



SORBONNE UNIVERSITÉ
MASTER ANDROIDE

Titre du stage

Stage de Master 1

Réalisé par :

Antony LECLERC

Encadré par :
Thibaut LUST, LIP6, Sorbonne Université

Référent :
Aurélie BEYNIER, LIP6, Sorbonne Université

13 septembre 2024

Table des matières

1	Introduction	1
2	État des lieux post projet (pendant le semestre)	2
3	Stage	6
4	Images non redimensionnées	8
4.1	Niveaux de gris	8
4.1.1	Courbes d'apprentissage	8
4.1.2	Tests visuels	9
4.2	Méthode Otsu	9
4.2.1	Courbes d'apprentissage	9
4.2.2	Tests visuels	10
4.3	Prétraitement	11
4.4	Courbes d'apprentissage	11
4.5	Tests visuels	12
5	Images redimensionnées - Rognage	13
5.1	Détection de contours	13
5.2	Extraction des numéros	15
5.3	Création du réseau de neurones et entraînement	18
5.4	Résultats	20
5.4.1	Niveaux de gris	20
5.4.2	Preprocessing	22
6	Test sur des images non utilisées à l'entraînement	23
6.1	Extraction et prédictions de zones	23
6.2	Agrégation des probabilités	24
7	Conclusion et Ouverture	26
8	Références	27

Chapitre 1

Introduction

Dans le domaine de la reconnaissance d’images, l’avancée technologique a permis de franchir de nombreux obstacles, notamment dans la reconnaissance optique de caractères (OCR). Cependant, la majorité des solutions actuelles se concentrent sur la reconnaissance de caractères sur surface planes et donc ne sont pas très adaptées à la reconnaissance des nombres sur les boules de LOTO, qui sont affichés sur une surface sphérique.



FIGURE 1.1 – Une boule de LOTO 9.



FIGURE 1.2 – Une boule de LOTO 6.

Notre base de données est constituée de 10429 photos de boules de LOTO contenant des numéros de 1 à 50, et de dimensions (240×320) . Elle nous a été fournie par notre encadrant, images qu’il a obtenu par le fabricant des machines de LOTO **Ryo-Catteau**, entreprise française basée dans le nord de la France à **Wattrelos**.

Le but du projet est donc de développer une approche basée sur les dernières avancées des réseaux de neurones convolutionnels (CNN [1]) et de l’apprentissage profond afin d’atteindre des prédictions proches des 100% pour la reconnaissance des nombres [2] sur les boules de loto, notamment les boules 6 et 9, dont le signe distinctif est le point sous le 6, ou la petite barre sous le 9.

Il est à noter que la reconnaissance automatique des nombres sur les boules se fait actuellement grâce à des puces RFID incluses dans les boules, mais cette solution présente le désavantage de coûts de fabrication et de maintenance assez élevés. Il serait donc intéressant de développer une méthode basée sur l’intelligence artificielle.

Par le biais de ce projet, nous espérons approfondir nos connaissances dans le domaine de l’apprentissage des réseaux de neurones et relever le défi d’apporter une solution améliorante par rapport à celles déjà existantes.

Lien vers le dépôt git du projet : <http://github.com/AnthonyLeclerc/ProjetLOTO.git>

Chapitre 2

État des lieux post projet (pendant le semestre)

Nous avions, lors de notre projet de M1 réalisé au 2eme semestre 2023, réussi à obtenir des résultats satisfaisants sur des images que nous avions rognées manuellement. Une fois rognées, l'ensemble des données avait été augmenté grâce à différentes transformations, disponibles dans la librairie Tensorflow.

Voici ci-dessous des exemples de résultats obtenus après apprentissage d'un réseau de neurones :

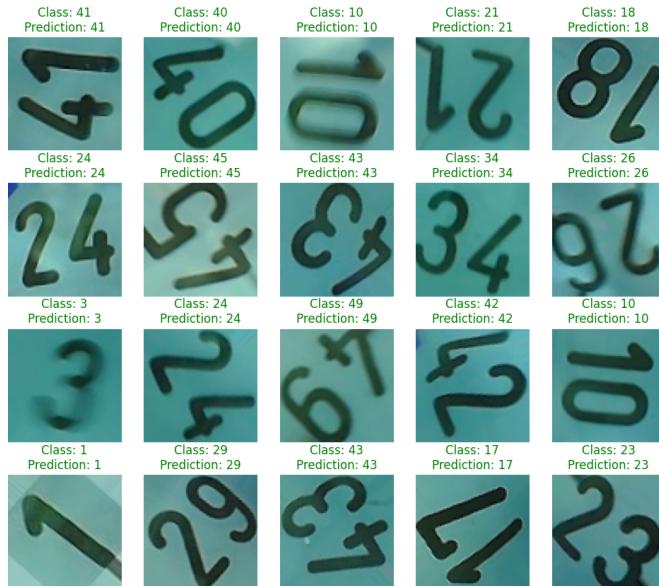
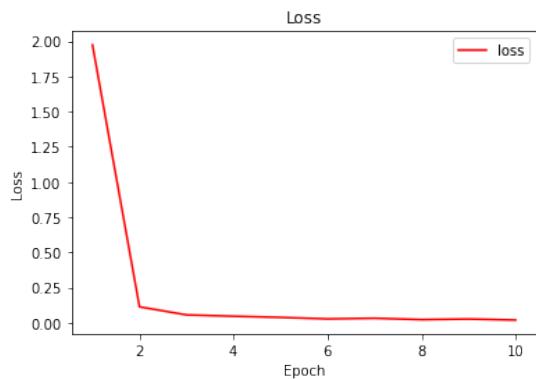
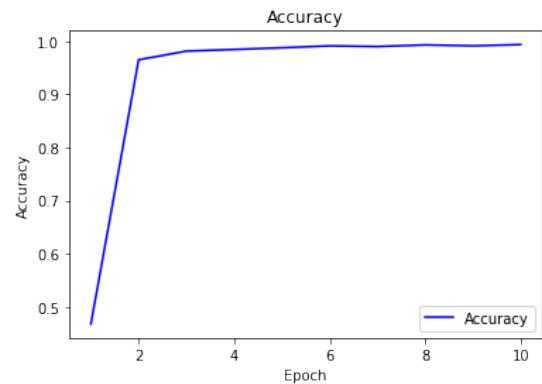


FIGURE 2.1 – Un exemple d'images artificiellement créées



(a) Loss



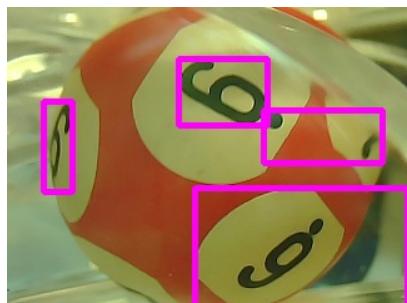
(b) Accuracy

Un test visuel de 20 images bien prédites (test non exhaustif), ainsi que les courbes d'apprentissage de *loss* et d'*accuracy*.

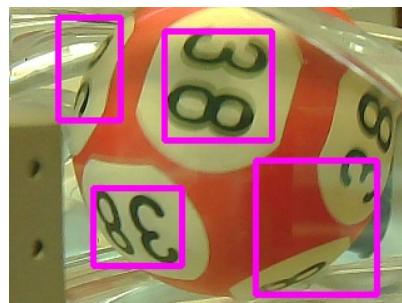
Différentes possibilités s'offraient alors à nous :

- Réussir à faire de la prédiction sur les images brutes, avec de plus grosses puissances de calcul pour l'apprentissage.
- Trouver un moyen de rogner nos images automatiquement et correctement sur les numéros, et faire notre prédiction sur ces images rognées.

