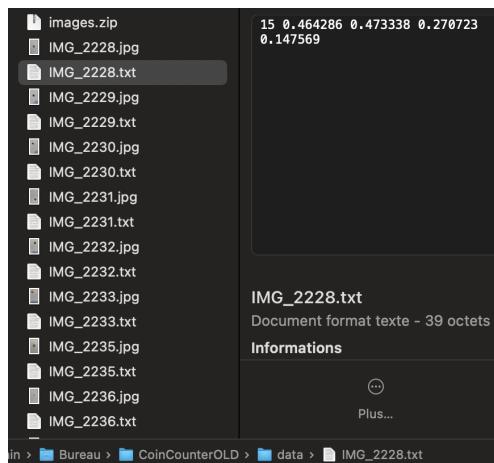


Illustration : Capture d'écran de l'outil open-source *labelImg*

Un des problème auquel j'ai été confronté et que je prenais des photos avec mon iPhone, et donc au format .HEIC, j'ai donc dû réaliser un petit outil CLI en Node.js afin de convertir toutes les images HEIC d'un dossier en JPG dans un dossier d'output. (github de l'outil développé : <https://github.com/Nooaah/heic-to-jpg>)

Une fois le fichier *images.zip* créé, j'ai donc commencé l'entraînement de mon modèle (courant fin novembre 2021). J'ai pu découvrir l'environnement Google Colab que je n'avais encore jamais utilisé, qui est un service permettant de faire tourner des programmes python sur un GPU puissant sur une machine distante, plutôt que de passer par notre machine.

```

Training_CoinCounter_models.ipynb
Fichier Modifier Affichage Insérer Exécution Outils Aide Toutes les modifications ont été enregistrées
+ Code + Texte
const int out_w = (w + 2 * pad - ksize) / stride + 1; // output_width=input_width for stride=1 and pad=1
[ ] #cp cfg/yolov3-tiny.cfg cfg/mask.cfg
<>
[ ] !cp cfg/yolov3.cfg cfg/yolov3_training.cfg
[x]
[ ] !sed -i 's/batch_size=64/' cfg/yolov3_training.cfg
[ ] !sed -i 's/subdivisions=16/' cfg/yolov3_training.cfg
[ ] !sed -i 's/max_batches = 50000/max_batches = 4000/' cfg/yolov3_training.cfg
[ ] !sed -i 's/192 80@classes=80@classes=28/' cfg/yolov3_training.cfg
[ ] !sed -i 's/196 80@classes=80@classes=28/' cfg/yolov3_training.cfg
[ ] !sed -i 's/198 80@classes=80@classes=28/' cfg/yolov3_training.cfg
[ ] !sed -i 's/603 255@filters=218/' cfg/yolov3_training.cfg
[ ] !sed -i 's/689 255@filters=218/' cfg/yolov3_training.cfg
[ ] !sed -i 's/776 255@filters=218/' cfg/yolov3_training.cfg
[ ] !ls -a /mydrive/yolo
[ ] !mkdir /mydrive/yolov4
[ ] total 133985
133985 all_images.zip
mkdir: cannot create directory '/mydrive/yolov4': File exists
[ ] !echo $'5 cents\n10 cents\n20 cents\n50 cents\n1 euro\n2 euros' > data/obj.names
[ ] !echo -e 'classes=6\ntrain = data/train.txt\nvalid = data/test.txt\nnames = data/obj.names\nbackup = /mydrive/yolov4' > data/obj.data
[ ] !mkdir data/obj
mkdir: cannot create directory 'data/obj': File exists
[ ] !wget https://pjreddie.com/media/files/darknet53.conv.74
--2021-11-29 22:19:54-- https://pjreddie.com/media/files/darknet53.conv.74
Resolving pjreddie.com (pjreddie.com)... 128.208.4.108
Connecting to pjreddie.com (pjreddie.com)|128.208.4.108|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 162482580 (155M) [application/octet-stream]
Saving to: 'darknet53.conv.74'

darknet53.conv.74 100%[=====] 154.96M 20.0MB/s in 8.4s
2021-11-29 22:50:03 (18.4 MB/s) - 'darknet53.conv.74' saved [162482580/162482580]

```

Illustration : Capture d'écran de Google Colab

Pour l'entraînement, j'ai passé de nombreuses heures de recherches afin de savoir comment entraîner un modèle grâce à YOLO. La façon la plus "simple" que j'ai trouvé était d'utiliser le réseau de neurones open-source appelé *Darknet* (github du projet open-source *Darknet* : <https://github.com/AlexeyAB/darknet>) qui permet d'implémenter l'algorithme YOLO grâce à nos images, nos coordonnées, et des fichiers de configurations bien précis.

