



ING3-IA

Gimme your plate

Auteurs :

BOUCHER Maëlle
CLAUZEL Daniel
CÔTE Pierre
ELOUALLADI Reda
PLANTÉ Matthieu
GROUPE3

A l'attention de :
M. JACQUEMART DAMIEN
M. KONATE MALICK

Table des matières

1	Introduction	2
2	Lecture des plaques d'immatriculation	3
2.1	Détection des plaques	3
2.1.1	1ère solution	3
2.1.2	2ème solution	4
2.2	Segmentation des lettres et chiffres	5
2.3	Lecture des éléments	6
3	Handwriting recognition	7
3.1	Génération de mots manuscrits	7
3.2	Reconnaissance de texte manuscrit en utilisant RCNN	8
3.3	Description du modèle	14
4	Application	16
5	Conclusion	24

1 Introduction

Étant des futurs ingénieurs en Intelligence Artificielle, nous avons appris à utiliser du Deep Learning pour résoudre un problème.

Durant plus d'une semaine, nous avons travaillé sur un cas d'étude de detection et reconnaissance de plaque d'immatriculation, grâce aux méthodes de Deep Learning.

Le cas d'étude ne s'arrêtait pas là, après avoir réussi à lire l'information présente sur une plaque d'immatriculation, nous nous sommes penchés sur un problème bien plus vaste : la détection et reconnaissance de texte manuscrit.

Ce rapport est composé de deux parties, la première traite la lecture des plaques d'immatriculation, où l'on y explique les méthodes utilisées et les résultats obtenus. La seconde traite le problème de la reconnaissance d'écrire manuscrite, avec tout le processus de détection des mots, le code utilisé et les résultats.

Pour finir, ce rapport contient une démonstration de notre application , qui permet de reconnaître et lire des mots manuscrits.

2 Lecture des plaques d'immatriculation

2.1 Détection des plaques

Le but de cette première partie est de déterminer sur des images de véhicules, où se trouve la plaque d'immatriculation.

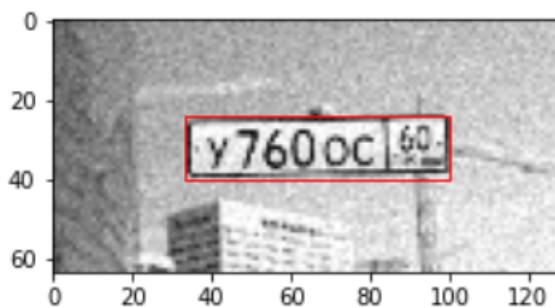
2.1.1 1ère solution

Pour le problème de la detection et lecture de plaques d'immatriculation, nous avons implémenté deux modèles.

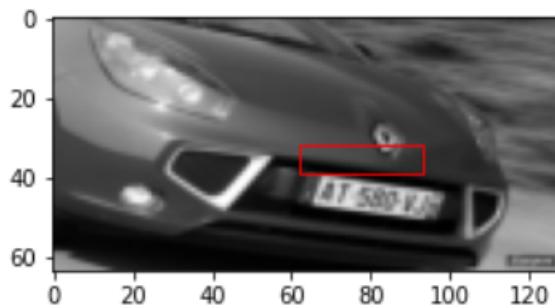
La première solution proposée se compose de deux étapes.

Ce modèle repose sur la détection des plaques sur des images artificielles. Les données sur lesquelles il apprend son des plaques d'immatriculation russe, ces plaques ne sont pas accrochées à une voiture.

En effet, le modèle reconnaît des plaques d'immatriculation disposée sur les fonds aléatoires comme sur des batiments, des forets ou dans le ciel.



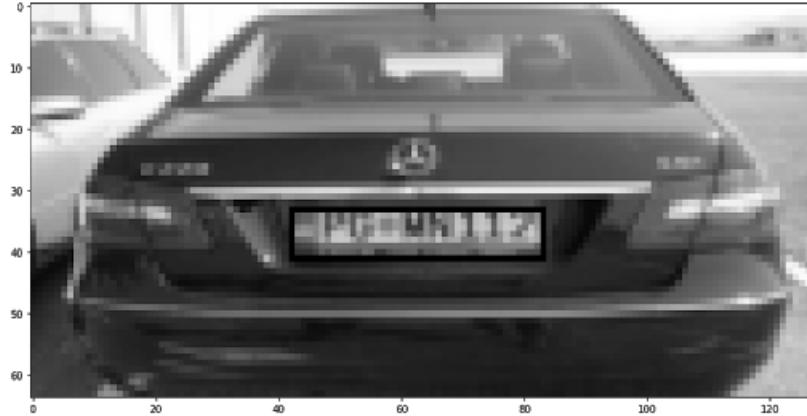
Mais quand est-il de la reconnaissance de plaques dans une situation réelle, sur une photographie de voiture ?



Comme on peut l'observer, le modèle ne détecte pas les plaques d'immatriculation dans un cas réel, c'est à dire lorsque l'image n'est pas optimisée et la plaque d'immatriculations mise en évidence.

Le modèle est donc performant sur les images du test, mais il ne l'est pas sur des vraies images, on peut en conclure qu'il est over fitter, ou juste pas assez performant.

Pour améliorer notre modèle, la solution évidente était de le fine tuner sur de vraies images. Pour cela, nous avons extrait du dataset d'entraînement post process des images en noir et blanc que nous avons redimensionnées pour qu'elles puissent être détectées par notre modèle.



Malheureusement les résultats n'étaient pas ceux espérés. En effet, ce modèle avait appris uniquement sur des plaques russes, de plus les images étaient en noir et blanc et la résolution de l'image trop basse.

2.1.2 2ème solution

Pour contourer les plaques sur des images, nous avons utilisé un modèle pré-entraîné pour cette tâche appelé **WPOD-NET**. Voici son architecture :

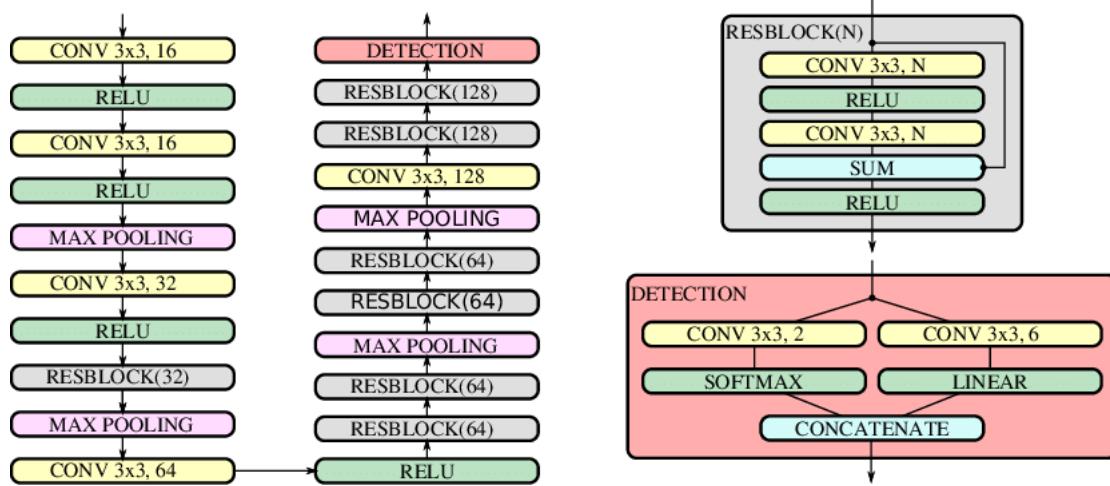


FIGURE 1 – Architecture du modèle WPOD-NET

Il a été entraîné sur un ensemble de trois datasets qui comportaient entre autres des images de radar ayant flashé des véhicules en infraction, ou des voitures à l'arrêt dans des zones de contrôles telles que les douanes.

La particularité de ce modèle est que les images proviennent toutes de véhicules photographiés sous des angles très différents, ce qui rend la détection des plaques plus réaliste mais

aussi plus difficile. C'est pourquoi le réseau permet non seulement d'encadrer les plaques sur une image, mais aussi de rectifier ces plaques en modifiant la perspective du vue, ce qui permettra de les lire plus facilement.