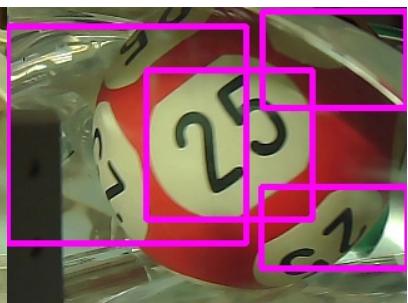
En utilisant certaines techniques d'extraction (expliquées en détails dans le chapitre 5 sur des images de meilleures qualité) voici quelques exemples de zones détectées.



(a) Un 6



(b) Un 38



(c) Un 25

En appliquant l'extraction et l'entraînement expliqués dans le chapitre 5, et sur des images passées en niveaux de gris, on obtient les courbes d'apprentissage suivantes :

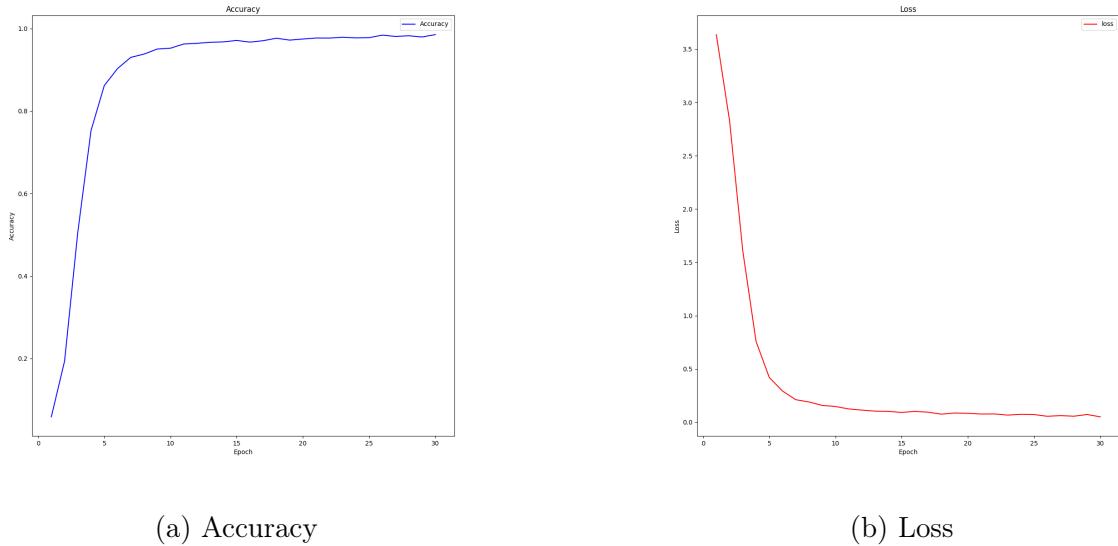


FIGURE 2.4 – Courbes d'apprentissage sur les images en niveaux de gris

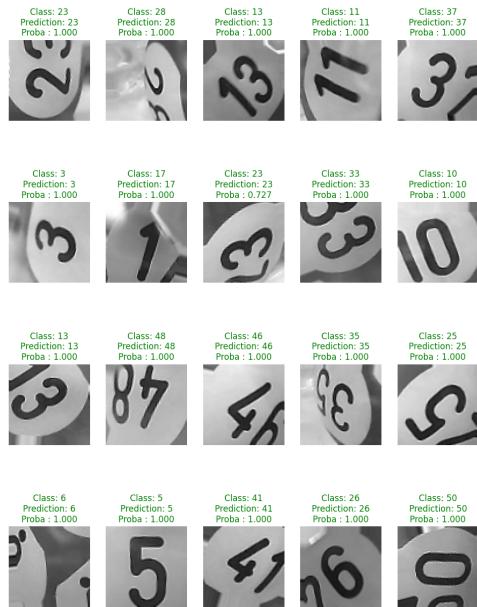
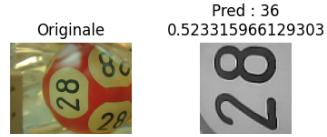
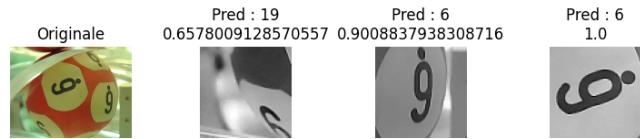


FIGURE 2.5 – Test visuel

Des résultats qui semblent être très bons, mais cela s'explique par le fait que les images étant de mauvaises qualités, nous n'avons pu extraire que 479 régions sur les 100 premières images de chaque classe pour les 50 classes. Nous avons par la suite artificiellement augmenté ce jeu de 479 images avec l'objet **ImageDataGenerator** de la librairie tensorflow qui permet de générer des images à partir d'images que nous possédons déjà et en y appliquant des modifications. Et malgré les modifications, les images se ressemblent énormément. En faisant par la suite quelques prédictions sur des images qui n'ont pas été prises pour l'entraînement, on obtient les résultats suivants :



(a) Prediction anciennes images



(b) Prediction anciennes images

Les mauvais résultats obtenus sur ces images peuvent s'expliquer par la qualité des images qui sont parfois très floues, avec une luminosité variable, les contours des numéros sont rarement détectés ce qui rend compliqué l'extraction, et par la suite l'apprentissage.

Chapitre 3

Stage

Nous avons pu obtenir auprès de l'entreprise *Ryo Catteau* un nouveau jeu de données avec des images de meilleure qualité.

Le cluster du LIP6 nous offre une capacité de calcul conséquente, nous avons ainsi pu tester à nouveau de faire apprendre à un réseau de neurones, les images non redimensionnées (images de dimension 960×1280). Et ce avec différents pré-traitements pour voir si l'un peut être plus efficace qu'un autre.

- Images en niveaux de gris
- Seuillage avec la méthode "Otsu", sépare l'avant de l'arrière plan d'une image
- Prétraitement utilisé pendant le projet lors du semestre

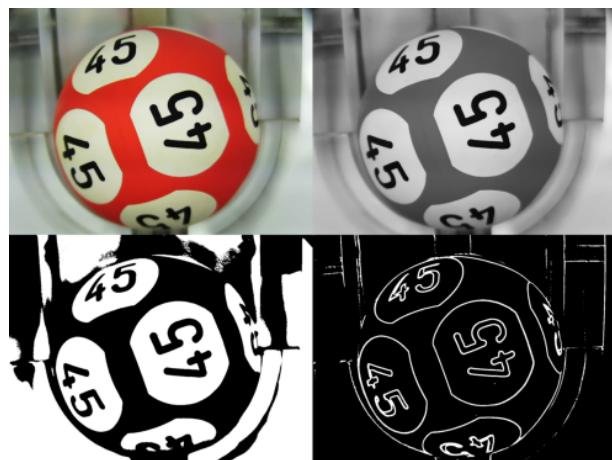


FIGURE 3.1 – Exemple d'image avec les différentes modifications

A noter cependant que nous avons essayé de faire de la prédiction sur les images brutes en couleurs, mais la mémoire occupée par les images en couleur étant trop importante (près de 3.6 Mo par image), il nous était impossible de faire un entraînement en important toutes les images, ce qui aurait pris près de 160 Go de mémoire.

Le seuillage Otsu étant une technique de seuillage sensée binariser une image en deux classes (premier plan et arrière plan), on aimerait avoir comme premier plan les numéros uniquement mais cela n'est pas parfait. Cependant les numéros étant relativement bien discriminés, nous avons appliqué ce seuillage pour voir si il est possible d'obtenir de bons résultats.

Aussi, afin de faciliter l'importation des images, il est important de suivre une convention de nommage de nos fichiers que nous avons choisi.

Le nom du dossier principal n'a que peu d'importance (je vais ici l'appeler LOTO pour faciliter les explications).

Dans le dossier LOTO, il doit y'avoir 50 sous dossiers, tous nommés **Boule** X , avec X allant de 1 à 50, soit un dossier par classe de boule.