Le fichier de configuration *cfg/yolov3_training.cfg* était assez complexe à renseigner, car il faut absolument que tous les paramètres

soient juste à 100% pour entraîner le modèle, sinon il y aura une erreur. Parmis ces paramètres, il fallait renseigner le nombre de classes que possèdera notre modèle, il faut indiquer aussi le nombre de *max_batch* qui suit une formule précise (*number_of_classes* * 2000) ainsi que les *steps*, qui équivaut à 80% et 90% du *max_batch* et aussi le paramètre *filter* avec la formule suivante : (*number_of_classes* + 5) * 3

```
!nvidia-smi
Mon Nov 29 22:45:07 2021
+-----+
| NVIDIA-SMI 495.44      Driver Version: 460.32.03    CUDA Version: 11.2 |
+-----+
| GPU  Name      Persistence-M | Bus-Id     Disp.A  Volatile Uncorr. ECC | | |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|                               |             |            | MIG M.   |
+-----+
| 0  Tesla P100-PCIE... Off | 00000000:00:04.0 Off |          0 |
| N/A   33C   P0    26W / 250W |           0MiB / 16280MiB |     0%      Default |
|                               |                           |            N/A |
+-----+
+-----+
| Processes:
| GPU  GI  CI      PID  Type      Process name        GPU Memory |
| ID   ID                   ID           Usage
+-----+
| No running processes found
+-----+
```

Illustration : Capture d'écran de la configuration GPU nvidia sur Google Colab

```
./darknet detector train data/obj.data cfg/yolov3_training.cfg darknet53.conv.74 -dont_show
2891: 0.045410, 0.036071 avg loss, 0.001000 rate, 5.122660 seconds, 185024 images, 14.942189 hours left
Loaded: 0.714775 seconds - performance bottleneck on CPU or Disk HDD/SSD
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 82 Avg (IOU: 0.911777), count: 2, class_loss = 0.000682, iou_loss = 0.013620, total_loss = 0.014302
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 94 Avg (IOU: 0.000000), count: 1, class_loss = 0.000000, iou_loss = 0.000000, total_loss = 0.000000
v3 (mse loss, Normalizer: (iou: 0.75, obj: 1.00, cls: 1.00) Region 106 Avg (IOU: 0.000000), count: 1, class_loss = 0.000000, iou_loss = 0.000000, total_loss = 0.000000
```

Illustration : Capture d'écran de l'entraînement du modèle avec Darknet

Une fois mon modèle entraîné à environ 2400 répétitions, j'obtiens un fichier *yolov3_training_2400.weights* qui sera à transmettre à OpenCV grâce à *Deep Neural Network module* dans mon futur code afin qu'il trouve les pièces sur mes images. (lien de mon Notebook d'entraînement : <https://colab.research.google.com/drive/1Ya0mJii8GBK7Ug2gpFRcaV56yRyWmO8A?usp=sharing>)

Au début, j'obtenais de nombreux problèmes et je pensais que cela venait de mon code et j'ai perdu beaucoup de temps. Et mon modèle arrêtait de s'entraîner au bout de 200 répétitions. Je me suis donc renseigné sur quelle perte minimum je devais obtenir pour avoir un résultat correct. J'ai découvert un site (capture d'écran ci-dessous) expliquant qu'il fallait une

perte inférieur à 0.060730 pour arrêter l'entraînement. J'ai donc pris la version supérieure de Google Colab afin de pouvoir laisser tourner toute la nuit, et j'ai atteint ce résultat, et les prédictions des pièces sur mes photos sont devenues très bonnes.

Batch output

Let's have a look at following line first, we'll break it down step by step. The output below is generated in `detector.c` on [this line of code](#).

```
9798: 0.370096, 0.451929 avg, 0.001000 rate, 3.300000 seconds, 627072 images
```

- `9798` indicates the current training iteration/batch.
- `0.370096` is the total loss.
- `0.451929 avg` is the average loss error, which should be as low as possible. As a rule of thumb, once this reaches below `0.060730 avg`, you can stop training.
- `0.001000 rate` represents the current learning rate, as defined in the `.cfg` file.
- `3.300000 seconds` represents the total time spent to process this batch.
- The `627072 images` at the end of the line is nothing more than `9778 * 64`, the total amount of images used during training so far.