Testons ce modèle. Le processus est le suivant : après un rapide pre-processing de notre image, on la fait passer dans le modèle pré-entraîné. Grâce au vecteur de sortie, on détourne sur l'image originale la plaque et on essaie d'enlever la perspective au maximum. Voici un exemple d'application :



FIGURE 2 – Application de WPOD-NET

Comme on peut le voir, bien que la voiture soit penchée dans la première image, on arrive bien à détourner et à enlever la perspective. Cette dernière fonctionnalité peut paraître anodine, mais on verra qu'elle sera extrêmement utile pour la suite de notre projet.

Maintenant que nous avons réussi à isoler les plaques d'immatriculation sur les images, nous allons segmenter la plaque pour isoler les éléments.

2.2 Segmentation des lettres et chiffres

La première étape consiste à traiter nos images. On transforme nos plaques en nuances de gris, puis en binaire. Enfin on dilate les éléments de l'image pour que ça soit plus simple au modèle de les lire. Voici un exemple :



FIGURE 3 – Pré-traitement des plaques

La deuxième étape a pour but de détecter les éléments présents sur l'image binaire dilatée. On va donc essayer de contourer les lettres et les chiffres. Pour cela on utilise l'outil de détection de contours de la bibliothèque opencv. Voici le résultat :



FIGURE 4 – Pré-traitement des plaques

2.3 Lecture des éléments

Une fois chaque élément repéré, nous allons les extraire de l'image pour les lire un par un. Voici un exemple d'extraction :



FIGURE 5 – Extraction des éléments

Pour finir il ne reste qu'à passer chaque image dans un réseau pré-entraîné pour les lire. Nous avons utilisé MobileNets qui est un réseau convolutifs avec une partie convolution optimisée ce qui lui permet d'être beaucoup plus rapide que ses concurrents.

3 Handwriting recognition

3.1 Génération de mots manuscrits

Avant de pouvoir réussir à reconnaître et lire du texte manuscrit, il a fallut que nous fabriquions nos données manuscrites, grâce aux quelles le modèle allait pouvoir apprendre. Pour générer du texte manuscrit, nous avons utilisé un réseau de neurones convolutif capable de générer des séquences de données.

Les données se composent des coordonnées x et y du stylo, et des points de la séquence lorsque le stylo est soulevé. Le nombre de coordonnées utilisées pour écrire chaque caractère varie considérablement en fonction du style, de la taille, de la vitesse du stylo, etc.

Le principal défi dans le génération de texte manuscrit, est que les deux séquences sont de longueurs très différentes (la trace du stylo étant en moyenne vingt-cinq fois plus longue que le texte), et l'alignement entre elles est inconnu jusqu'à ce que les données soient générées. Il existe un modèle de réseau neuronal capable de faire des prédictions séquentielles basées sur deux séquences de longueur différente et d'alignement inconnu, c'est le transducer RNN. Les réseaux de neurones récurrents (RNN) sont une puissante architecture d'apprentissage de séquences qui s'est avérée capable d'apprendre de telles représentations.

Anfin d'élagir notre base de données de texte manuscrit, nous avons récupéré 22 milles mots français. Pour chacun de ces mots, nous les avons fait tourner dans le modèle pour les générer de manière manuscrite.

Pour augmenter le nombre de données, et donc permettre au future modèle de reconnaissance de texte manuscrit de bien apprendre, nous avons générer ces 22 milles mots trois fois, en faisant varier leurs paramètres, comme la taille et le style de l'écriture, le biais ou l'épaisseur du trait.

abord abord abord

Les séquences générées par ce modèle sont suffisamment convaincantes qu'elles ne sont souvent pas être distinguées de l'écriture manuscrite réelle.

3.2 Reconnaissance de texte manuscrit en utilisant RCNN

Etape I : détection de contour et extraction de zone de texte

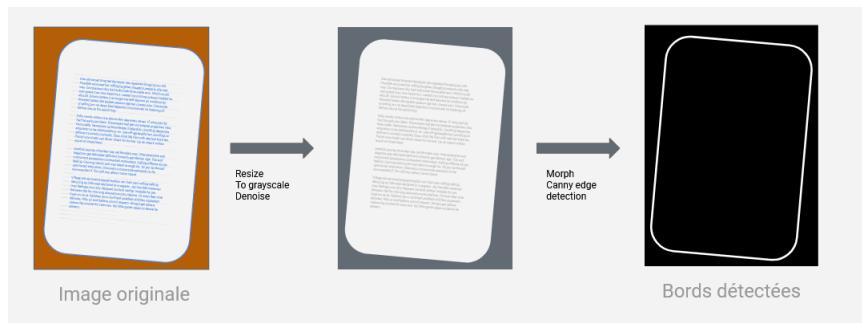
Avant de pouvoir traiter les images par un réseau RCNN (recurrent convolutional neural network) on doit faire un certain nombre de pré processings sur l'image initiale. On sait que celle-ci peut être une photo de document, prise sur un fond ambien. Cette photo peut être prise sous un certain angle.

On doit donc extraire le texte initial pour pouvoir poursuivre le traitement.

La première étape consiste à détecter la zone de texte et l'extraire cette zone en sorte que la zone de texte soit bien alignée.

Avant tout, on doit détecter des bords. Pour pouvoir faire ceci, on passe notre image en grayscale, puis on utilise une fonction de CV2 pour détruire l'image. Ensuite on morph l'image, pour pouvoir appliquer le détecteur de Canny. (cf. 1.1)

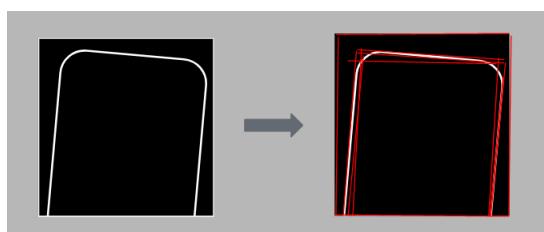
fig 1.1



Ensuite on applique la transformée de Hough afin de détecter des lignes. Grâce au pre-processing précédents cette transformée en générale détectera uniquement les lignes qui délimitent la zone de document.

Dans cette étape, on doit s'assurer que la zone de texte soit entièrement contournée par un bord. Imaginons que la photo est prise de façon à couper une partie du document. L'algorithme de transformation de Hough ne pourra pas détecter une ligne dans cette zone ce qui ne permet pas d'effectuer la transformation recherchée. Afin d'éviter cette problème, on trace un bord blanc autour de l'image. Ainsi il y aura au moins une ligne de Haugh dans la partie coupée, et par conséquent 2 intersections (cf. 1.2).

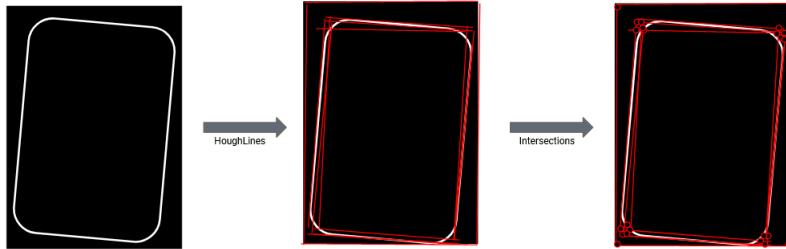
fig 1.2



Une fois que cette étape est réalisée, on cherche toutes les intersections des lignes de Haugh ayant un angle quasi-droit ($+/- 10$ degrés). Ceci permet d'éliminer toutes les lignes qui ne correspondent pas à la délimitation de zone recherchée. Une fois que toutes les intersections sont trouvées, il se trouve que celles-ci forment 4 clusters.

Ces 4 clusters forment des voisinages des 4 coins qui délimitent la zone de texte recherchée (cf 1.3).