Dans ces sous dossiers **Boule** X , on y met nos images, qui auront pour nom $X_num_Y.jpg$, X étant à nouveau la classe de la boule, afin d'associer plus facilement une image à son label, et Y son identifiant dans la classe. Ici, Y sert dans un premier temps évidemment à avoir des fichiers avec des noms différents, et servira par la suite à n'importer qu'une partie de nos images par classe si on le souhaite (les 500 premières d'une classe qui aurait 1000 images par exemple).

Chapitre 4

Images non redimensionnées

4.1 Niveaux de gris

Nous avons donc, dans un premier temps, effectué nos tests sur les images non redimensionnées, passées en niveaux de gris.

4.1.1 Courbes d'apprentissage

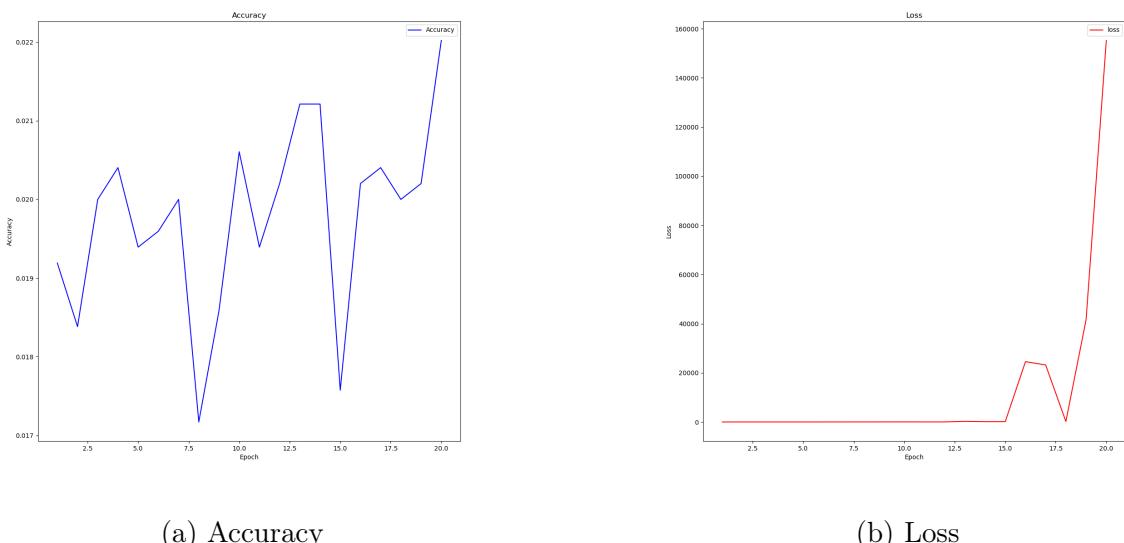


FIGURE 4.1 – Courbes d'apprentissage sur les images en niveaux de gris

4.1.2 Tests visuels

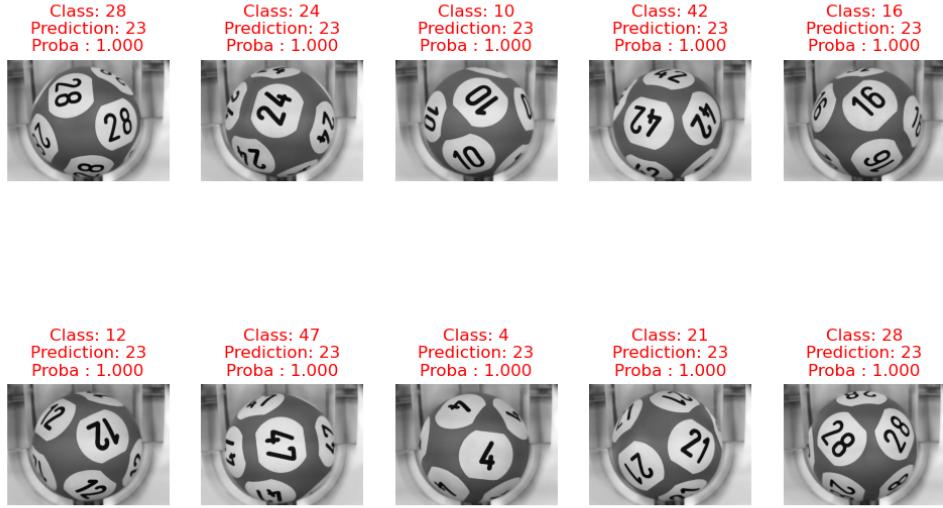


FIGURE 4.2 – Test visuel

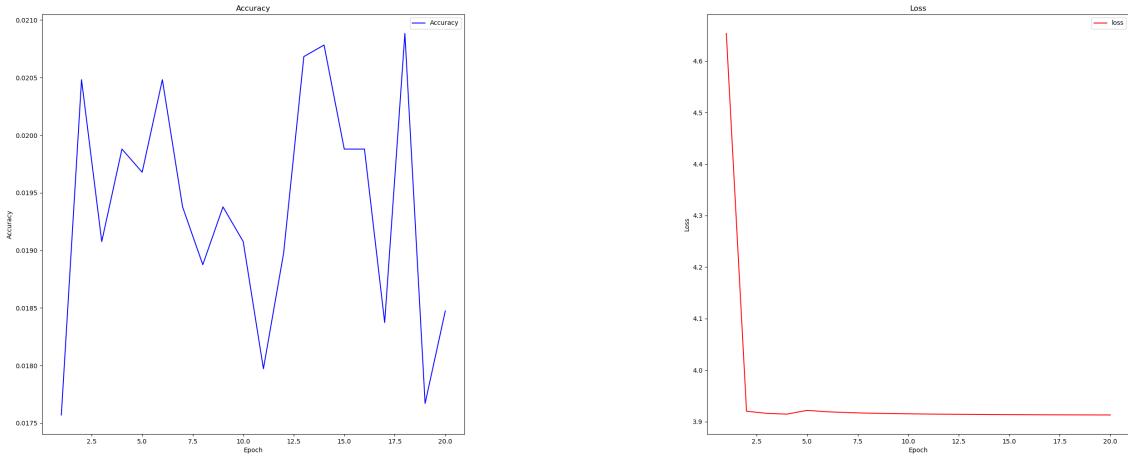
Sur ces quelques tests visuels (non exhaustifs), on voit que toutes les images ont été prédites comme étant une boule de numéro 23, avec une probabilité de 1. Je pense ici que, parce que les images présentent trop de détails, le réseau de neurones n'a pas su s'adapter pour chaque classe, et a convergé vers le fait de prédire l'image comme une classe unique de boule, et non de prédire les numéros comme on le souhaite.

4.2 Méthode Otsu

Nous avons ensuite réalisé les mêmes tests, mais cette fois ci en seuillant nos images avec la méthode Otsu.

4.2.1 Courbes d'apprentissage

Nous pouvons voir sur les courbes ici à la fois que l'accuracy est très faible et ne dépasse pas les 0.210 (soit 2.10% d'accuracy) et la loss reste très élevée (entre 3.9 et 4).