2.4 Traitement des requêtes reçues

2.4.1 Requête de traitement d'image et recherche de pièces

Pour la réalisation de la réception d'une requête d'image, j'ai développé la route `@app.route('/send_image', methods=['POST'])` qui :

1. Reçoit l'image
2. Renomme le fichier jpg avec des caractères alphanumériques grâce à la fonction `alphanumeric(length_of_word)`
3. Sauvegarde le fichier dans UPLOAD_FOLDER (= /files_reception)
4. Appelle la classe `object_detection` avec le chemin de cette image en paramètre et le chemin d'output de l'image calculée
5. Retourne la réponse de la classe `object detection` qui est un objet au format JSON contenant le chemin de la nouvelle image `calculated_image_path` dite "calculée" avec les pièces entourées, la valeur monétaire se trouvant sur la photo "coins_total" puis `execution_time` qui permet d'obtenir en combien de temps le calcul a été effectué.

Étant donné que mon serveur tourne sur l'adresse 0.0.0.0 et sur le port 5000, j'appelle l'adresse IP de mon Mac via mon iPhone afin de joindre l'API. Le chemin de l'image calculé est donc :

http://192.168.1.12:5000/get_image/ + self.alphanumeric_filename (qui est le nouveau nom de l'image)

La route `@app.route('/get_image/<filename>')` permet de retourner une image au format JPG. Maintenant, lorsque l'adresse http://192.168.1.12:5000/get_image/V7E2IE9EJ.jpg sera appelée, l'image V7E2IE9EJ.jpg sera retournée, et je pourrais l'afficher dans mon Application grâce au Widget :

```
Image.Network("http://192.168.1.12:5000/get_image/V7E2IE9EJ.jpg")
```