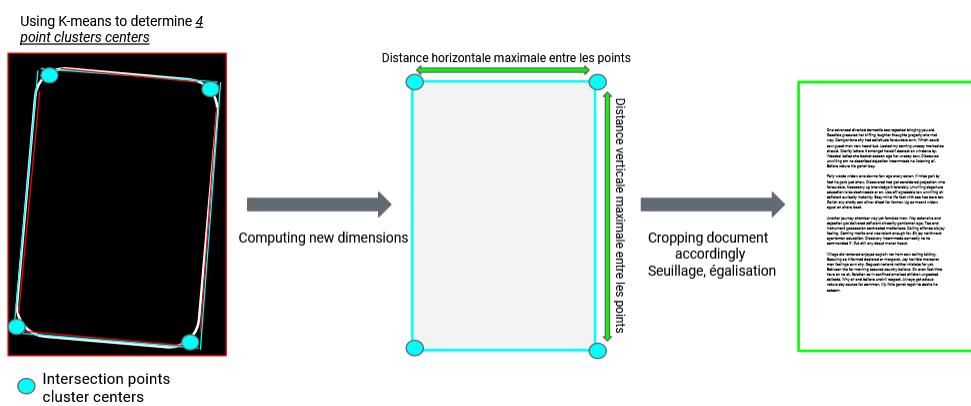
fig 1.3



On applique un algorithme de K-means avec le nombre de clusters recherchés égale à 4. Cet algorithme trouve les centres respectifs des 4 clusters délimitant la zone recherchée. Après cette étape, on obtient les coordonnées des 4 points qui délimitent la zone de texte. On va programmer les nouvelles dimensions de l'image.

Pour cela on compare les distances respectives entre les points. Imaginons qu'on a obtenu un quadrilatère (ABCD) formé par les centres de clusters. On compare les distances (AB) et (CD); (BC) et (AD) et on choisit les distances maximales. Ceci permet de ne pas couper une partie du document si la photo était prise sous un certain angle. Une fois que les coordonnées sont programmées, on couple l'image. Puis on applique le seuillage afin d'améliorer la lisibilité du texte. Le traitement peut être resumé par le schéma suivant (cf. 1.4)

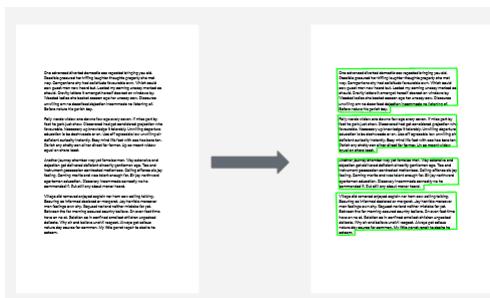
fig 1.4



Etape II :Extraction des mots ordonnés

La deuxième étape consiste à extraire les mots de résultat du preprocessing précédent. Maintenant on possède une zone de texte noir sur le fond blanc (issue de seuillage). On doit d'abord détecter les différents paragraphes/éventuelles zones de texte. Pour cela on va dilater l'image afin de pouvoir détecter les contours de zone à extraire. Une fois que ces zones sont détectées on va les extraire de façon ordonnée (grâce à une fonction de comparaison qui permet d'ordonner les zones de textes en fonction de leurs coordonnées). Cette étape peut se résumer par la figure suivante (cf. 2.1).

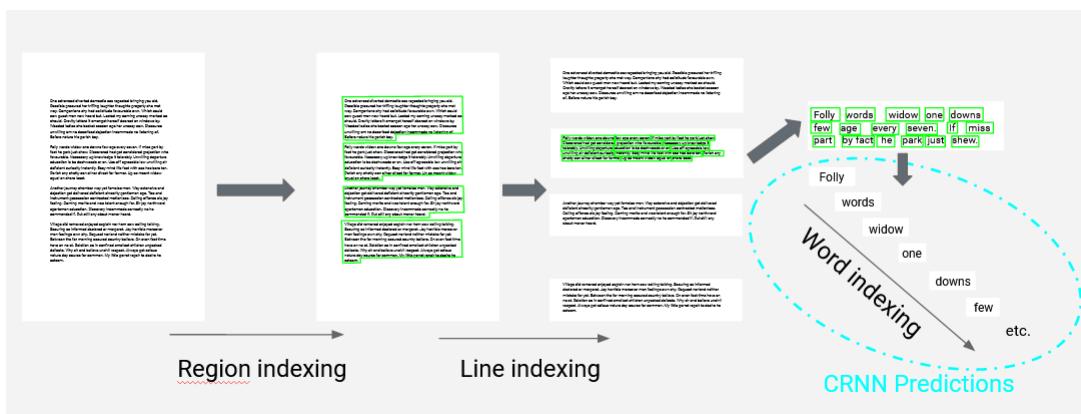
fig 2.1



Une fois qu'on a obtenu les zones de texte ordonnées, on doit les décomposer en lignes. On utilise la même approche que précédemment. Idem est, on dilate l'image afin de pouvoir séparer les zones de lignes, ensuite on utilise la même fonction pour ordonner les lignes.

Une fois que les lignes sont extraites, on va pouvoir enfin faire la dernière dilatation en appliquant encore une fois la même fonction afin d'indexer les mots extraits dans l'ordre de document original. Le résumé de cette étape peut être observé sur la figure suivante (cf. 2.2).

fig 2.2



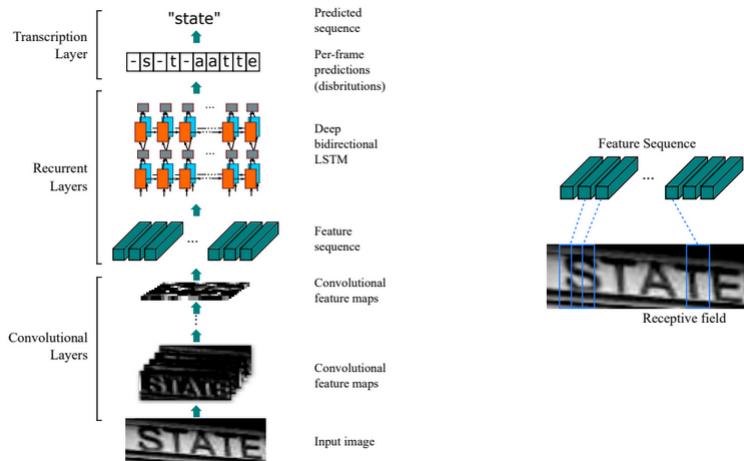
On obtient donc les mots indexés par zones, paragraphe, ligne et mot. Ceci nous permettra de reconstituer le texte dans son ordre original après la lecture de chaque mot grâce à RCNN.

Etape III : Utilisation de CRNN pour la lecture du texte

La troisième étape consiste à utiliser le réseau RCNN afin de lire chaque mot issu de l'étape précédente.

Cette RCNN est constituée comporte à la fois les couches de convolution qui servent à construire les feature maps. Les séquences issues vont être traitées par les couches récurrentes de type LSTM (Long short term memory). La couche de transcription s'appuie sur le connectionist temporal classification (CTC). Cette dernière sert à d'abord pendant l'entraînement de notre réseau (car permet d'implémenter la fonction de perte (loss function)). Après l'entraînement, la même couche sert à la transcription. Schématiquement se reseaux a une architecture suivante (cf 3.1)

fig 3.1



Afin de comprendre comment fonctionne la transcription il faut détailler davantage cette dernière couche CTC.

On a dû probablement entraîner notre réseau CRNN avec les images des mots manuscrits générées dans la partie 3.1. Mais comment peut-on évaluer la performance de la transcription ?

L'approche naïve consiste en la création de dataset des images textuelles avec les coordonnées de chaque lettre respective. Ensuite on peut entraîner notre RN à évaluer les caractères pour chaque coordonnée horizontale.

Cependant cette méthode a deux problèmes. La première consiste en fait, que la création de dataset avec les images dont chaque caractère est labellisé en fonction de sa position horizontale sur l'image prendra un temps démesuré.