(a) Accuracy

(b) Loss

FIGURE 4.3 – Courbes d'apprentissage sur les images en niveaux de gris

4.2.2 Tests visuels

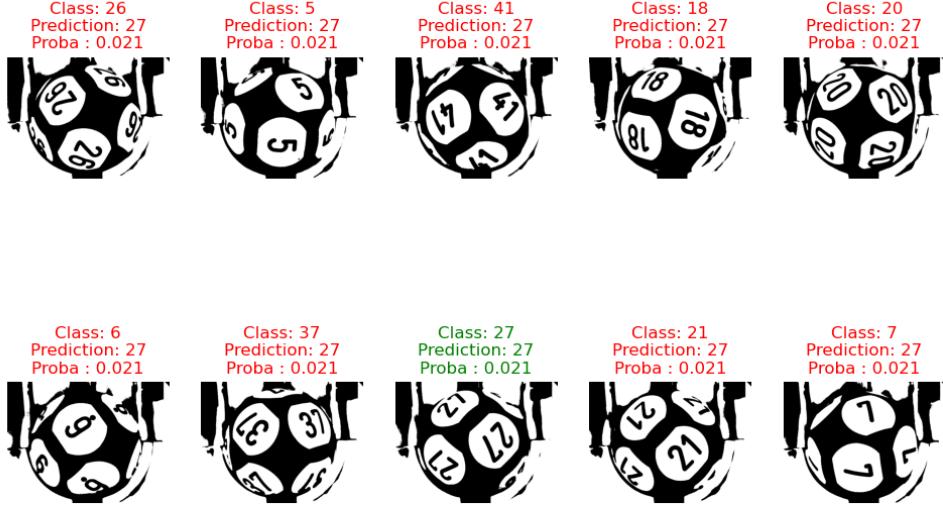


FIGURE 4.4 – Test visuel 2

On peut voir sur ces quelques tests visuels que la prédiction effectuée sur nos images est la même à chaque fois. On peut ici supposer que, comme pour l'entraînement avec les images en niveaux de gris (section 4.1) le réseau a convergé dans le fait de reconnaître la boule en elle-même, et non le numéro inscrit (chose qui nous intéresse en réalité). Refaisons les tests à nouveau avec le dernier prétraitement que nous avons.

4.3 Prétraitemet

Nous allons ici appliquer à nos images une suite de transformations, ayant pour objectif d'améliorer la qualité de l'image pour pouvoir dans l'idéal, faire de meilleures prédictions.

Le prétraitemet ainsi utilisé suit les étapes suivantes :

- Passage de l'image en niveaux de gris
- Application d'un flou gaussien pour supprimer le bruit des images
- Seuillage de l'image afin de séparer les numéros du reste
- Remplissage des trous qui peuvent apparaître dans les numéros, pour avoir des numéros pleins.

Pour donner un exemple de la dernière étape, voici une image que l'on peut trouver sur la documentation de la librairie [opencv](#).



FIGURE 4.5 – Remplissage des trous, sur la documentation opencv

4.4 Courbes d'apprentissage

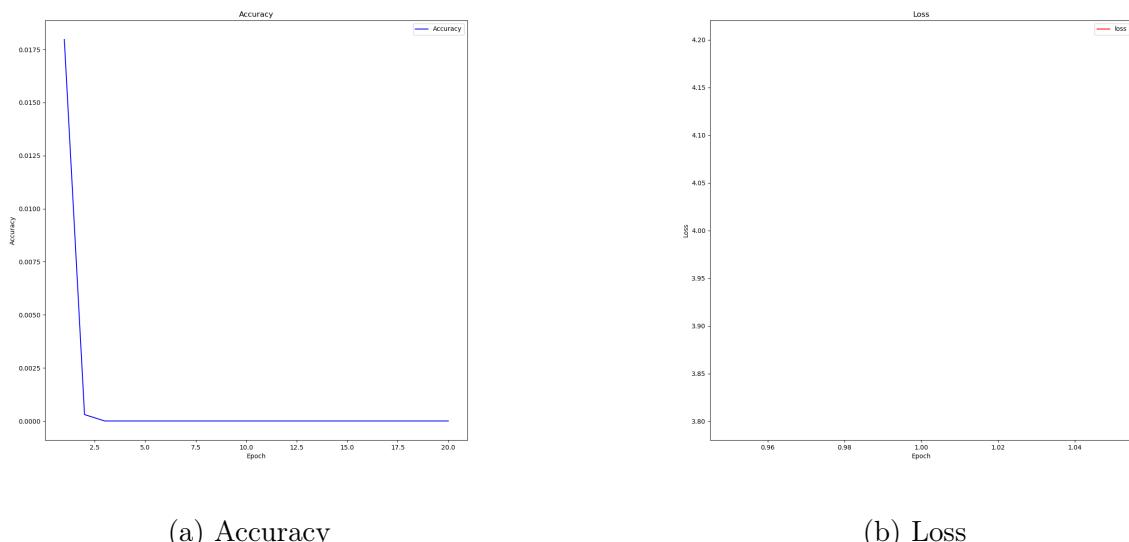


FIGURE 4.6 – Courbes d'apprentissage sur les images pré-traitées

On voit ici que la courbe d'accuracy a atteint 0 dès la première époque d'apprentissage, et la courbe de loss n'apparaît pas. Sur les tests visuels de la figure 4.7, que les probabilités sont à NaN (Not A Number). Il y a eu un problème ici lors de l'apprentissage, mais je

ne saurais dire pourquoi, le code utilisé étant le même que pour les courbes obtenues lors des sections 4.1 et 4.2, on aurait pu s'attendre à tout de même avoir des résultats, même mauvais.

4.5 Tests visuels

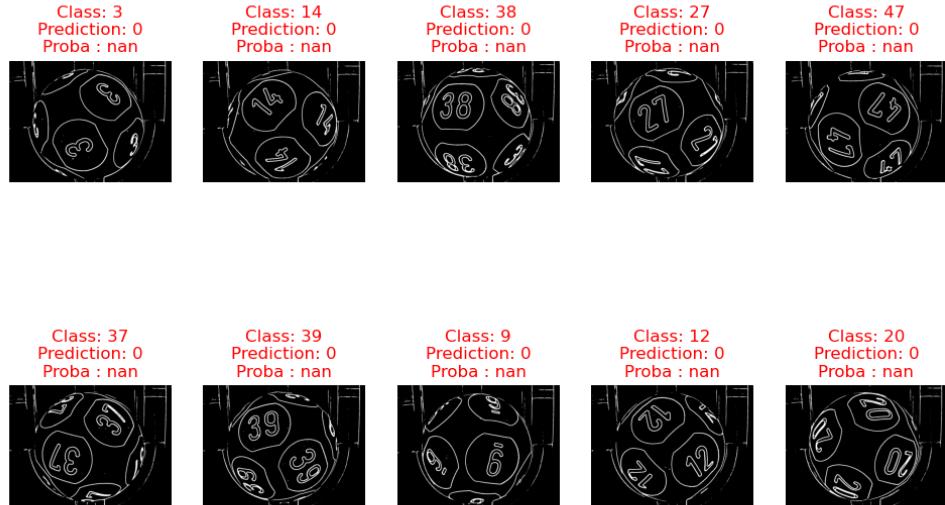


FIGURE 4.7 – Test visuel 1

Nous pouvons voir que, sur les 3 versions différentes (avec les 3 prétraitements différents), nos résultats sont mauvais, avec même des prédictions à NaN (Not A Number) pour la 3^e version de prétraitement.

Nous allons donc changer de méthode, et essayer de trouver une manière de rogner automatiquement nos images autour des numéros afin de faire de la prédiction sur les numéros uniquement, et non la boule entière.

Chapitre 5

Images redimensionnées - Rognage

Afin de rogner les images au niveau des numéros, nous allons utiliser la bibliothèque ***opencv-python*** qui nous offre beaucoup d'outils de manipulation d'image. OpenCV voulant par ailleurs dire Open Source **Computer Vision**.