3. Développement de la partie Mobile (Flutter & Dart)

3.1 Les autorisations

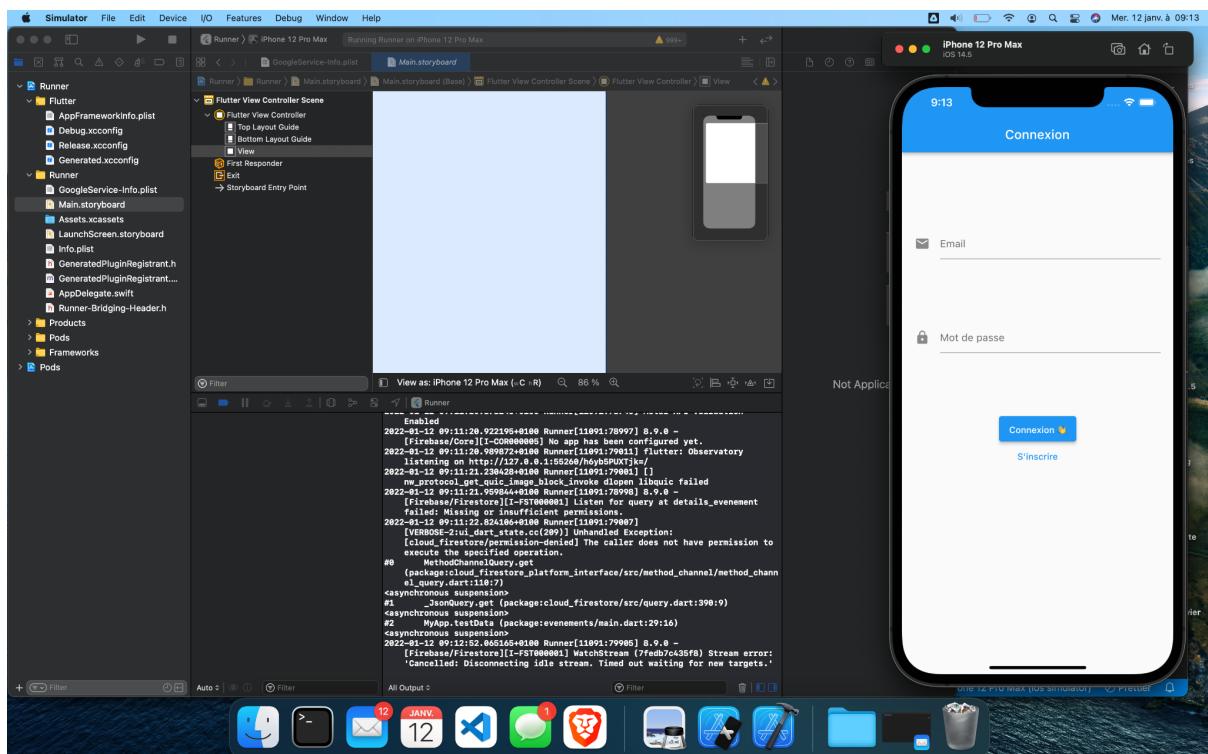


Illustration : Capture d'écran du développement de l'application Mobile

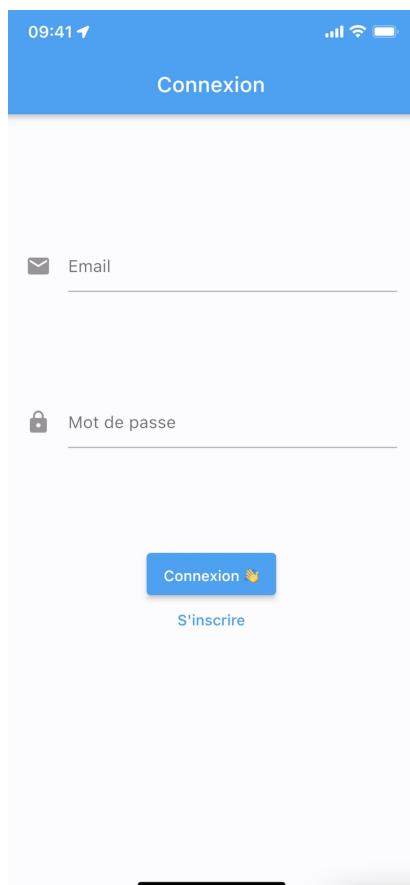
J'ai développé l'application mobile avec Flutter en langage Dart. L'application a été testée avec un iPhone, mais pas sous Android.

Dans le fichier `ios/Runner/Info.plist` j'ai ajouté les autorisations de :

- Caméra (`NSCameraUsageDescription`)
- Micro (`NSMicrophoneUsageDescription`)
- D'accès aux appareils sur le réseau local, pour faire une requête sur l'IP de mon ordinateur où le serveur tourne (`NSBonjourServices`)

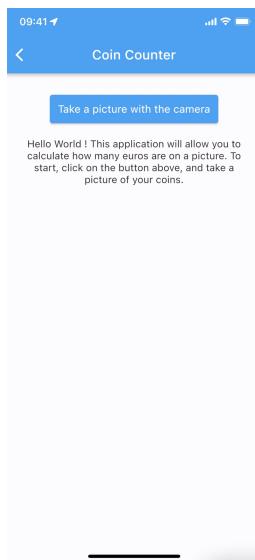
Ces autorisations sont demandées à l'utilisateur à l'ouverture de l'application pour la première fois.

3.2 Firebase, authentification et inscription



Pour lier Firebase à l'application, j'ai créé un nouveau projet Firebase. J'ai récupéré le fichier `GoogleService-Info.plist` pour iPhone (j'ai configuré aussi pour Android mais je n'ai pas testé son bon fonctionnement) où se trouve les informations du projet et les clés d'API pour lier l'application à Firebase, et je l'ai ajouté au dossier `ios/`.

Pour créer une page de connexion fonctionnelle, avec des messages d'erreurs, et avec la possibilité de s'inscrire via son adresse email. Je me suis inspiré d'un des derniers TP Flutter que nous avions eu à réaliser, qui possédait exactement le même fonctionnement.



Ensuite, lorsque l'on est connecté, on arrive sur l'interface avec le bouton pour prendre une photo. Lors du clique, on lance la fonction `_getFromCamera()` qui se chargera de lancer l'interface caméra, enregistrer la photo dans un dossier temporaire du téléphone, puis ajouter ce chemin temporaire à notre variable de type File : `imageFile`

Une fois la photo prise et validée, l'utilisateur obtient une nouvelle page, avec la photo actuelle qu'il vient de prendre. Lorsqu'il n'y a pas encore d'image calculée, c'est cette image qui est affichée, mais lorsqu'il y en a une, alors on affiche l'image avec la méthode Network et le lien vers le serveur de cette image calculée.