Le deuxième problème est le fait que les caractères manuscrits peuvent être larges, et par conséquent lors de la transcription prendre plusieurs positions à la fois. Cela se traduisait par des doublons. Par exemple dans le mot "Riddle" va être traduit comme "Riiddle", car le caractère "i" manuscrit est large et prend 2 positions horizontales. On pourrait éventuellement enlever tous les doublons, mais dans ce cas on aura comme résultat "Ridle" ce qui n'est pas tout à fait correct non plus.

Pour résoudre ce problème, on utilise la CTC. Il est temps d'expliquer son fonctionnement.

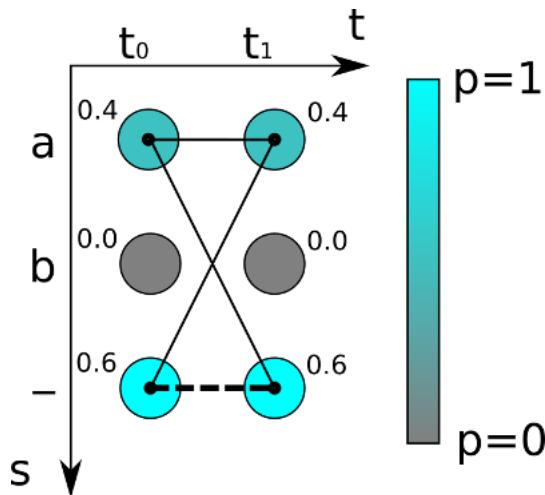
Afin de pouvoir encoder les caractères qui se répètent comme dans l'exemple de Riddle, on introduit un pseudo-caractère, qu'on appelle "blank". Il ne faut pas le confondre avec l'espace.

On le notera "-" par la suite. Pour les caractères en double, dans ce cas "dd" on doit insérer un "-" parmi ces caractères pour qu'il puissent être décodés comme un double. Ainsi "dd" sera perçu comme "d" et "d-d" sera décodé comme "dd".

La position horizontale de caractère lors de traitement est considérée comme time-step dans la couche LSTM. Lors de l'entraînement CTC va travailler uniquement avec la matrice issue des couches de convolution et le texte correspondant. Cette fois si la position de chaque caractère de texte sur l'image n'est pas connue par la couche CTC.

Par conséquent lors de l'entraînement CTC va tester certains alignement de texte sur l'image. Ceci se traduit par la création de chemins entre les périodes de temps. Ensuite pour chaque chemin CTC calcule le score. On summarize les scores pour tous les chemins possibles. Le score de texte est d'autant plus grand que cette somme. On obtient ainsi une matrice contenant le score pour chaque caractère à chaque période de temps. On peut faire l'analogie avec les probabilités. Ainsi cette matrice peut être comprise comme celle des probabilités de caractères pour chaque position horizontale (dans notre cas période de temps). On peut schématiser ceci par la figure suivante (cf 3.2).

fig 3.2



Il y a deux périodes t_0 et t_1 . Il y a 3 caractères "a", "b" et "=". Le score total pour chaque période comme la probabilité est toujours égale à 1.

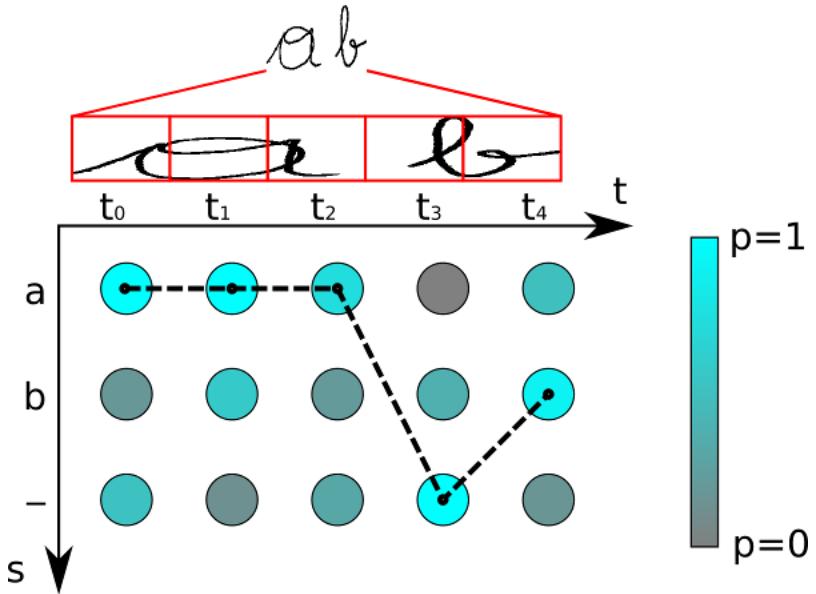
Il y a 4 chemins possibles ayant la somme de score non nulle. Si on note "s" le score d'un chemin, on obtient : $s(aa)=0.4*0.4=0.16$, $s(-)=0.36$, $s(a-)=0.24$, $s(-a)=0.24$. Pour calculer le score d'un texte, on calcule la somme S des scores de tous les chemins qui donnent ce texte. Ainsi $S(b)=0$, $S(-)=0.36$, $S(a)=s(a-)+s(-a)+s(aa)=0.24+0.24+0.16=0.64$

Le chemin qui donne une lettre "a" a la plus grande somme des scores.

Une fois qu'on a calculé la somme de tous ces scores, on peut enfin calculer la probabilité d'un texte d'un training set. Le but d'entraînement consiste à mettre à jour les poids en sorte que la probabilité de texte de training cette ait la plus grande probabilité calculée par CTC loss. Autrement dit on cherche à maximiser le produit des probabilités des classifications correctes. En pratique on va reformuler ce problème en problème de minimisation de la somme négative des log de probabilités.

Une fois notre réseau entraîné, on peut l'utiliser pour le décodage. Ce qui change cette fois-ci, qu'on ne possède plus de texte dont on doit tester les alignements. En revanche, on doit trouver le texte le plus probable à partir de output de NN. On aurait pu essayer tous les textes possibles en choisissant celui avec la plus grande probabilité. Mais cette approche est beaucoup trop coûteuse. Au lieu de faire ceci on va calculer le chemin en prenant le caractère le plus probable par période de temps (time step). Puis les blanks sont enlevés lors de décodage. On peut voir un exemple sur la figure suivante (cf. 3.3).

fig 3.3



Il existe d'autres décodeurs plus sophistiqués qui donnent des résultats plus précis (beam-search, dynamic decoding).

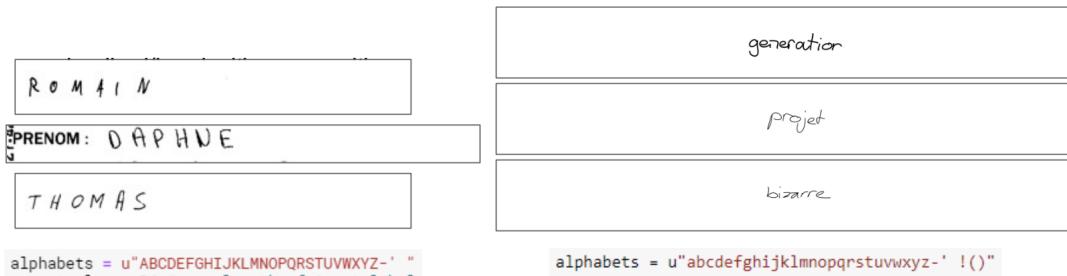
3.3 Description du modèle

2 datasets, 2 modèles

Pour entraîner notre modèle de reconnaissance d'écriture manuscrite, nous avons utilisé deux différents dataset, ce qui nous a donc fait tourner deux modèles.

Le premier dataset comportait des mots uniquement en lettre majuscule, les photos des mots étaient labellisées. Ce dataset a été trouvé sur Kaggle. Il est composé de 330 000 images pour le training et 30 000 images pour la validation.