5.1 Détection de contours

La bibliothèque opencv nous offre la possibilité sur des images en niveaux de gris de détecter des contours.

```
img = cv.imread(f"path_to_image")

img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

_, thresh_gray = cv.threshold(img_gray, 128, 255, cv.THRESH_BINARY)

contours_gray, hierarchy_gray = cv.findContours(thresh_gray, cv.RETR_TREE,\n        cv.CHAIN_APPROX_SIMPLE)
```

Ces 4 lignes de codes, qui seront à la base de tous les tests qui vont suivre, permettent de :

- Lire une image en couleurs
- Transformer cette image en une image en niveaux de gris
- Seuiller cette même image selon un seuil choisi, ici 128. Tout pixel en dessous de 128 en niveau de gris sera transformé en pixel noir, sinon, en pixel blanc.
- Détecter les contours, la dernière ligne renvoie un vecteur de points définissant ainsi les contours. On peut très facilement afficher ces contours là par la suite sur l'image importée initialement. Un exemple sur la figure 5.1 :

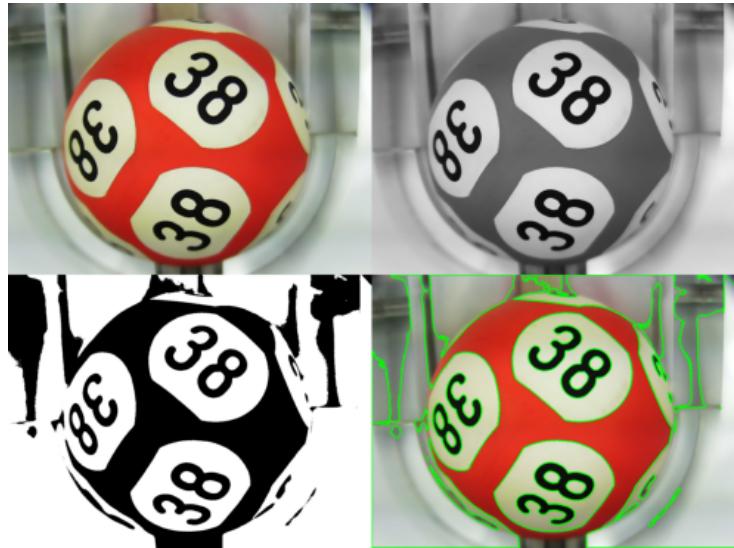


FIGURE 5.1 – Exemple

On voit ainsi sur la figure 5.1 quatre images. L'image qui a été initialement importée. La même image en niveaux de gris. L'image en niveau de gris qui a été seuillée, et enfin l'image initiale avec les contours déssinés, détectés sur l'image seuillée.

On peut cependant voir qu'opencv détecte certains contours non voulus tout autour de la boule. Ce n'est pas optimal, mais pour pallier à ce problème, opencv nous offre la possibilité avec une liste de points définissant un contour, de calculer l'aire induite en nombre de pixels, et on peut ainsi filtrer nos contours selon une aire minimale ou maximale.

En appliquant ce filtrage avec une aire minimale voulue de, 700 pixels et une aire maximale de 25000 pixels, on peut obtenir un filtrage comme sur la figure 5.2 :

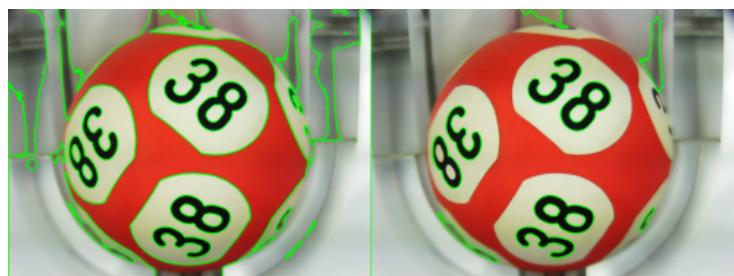


FIGURE 5.2 – Contours filtrés

Ce filtre n'est pas des plus parfait, en voyant les quelques petites zones restantes, qu'on pourrait augmenter l'aire minimale voulue, mais cela pourrait poser problème pour les boules 6 et 9 qui se distinguent avec un détail en plus.

Si nous décidions de mettre une aire minimale voulue trop grande, ce "détail" en plus pourrait être exclu, et il serait par la suite impossible de différencier un 6 d'un 9.



FIGURE 5.3 – Exemple d'un 6



FIGURE 5.4 – Exemple d'un 9

Pour ce qui est du 9, on voit que la petite barre n'a pas été détectée dans la troisième partie de la photo car l'aire minimale requise a été mise ici à 3000 pixels. La petite boule pour le 6 semble avoir été détectée. Cela est dû au fait que sur les boules de loto 6, la petite boule sous le 6 est souvent très proche du numéro en lui-même. Il arrive ainsi que le 6 ainsi que sa boule soit considérés dans le même contour.



FIGURE 5.5 – Un 6 et un 9

5.2 Extraction des numéros

Par la suite, il nous faut pouvoir lier ensemble différentes régions si les contours associés à ces régions sont proches, pour pouvoir compter le 3 et le 8 comme un 38 de la figure 5.2 et non comme deux nombres séparés, ou pour tout autre nombres à deux chiffres.

Pour cela, nous utilisons la fonction **boundingRect** de la librairie opencv qui nous permet, avec un contours, de déterminer le rectangle englobant de ce contour.



FIGURE 5.6 – Rectangles englobants des contours

Nous avons donc la possibilité de trouver des rectangles englobant les contours, mais nous aimerais pouvoir combiner des rectangles si ceux-ci sont proches (comme sur la figure 5.7) afin de considérer un plus grand rectangle englobant.

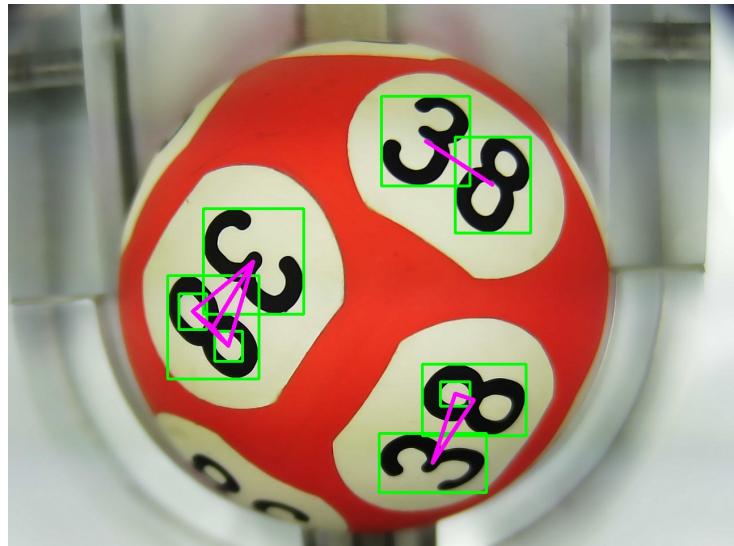


FIGURE 5.7 – Rectangles englobants proches

Les lignes en rose sur la figure 5.7 signifient que 2 régions ont été déterminées comme proches.

La librairie opencv définit les rectangles comme un point initial (x, y) , et des distances représentant la largeur et la hauteur des rectangles. Ainsi, on peut définir un rectangle par le 4-uplet : (x, y, w, h) .

Ainsi, deux rectangles peuvent être considérés comme deux 4-uplet : (x_1, y_1, w_1, h_1) et (x_2, y_2, w_2, h_2) .