Lors du clique sur le bouton Calculer, l'image est envoyée au serveur en multipartFile à la route :

http://192.168.1.12:5000/send_image

qui est l'adresse de mon Mac où tourne le serveur



Le serveur se chargera alors de calculer la valeur monétaire de l'image, d'enregistrer la nouvelle image calculée avec les pièces entourées et de retourner ces informations au format JSON.

L'application va recevoir ce résultat. Il existe donc une image calculée, donc on l'affiche à la place de l'ancienne. Puis on modifie la valeur de Result afin d'afficher la valeur monétaire.

Le bouton pour prendre une autre photo permettra de remettre les valeurs utilisées à null et recommencera le processus.

3.2 Schéma des communications entre l'application et le serveur

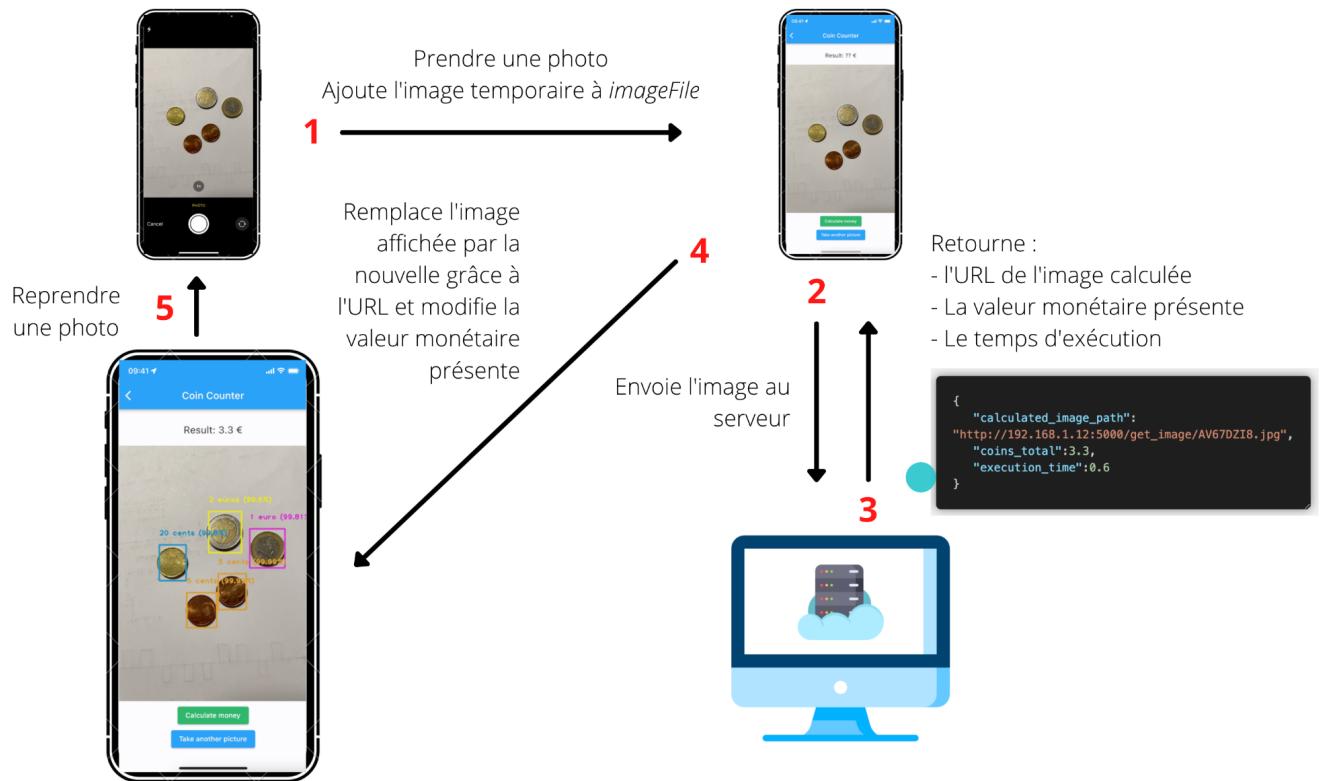


Illustration : Schéma des communication entre l'appli Mobile et le serveur