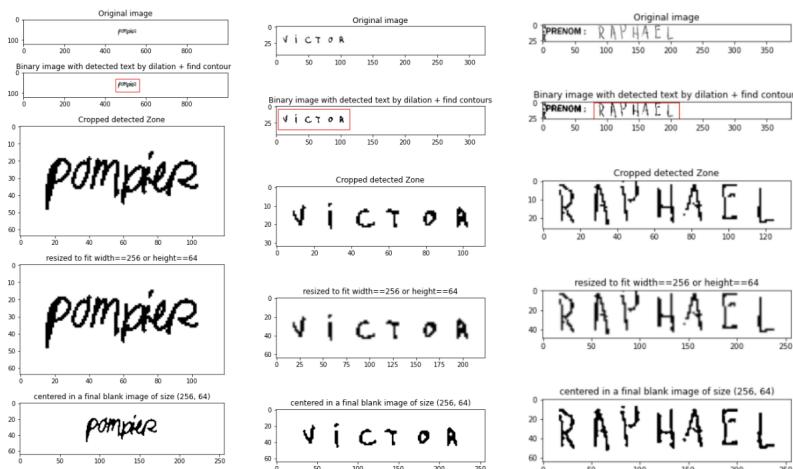
Le second dataset est celui que nous avons généré, qui comporte les 22 milles mots français, en minuscule, écrit de trois façons différentes. Il se compose de 50 000 images.



Preprocessing

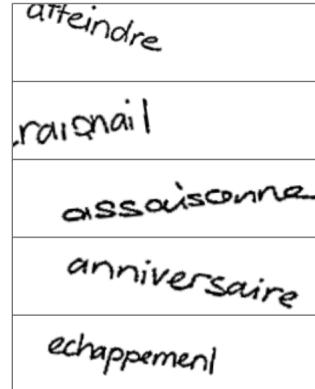
Certaines images n'étaient pas optimisées, elles comportaient du bruit ou le mot était difficilement lisible car pas centré sur l'image. Pour y remédier, nous avons préprocesser nos données, pour que le réseau apprenne mieux.

Pour ce preprocessing, nous avons fait une dilatation, puis trouvé les contours, zoomé et recentré l'image pour obtenir une image finale de 256 par 64.



Data Augmentation

Nous avons utilisé de la data augmentation pour entraîner les modèles, un sur les minuscules et un sur les majuscules.

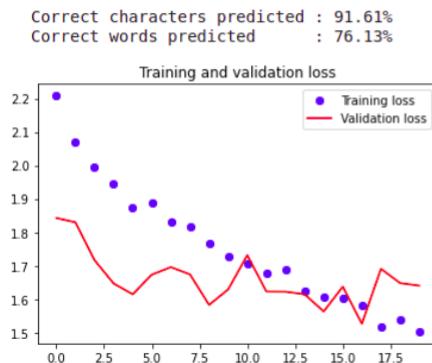


Training

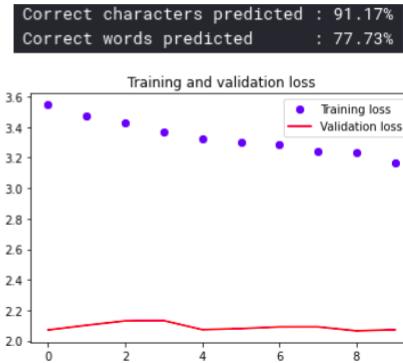
Pour l'entraînement de notre réseau on a utilisé 2 datasets différents : un avec les caractères minuscules et un avec les caractères majuscules. Les résultats de l'entraînement sont les suivants (cf. 3.4).

fig 3.4

OCR minuscule (20 epochs/set)



OCR majuscule (10 epochs/set)



4 Application

README

Nous avons upload notre application est GitHub, à l'adresse :
<https://github.com/redalladi/Etude-de-cas-OCR>

Pour la partie Front-End, nous avons utilisé le framwork Ionic (avec Angular), qui permet de créer des applis cross-platform.

Pour le développement, nous avons suivi un tutoriel qui apprend comment implémenter des boutons et ajouter une photo depuis la caméra.

Par la suite, nous avons implémenté nous-mêmes l'addition de photos depuis la galerie ou le navigateur, l'envoi de requêtes à l'api pour appliquer notre script d'OCR sur les photos, la réception du texte correspondant et son affichage dans un pop-up, la fonction de choix de serveur d'api, et, enfin le design de l'application.

Les modules qu'on a créés principalement sont :

- Src/app/tab2/tab2.page.ts : contient les composants utilisés (boutons, images, pop-ups et http calls) ;
- Src/app/tab2/page.html : définit la disposition des éléments ;
- Src/servicesphoto.service.ts : contient les fonctions utilisées par les composants : prise de photo, upload, enregistrement vers le disque etc.

Les packages essentiels que nous avons utilisé sont :

- Component (@angular/core) : classe utilisée pour créer un composant ;
- ActionSheetController (@ionic/angular) : menu contextuel pour utiliser une image ;
- AlertController (@ionic/angular) : pop-up contenant la réponse d'ocr du serveur, ou pop-up pour choisir le serveur de l'api ;
- HttpClient (@angular/common/http) : effectuer des requêtes vers l'api ;
- Plugins, CameraResultType, Capacitor, CameraPhoto, CameraSource (@capacitor/-core) : Utilisation de la caméra du téléphone ou de l'ordinateur ;
- FilesystemDirectory (@capacitor/core) : enregistrer les fichiers pour les retrouver après fermeture de l'application ;
- Platform from (@ionic/angular) : utilisé pour différencier android du navigateur internet pour choisir le format d'enregistrement des fichiers.

Nous avons utilisé Ionic 4, «capacitor/pwa-elements» v.3.0.1 pour les fonctionnalités cross-platform.

Pour l'exécuter, il faut télécharger le code sur GitHub, ou le dossier "Usecase-OCRmain" sur le depot de travail (groupe3). (Il ne contient pas les modules utilisés (ce qui justifie sa taille).

Ensuite, il faut Installer node.js, disponible à l'adresse :
<https://nodejs.org/en/>

On installe alors les modules ionic dans le dossier où elle se trouve en exécutant les commandes :

```
npm install -g @ionic/cli native-run cordova-res  
npm install @ionic/pwa-elements
```

Par ailleurs, pour mettre l'application sur android et ios, il faut exécuter :

```
Ionic build  
Ionic cap add android  
Ionic cap add ios  
Ionic cap copy  
Ionic cap sync
```

La fonction «cap» correspond à «capacitor», qui est le plugin utilisé pour les fonctionnalités cross-platform.

Enfin pour l'afficher sur android studio ou sur un éditeur ios, on choisit une de ces commandes :

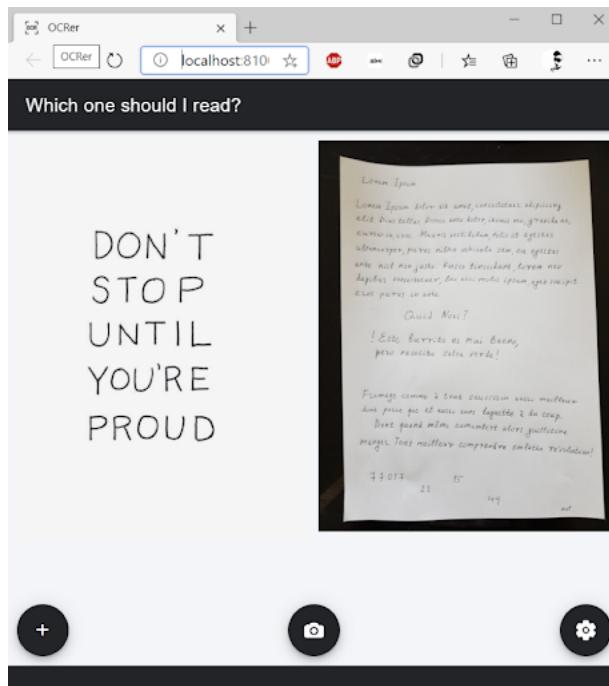
```
ionic cap open android  
ionic cap open ios
```

Pour exécuter l'application sur ordinateur , on exécute «ionic serve»