Ces rectangles seront considérés comme proches si leurs points centraux sont proches deux à deux. L'idée ici est donc de calculer les coordonées des points centraux des rectangles, de calculer la distance Euclidienne entre les centres deux à deux, et si cette distance est inférieure à un seuil choisi, alors on considère les rectangles comme proches.

```
(x1, y1, w1, h1) = rect1
(x2, y2, w2, h2) = rect2

centerX1, centerY1 = (x1 + x1 + w1) // 2, (y1 + y1 + h1) // 2
centerX2, centerY2 = (x2 + x2 + w2) // 2, (y2 + y2 + h2) // 2

distance = sqrt((centerX2 - centerX1)**2 + (centerY2 - centerY1)**2)

if distance < seuil:
    return True
```

En appliquant ce bout de code sur nos différents rectangles, on obtient par la suite, les rectangles fusionnés suivant :

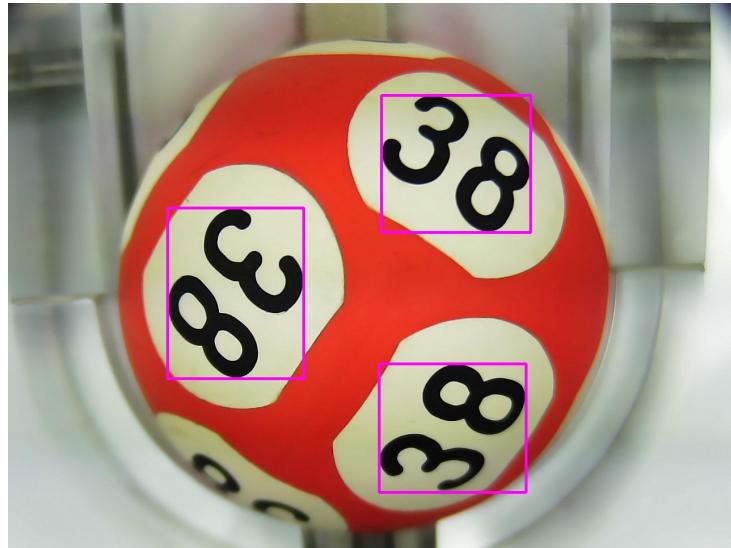


FIGURE 5.8 – Rectangles fusionnés

Enfin, de ces rectangles fusionnés, nous déterminons le point central, et, ne pouvant s'assurer que tous les nombres ne seront pas détectés de la même manière et n'auront pas la même orientation, nous extrayons une zone arbitrairement large autour du point central pour s'assurer au mieux d'extraire le nombre entier.

Les réseaux de neurones de la librairie tensorflow que nous utilisons ne prennent que des images de taille fixe, nous ne pouvons donc pas extraire la partie de l'image incluse dans un rectangle précisément, celle ci variant d'une région à l'autre.

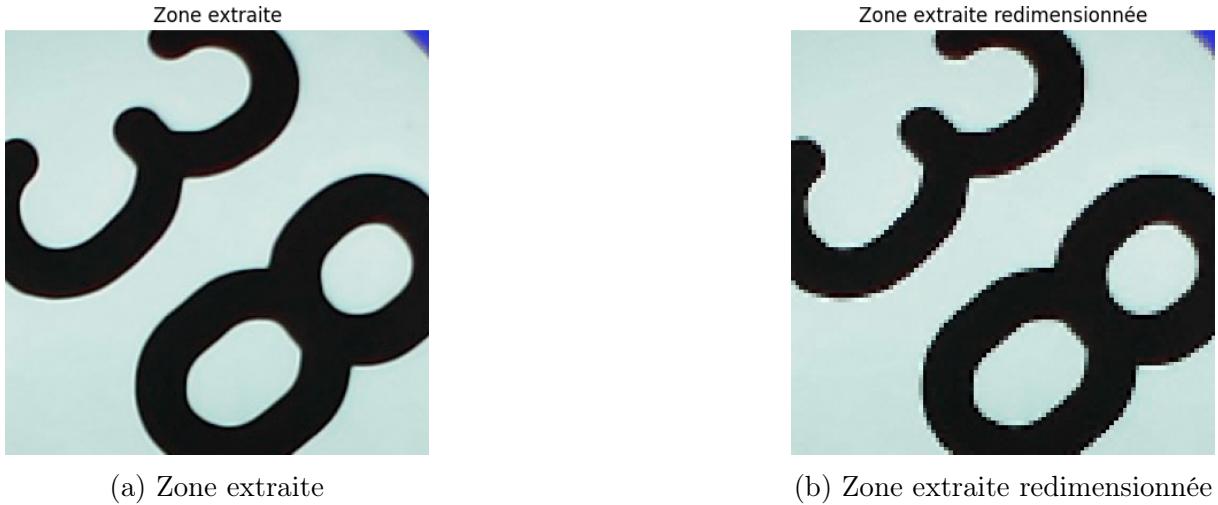
Soit le 4-uplet (x_1, y_1, x_2, y_2) définissant un rectangle fusionné, cette fois ci, le point supérieur gauche, et le point inférieur droit. Nous pouvons donc extraire une partie via le code suivant :

```

x1, y1, x2, y2 = merged_rectangle
centerX, centerY = (x1+x2)//2, (y1+y2)//2
toExtract = img_extraction[centerY-120:centerY+120, centerX-120:centerX+120]

```

L'image **img_extraction** représente l'image prétraitée, si un prétraitement a été demandé. Du point central de ce rectangle, nous prenons une zone carrée l'entourant de taille (240×240) , que nous redimensionnons en une image de taille (100×100) , afin de réduire le temps de calcul et d'entraînement, ainsi que la taille en mémoire utilisée par toutes nos images.



Enfin, en appliquant tout cela sur les images de notre jeu de données, il est ici facile de se créer un ensemble d'images rognées et leurs labels associés, qu'on passera par la suite à notre réseau de neurones.

5.3 Crédation du réseau de neurones et entraînement

La création d'un réseau de neurones avec la librairie *Keras* de *Tensorflow* se fait très simplement en quelques lignes de code.

Une première ligne de code qu'on appelle en général l'*input_layer*, où l'on indique notamment la *shape* (dimension) de nos images. On peut donc indiquer ce paramètre de manière dynamique après avoir importé nos images, et appliqué ou non nos modifications en récupérant la *shape* d'une image (toutes les images devant avoir la même *shape*). Par exemple, une image en couleur pourra avoir une *shape* égale à $(240, 240, 3)$ (image en couleurs), tandis que la même image, passée en niveaux de gris aura pour *shape* $(240, 240)$ ou alors $(240, 240, 1)$. Ces deux *shape* veulent dire la même chose, chaque pixel sera alors défini non par 3 valeurs définissant une valeur RGB, mais 1 seule valeur, définissant son intensité de gris, pour un CNN, on passera dans l'*input_layer*, la shape $(240, 240, 3)$ ou $(240, 240, 1)$, les CNN demandant des images en 3 dimensions.

Les lignes suivantes sont des lignes de couches de convolution **Conv2D** et/ou de pooling **MaxPooling2D**.

Les couches de convolution **Conv2D** est chargée de faire des convolutions sur des zones de l'image. La convolution consiste en des calculs matriciels afin d'en sortir une valeur

spécifique. C'est principalement ces couches là qui vont détecter les **features** de nos numéros et qui permettront de différencier un 1 d'un 7 par exemple.

Les couches de pooling **MaxPooling2D** sont quant à elles chargées de réduire la taille de nos images en prenant dans une taille de fenêtre (entrée en paramètres) la valeur maximale de ces fenêtres. Cela permet de réduire petit à petit la taille de nos images, et donc de réduire le temps de calcul total, et permet aussi d'introduire une invariance par translation.



(a) Croisement 1



(b) Croisement 2

Pour illustrer la couche **MaxPooling2D**, les figures 5.10a et 5.10b sont extraites d'une même image, mais la croix du 4 est située à différentes coordonnées (translation). À terme, la couche **MaxPooling2D** permettra donc pour un même croisement de 4, d'obtenir la même valeur après convolution, et donc pour des croisements situés à différents endroits, le réseau de neurones fera ainsi la prédiction d'un 4.

Les couches **Conv2D** étant également accompagnées d'une fonction d'activation, dans notre cas, nous utilisons la fonction d'activation *Relu* pour Rectified Linear Unit, définie par $f(x) = \max(x, 0)$. Un neurone s'activera donc si après avoir fait une convolution, une feature est détectée et la valeur x sera renvoyée, sinon, la valeur renvoyée est 0, et la feature sera considérée absente.

Après les couches **MaxPooling2D** et **Conv2D**, on rajoute une couche **Flatten** chargée de transformer une matrice en un vecteur unidimensionnel afin de pouvoir passer ce qu'il reste de notre image à la fin de notre réseau de neurones.