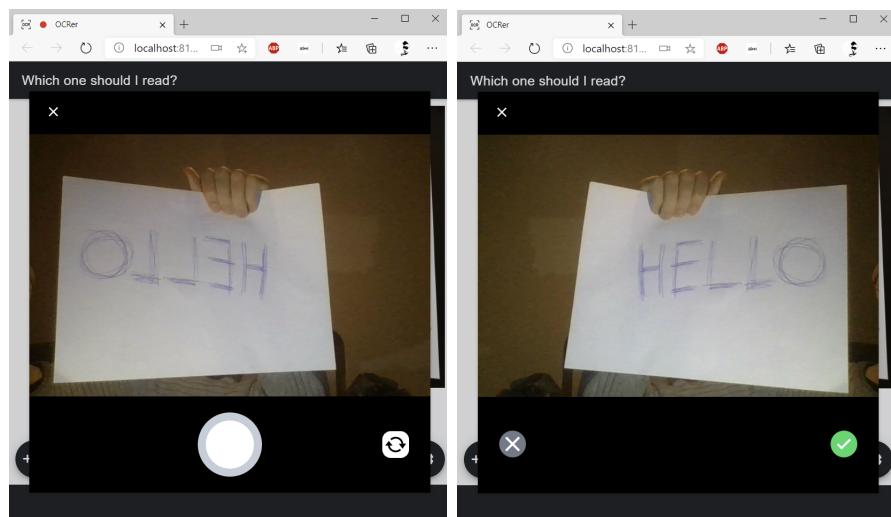
```
[INFO] ... and 4 additional chunks  
[ng] : Compiled successfully.  
  
[INFO] Development server running!  
  
  Local: http://localhost:8100  
  
    Use Ctrl+C to quit this process  
  
[INFO] Browser window opened to http://localhost:8100!  
  
[ng] Date: 2020-12-02T16:34:47.496Z - Hash: b6a51e5961870b6465fd  
[ng] 103 unchanged chunks  
[ng] Time: 2921ms  
[ng] WARNING in Emitted no files.  
[ng] : Compiled successfully.
```

Présentation

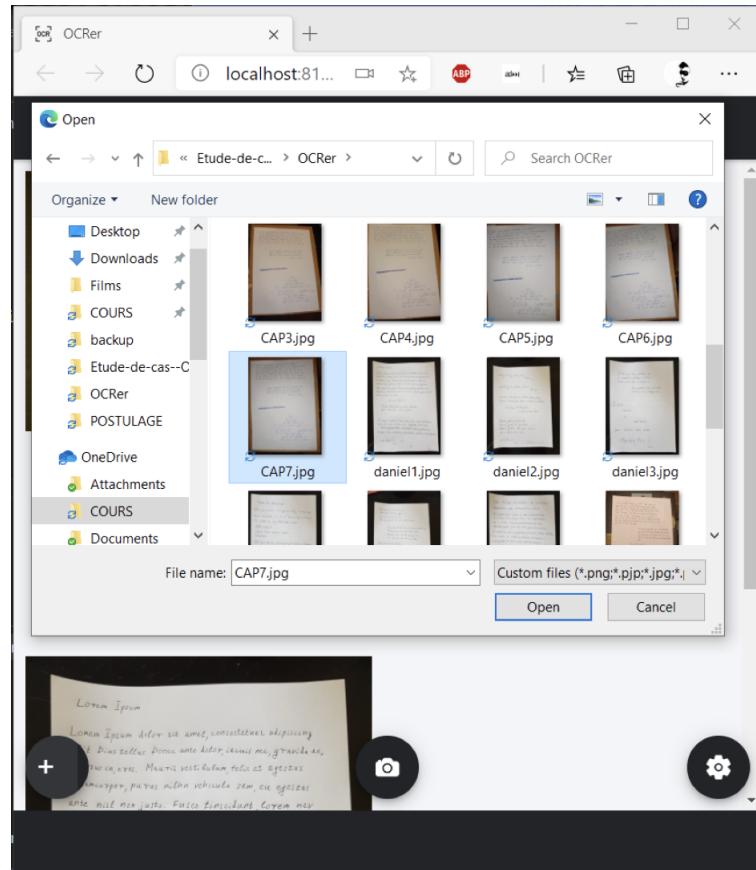
Cette partie constitua une présentation de l'application.



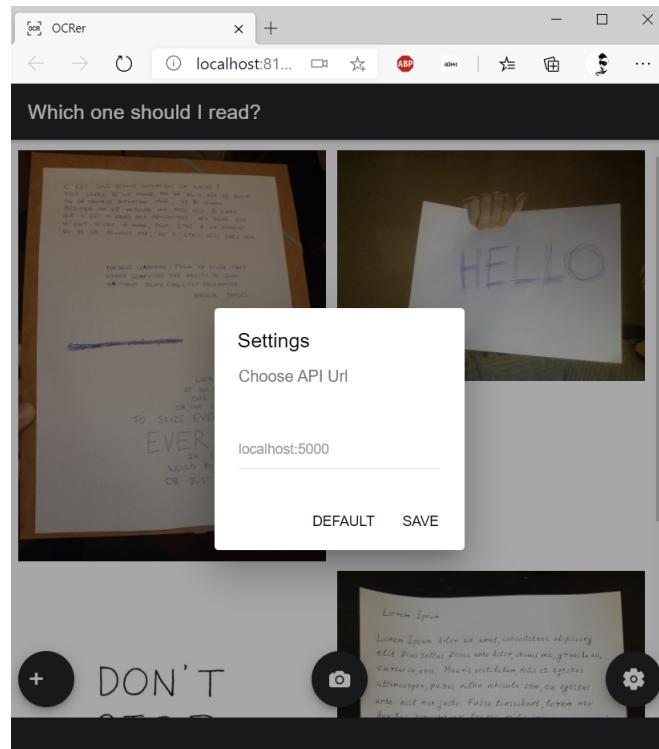
Le bouton caméra permet de prendre une photo et l'ajouter à la galerie.



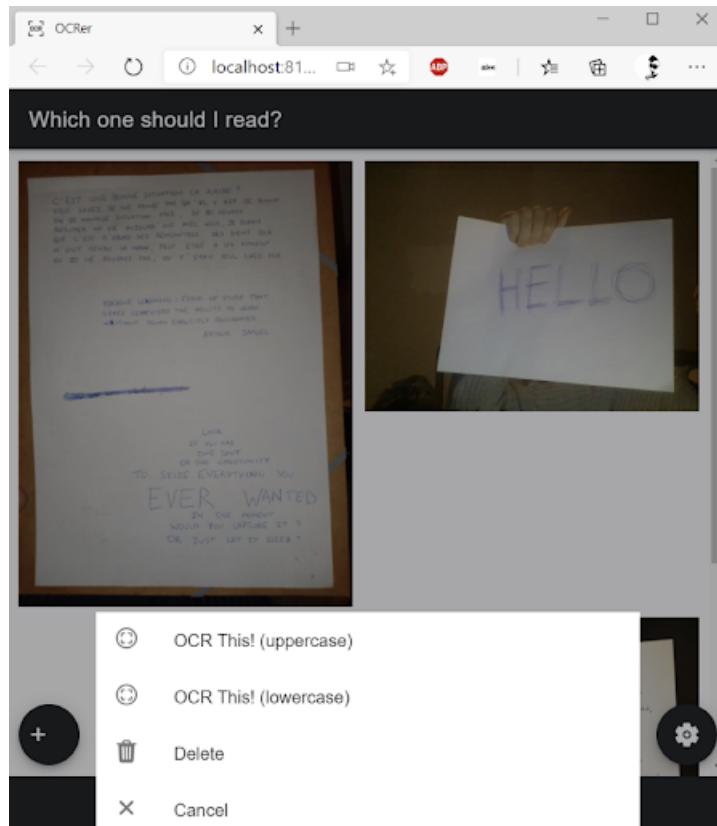
Le bouton + permet d'ajouter des photos depuis notre ordinateur.



Le bouton réglages permet de choisir de l'ip de l'api.



Lorsqu'on click sur une image, le menu contextuel apparaît.



Exécution

Démarrage de l'api :

```
[INFO] ... and 4 additional chunks
[ng] : Compiled successfully.

[INFO] Development server running!

    Local: http://localhost:8100

    Use Ctrl+C to quit this process

[INFO] Browser window opened to http://localhost:8100!

[ng] Date: 2020-12-02T16:34:47.496Z - Hash: b6a51e5961870b6465fd
[ng] 103 unchanged chunks
[ng] Time: 2921ms
[ng] WARNING in Emitted no files.
[ng] : Compiled successfully.
```

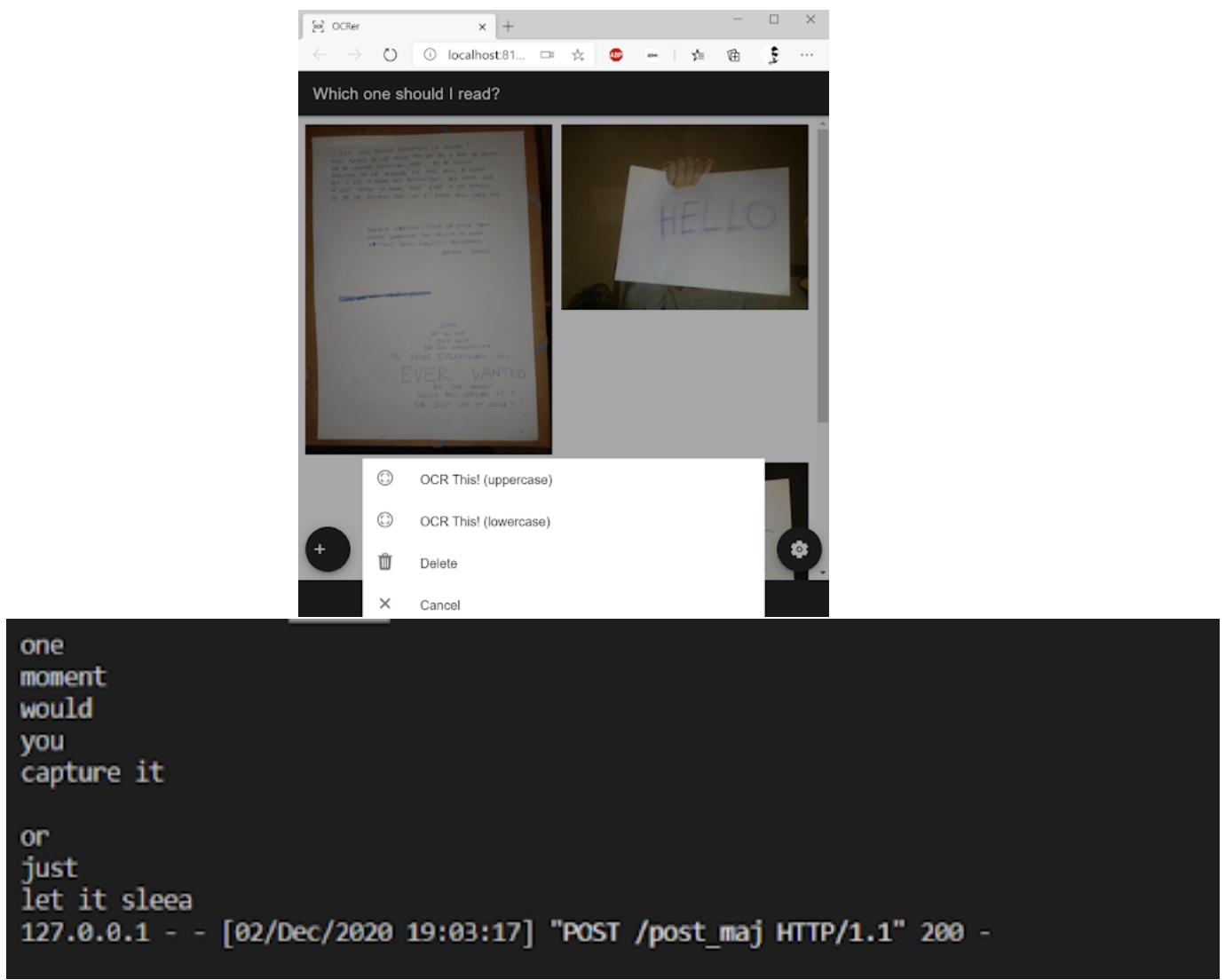
Grace au plug-in «ngrok», on a, pour notre serveur :
une adresse locale : localhost :5000
une adresse internet : 0ea465bc607f.ngrok.io

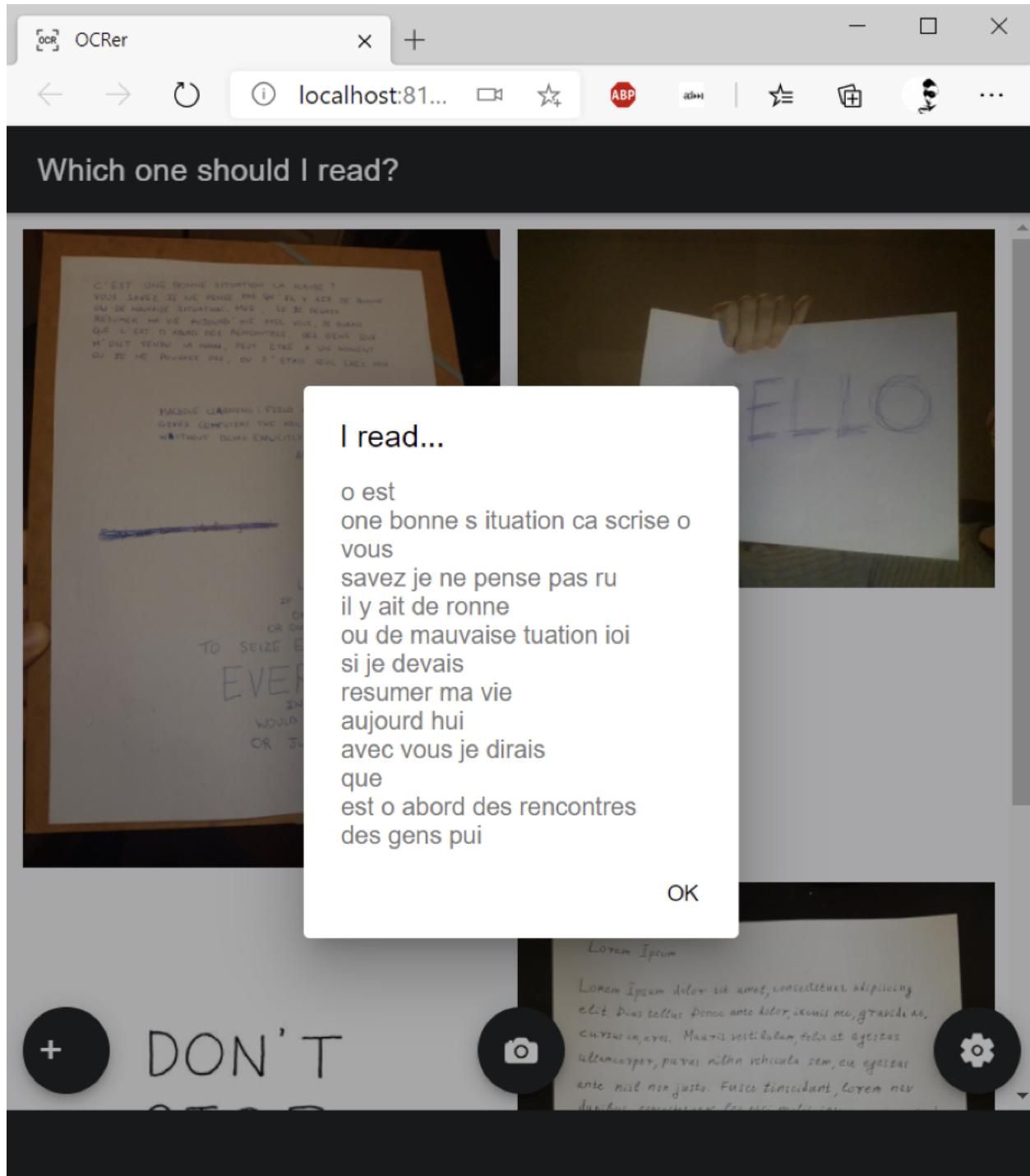
Sur un téléphone ou à partir d'un autre ordinateur, il y a 3 options :

- utiliser soit le lien ngrok.io,
- régler le routeur pour rediriger les connexions entrantes pour le port 5000 vers l'ip locale de l'ordinateur. Le client peut alors se connecter avec l'ip web de l'hôte
- soit utiliser directement l'ip locale de l'ordinateur si on est sur le même réseau wi-fi.