Puis une couche **Dense** chargée d'extraire les features (les couches **Conv2D** détectaient les features, les couches **Dense** les extraient), et enfin, une dernière couche **Dense** qui sera aussi une couche d'activation, avec pour fonction d'activation **softmax**. La fonction softmax nous renverra une distribution de probabilités, et nous renverrons de notre côté la prédiction ayant la plus grande probabilité dans cette distribution.

```

model = tf.keras.Sequential(
[
    tf.keras.layers.Conv2D(32, (3,3), padding='same', activation="relu",
    input_shape=(32, 32, 3)),

    tf.keras.layers.MaxPooling2D((2, 2), strides=2),

    tf.keras.layers.Conv2D(64, (3,3), padding='same', activation="relu"),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
]
)

```

Un exemple [ici](#) de réseau de neurones convolutionnel simple permettant ici :

- De prendre en entrée des images en couleurs de taille 32 x 32 (input_shape = (32,32,3))
- De prédire 10 classes différentes, la dernière ligne étant la couche d'activation softmax, ayant pour premier paramètre 10, pour 10 classes.

Une fois notre réseau de neurones construit, nos images importées et séparées en un ensemble d'entraînement, et un ensemble de test, l'entraînement de notre réseau se fait en appelant la fonction **fit** de notre modèle :

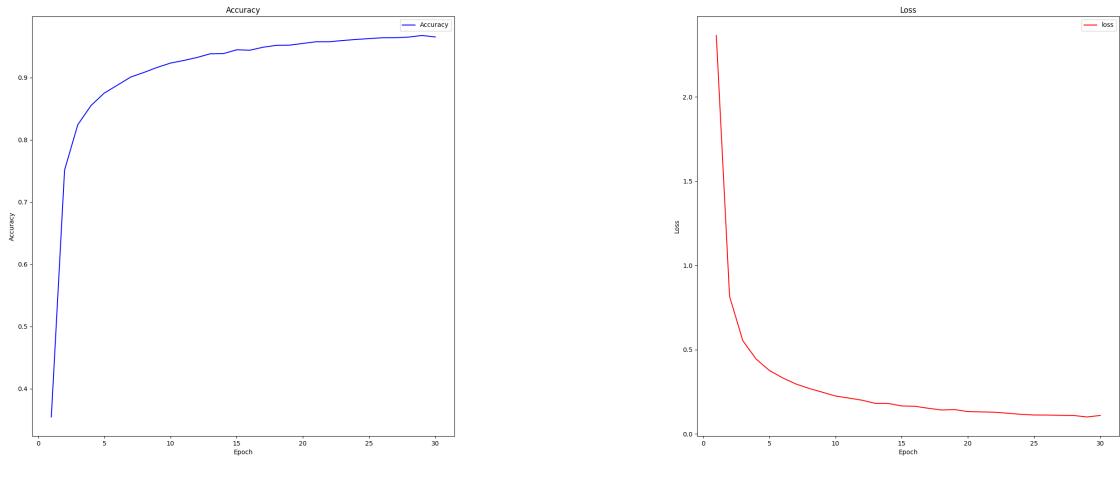
```
model.fit(X_train,y_train, epochs=600,
           validation_data=(X_test,y_test), callbacks=callbacks)
```

5.4 Résultats

Les entraînements qui vont suivre ont été effectués sur 30 époques d'entraînement, en ayant importé et extrait des régions sur les 500 premières images de chaque classe. Ce qui conduisait à environ 78.000 images pour le jeu de données complet, soit environ 1.560 images par classe.

5.4.1 Niveaux de gris

On voit ici une accuracy atteignant un très haut score proche des 0.95 ainsi qu'un loss très faible, proche des 0.2.



(a) Accuracy

(b) Loss

FIGURE 5.11 – Courbes d'apprentissage sur les images rognées en niveaux de gris



FIGURE 5.12 – Test visuel

5.4.2 Preprocessing

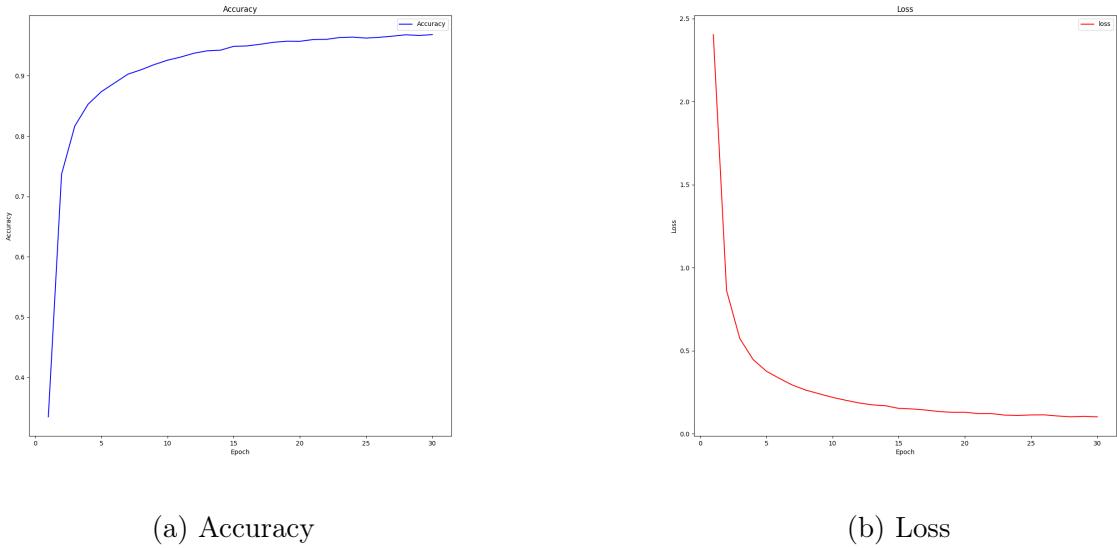


FIGURE 5.13 – Courbes d'apprentissage sur les images rognées en niveaux de gris

Sur cette version pré-traitée des images, on atteint là aussi, une haute accuracy proche de 0.95, ainsi qu'une faible loss.

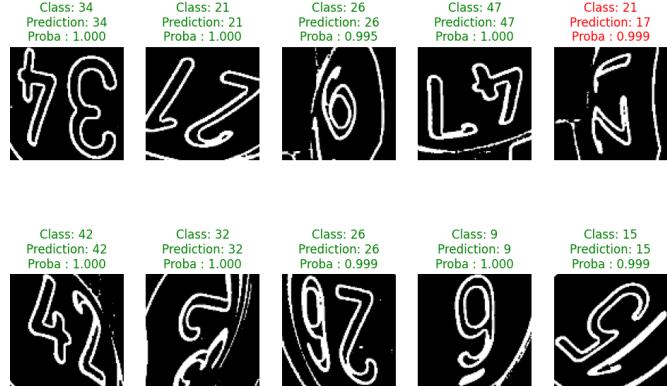


FIGURE 5.14 – Test visuel

Chapitre 6

Test sur des images non utilisées à l'entraînement

6.1 Extraction et prédictions de zones

Le fichier **prediction.py** (utilisé rapidement à la fin du chapitre 2) permet de sélectionner un modèle avec lequel on souhaite faire des prédictions, et une image à prédire. L'image qui aura été sélectionnée va passer par les étapes de détection de contours et d'extraction de zones, puis le modèle va faire une prédiction sur chacune de ces zones extraites.