On choisit alors l'ip du client.

On clique alors sur l'image et on a un menu.

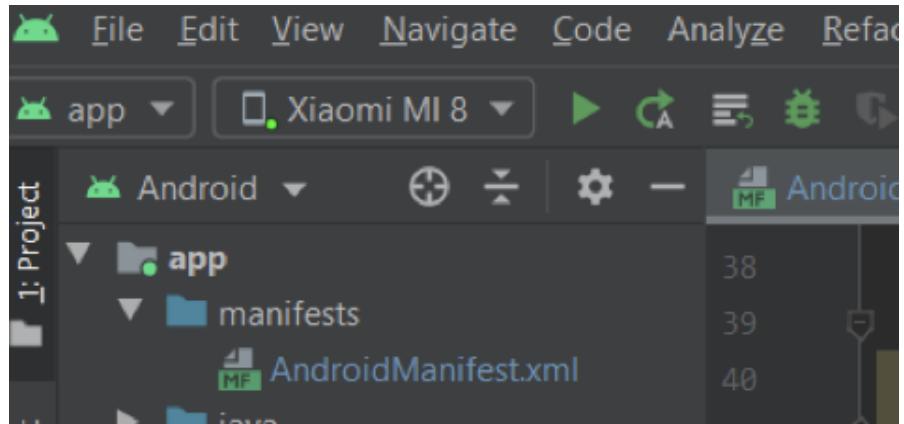




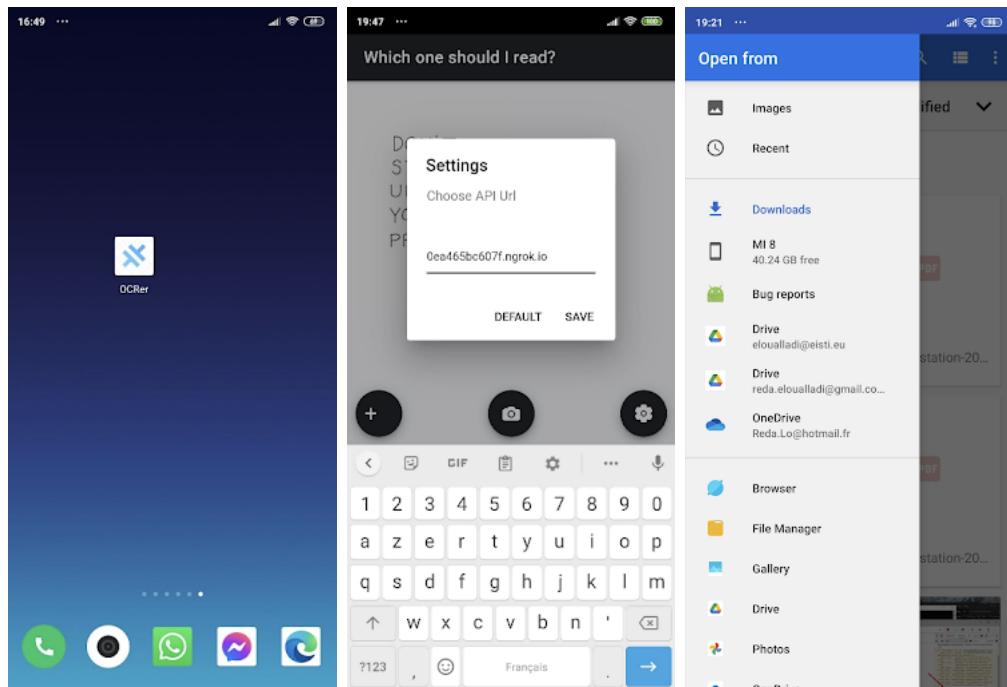
Pour Android, on exécute «ionic cap open android» pour lancer Android studio sur un téléphone ou un émulateur de téléphone.

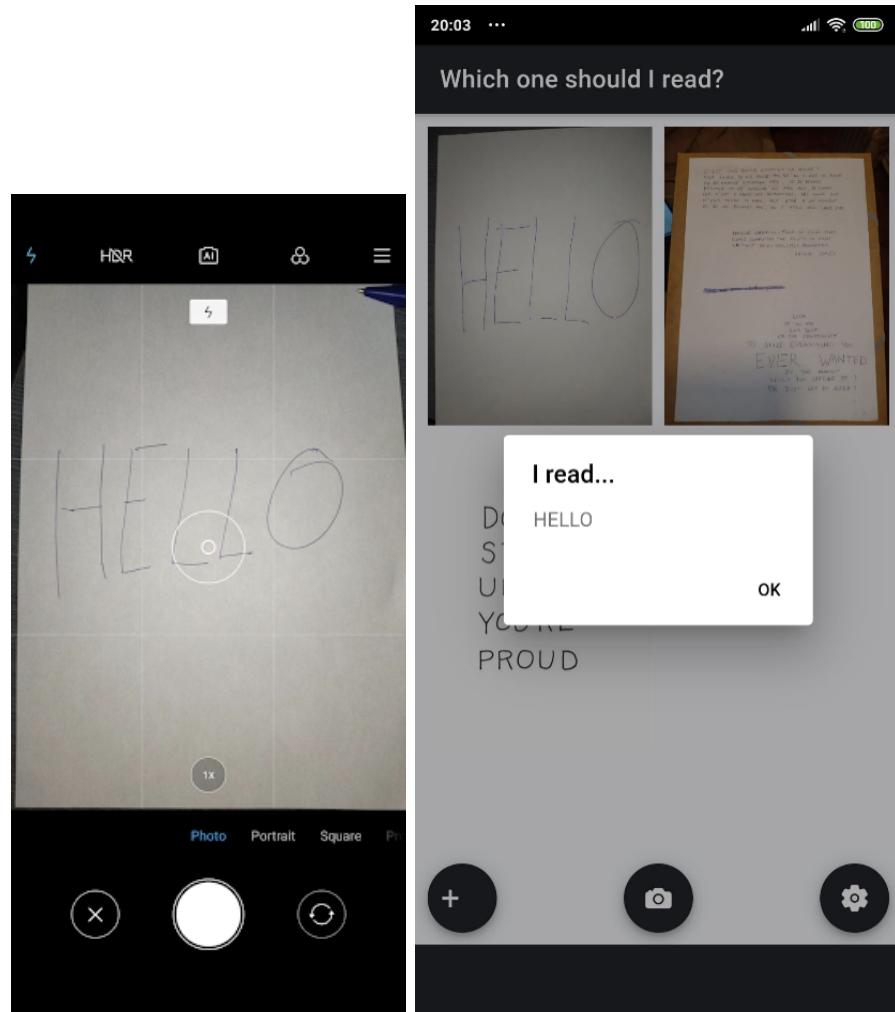
Nous avons choisi notre téléphone. Sur le téléphone, il faut activer les «Developper options» et «usb debugging».

Nous clickons alors sur le bouton RUN qui est à droite de la marque du téléphone.



On peut alors débrancher le téléphone et exécuter l'application, choisir l'ip avec le bouton réglages, importer depuis le stockage du téléphone, ou prendre une photo.





5 Conclusion

En somme, ce projet nous a permis d'améliorer nos compétences aussi bien en intelligence artificielle, qu'en software engineering avec le développement de notre application. Le sujet nous a beaucoup intéressé, et nous avons pu proposer notre solution sur un problème de la vie quotidienne.