FIGURE 6.1 – Prédiction sur une boule 37

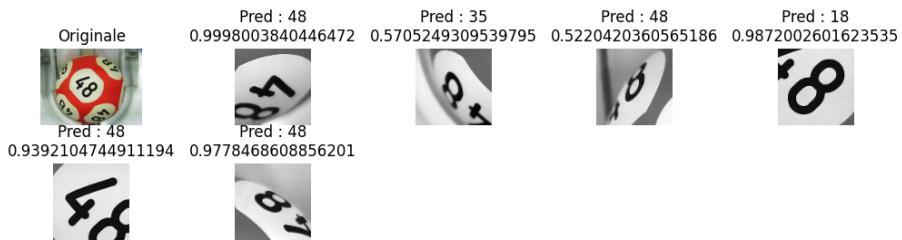


FIGURE 6.2 – Prédiction sur une boule 48

On voit qu'on a déjà de meilleurs résultats que les essais effectués au chapitre 2, mais cela ne semble pas encore parfait, puisque cela dépend de si la détection de contours et d'extraction a bien marché. Notamment sur la figure 6.2 où sur les 6 régions extraites, il y a eu 1 région prédite comme un 35, ou sur la figure 6.1 où sur 2 régions, une a été prédite comme un 23 au lieu d'un 37, avec une probabilité tout de même plus faible.

6.2 Agrégation des probabilités

Enfin, maintenant que nous pouvons charger une image, faire de la prédiction avec un réseau de neurones pré-entraîné, et obtenir des probabilités sur les régions, il nous faut un moyen de mettre en relation les différentes probabilités calculées.

Les deux possibilités qui s'offre à nous les plus simples sont :

- Un produit de probabilités
- Une moyenne des probabilités

Quelques exemples :

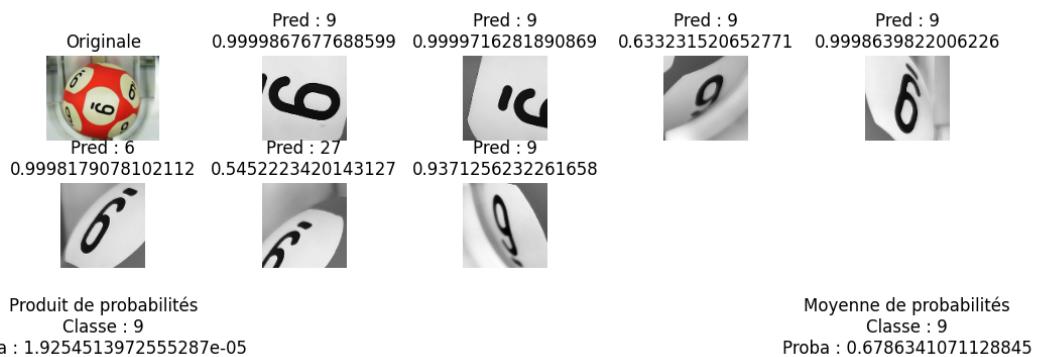


FIGURE 6.3 – Prédiction sur une boule 9

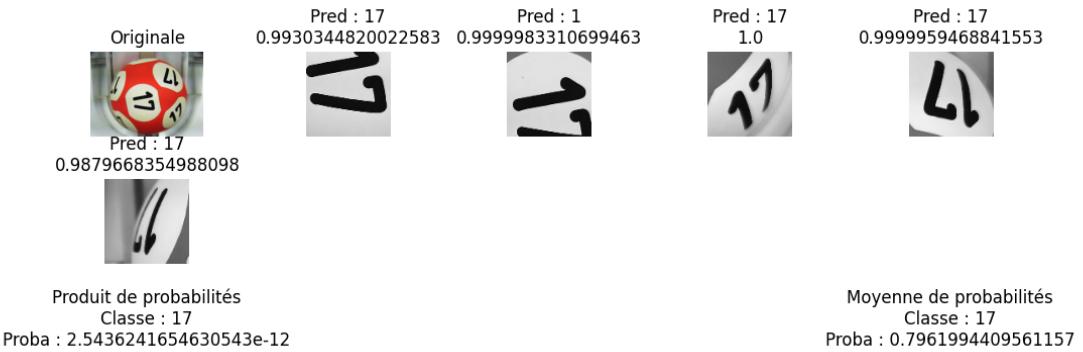


FIGURE 6.4 – Prédiction sur une boule 17

En revanche, aucune de ces deux manières de faire n'est infaillible, il arrive que la version "moyenne" se trompe mais pas la version "produit". Comme on peut le voir sur la figure 6.5.

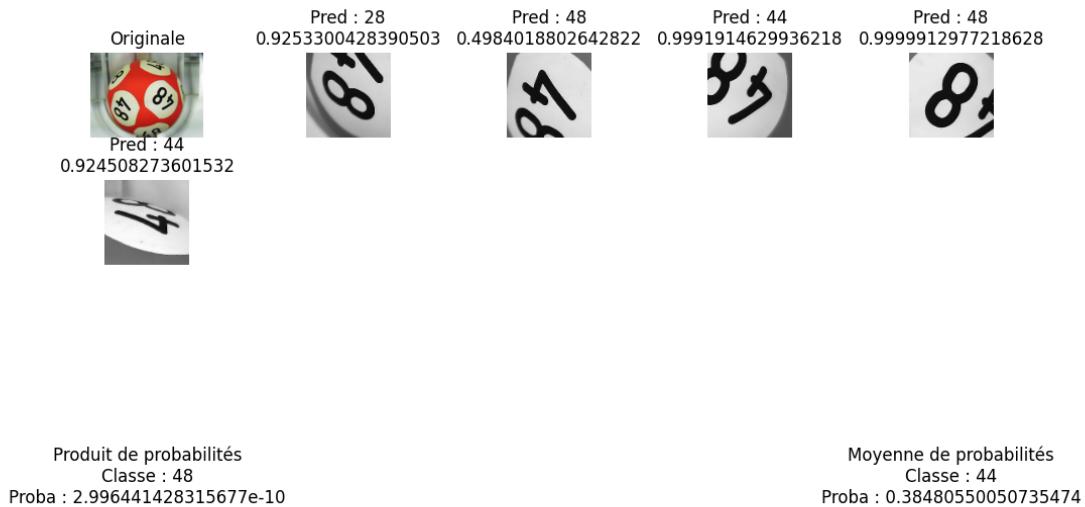


FIGURE 6.5 – Prédiction sur une boule 48

On voit ici que les prédictions faites par produit de probabilités atteints des valeurs très basses, jusqu'à $2.54e - 12$ (figure 6.4), s'expliquant par le fait que les probabilités étant inférieures à 1, des valeurs basses sont vite atteintes, mais si la valeur correspondant à la bonne prédiction est élevée, il est normal que la prédiction finale soit la plus élevée pour la bonne prédiction.

Chapitre 7

Conclusion et Ouverture

Pour conclure, je dirais que ce stage m'a permis d'approfondir mes connaissances sur les réseaux de neurones et le machine learning, ainsi que mes connaissances sur le traitement d'image et notamment l'utilisation de la librairie OpenCV qui est une librairie très puissante pour le traitement d'image.

En ouverture de ce stage, il serait intéressant de trouver peut être une meilleure manière d'agréger les probabilités sur les prédictions finales, et/ou de trouver une manière d'ajouter un filtre à l'extraction ou sur les régions extraites, afin de ne prendre que les régions où l'on voit de manière optimale les numéros et qu'ils soient coupés le moins possible.

Chapitre 8

Références

<https://fr.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks-1489512765771.html>

<https://cs231n.github.io/convolutional-networks/>

https://www.tensorflow.org/api_docs/python/tf

Bibliographie

- [1] T. RAGUNTHAR, Anuj SINGH et Sanskar SARAWGI. « Handwritten digit recognition using CNN and KNN ». In : *AIP Conference Proceedings*. 2024 (page 1).
- [2] Savita AHLAWAT, Amit CHOUDHARY, Anand NAYYAR, Saurabh SINGH et Byung-un YOON. « Improved Handwritten Digit Recognition Using Convolutional Neural Networks (CNN) ». In : *Sensors* (2020) (page 1).