



SORBONNE UNIVERSITÉ
MASTER ANDROIDE

Reconnaissance de nombres contenus sur surfaces sphériques : application au tirage du loto

UE de projet M1

Antony LECLERC – Shirel AMOZIEG – Zhirui CAI
Sous la direction de : Thibaut Lust

2023 – 2024

Table des matières

1	Introduction	1
2	État de l'art	2
3	Contribution	3
3.1	Outils utilisés	3
3.2	Mais qu'est-ce qu'un Réseau de Neurones ?	5
4	CNN - Convolutional Neural Network	6
4.1	Comment est construit un CNN ?	6
4.2	Les filtres dans les CNN	8
4.3	Fonctionnement de notre code	8
5	Images Brutes	9
5.1	Images en couleur	9
5.2	Images en noir et blanc	12
5.3	Images preprocessed	13
6	Images rognées	15
6.1	MNIST	15
6.2	Problème des images rognées	18
6.3	Images en couleur	19
6.4	Images en noir et blanc	20
6.5	Images preprocessed	21
6.6	Images en couleurs, et rotation à 360°	22
7	Conclusion	24
A	Cahier des charges	25
B	Manuel utilisateur	26
C	Références	29

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 ; surface sphérique.

C’est dans ce contexte que s’inscrit notre projet, qui vise à explorer et adapter les techniques de reconnaissance des nombres sur l’application concrète du LOTO.

Les boules du LOTO, avec leur forme sphérique, présentent un défi pour la reconnaissance de leur numéro traitée par l’OCR standard.

Le but du projet est donc d’une part, d’analyser les limites des méthodes traditionnelles d’OCR existantes dans notre contexte de reconnaissance sur surface sphérique. Et d’autre part, de réfléchir à une solution éventuelle qui surmonte ces obstacles en exploitant les dernières avancées des réseaux de neurones convolutionnels (CNN) et de l’apprentissage profond.

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.

Chapitre 2

État de l'art

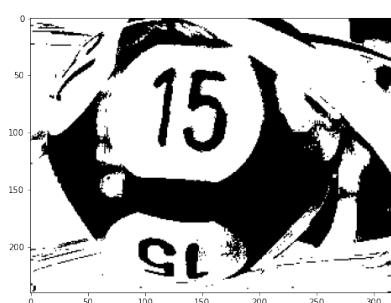
Les méthodes déjà existantes pour la reconnaissance d'images sont, en général, des méthodes se basant sur l'*OCR* (pour « *Optical Character Recognition* »), soit la Reconnaissance Optique de Caractères.

Le principe de l'*OCR* est de lire une image, et de binariser cette dernière, les zones sombres de l'image seront classées comme 0, et les zones claires comme 1. Et par la suite effectuer un *matching pattern* avec ce que le programme utilisé connaît déjà.

Grâce à cette méthode, les traitements effectués sur l'image résultante ont un pourcentage plus élevé de satisfaction.



(a) Image normale



(b) Image binarisée

FIGURE 2.1 – Exemple d'image binarisée pour *OCR*.

Chapitre 3

Contribution

3.1 Outils utilisés

Les deux outils principaux que nous avons ainsi choisi d'utiliser sont :

1. Des Réseaux de Neurones, RNN (*«Regular Neural Network»*) puis CNN (*«Convolutional Neural Network»*), soit Réseau de Neurones Convolutionnel en Français ; il s'agit d'une architecture réseau pour le deep learning qui apprend directement depuis des images. Ces derniers sont donc particulièrement adaptés pour travailler sur nos images de LOTO.
2. Le langage de programmation Python.

Plusieurs outils s'offraient à nous :

- Pytorch
- Tensorflow
- Keras
- Vertex AI
- Scikit-Learn

Tous ont cependant leurs avantages et leurs inconvénients que nous avons mis en avant dans le tableau suivant.

	Avantages	Inconvénients
PyTorch	<ul style="list-style-type: none"> • Simple d'utilisation • Rapide • Efficace 	<ul style="list-style-type: none"> • Peu stable • Petite communauté
TensorFlow	<ul style="list-style-type: none"> • Beaucoup d'outils • Flexible • Grande communauté 	<ul style="list-style-type: none"> • Difficile à prendre en main (pour les débutants) • Lent
Keras	<ul style="list-style-type: none"> • Se base sur TensorFlow • Prototypes rapides et faciles 	<ul style="list-style-type: none"> • Pour des implémentations rapides • Petits jeux de données • Petite communauté
Vertex AI	<ul style="list-style-type: none"> • Bien pour les néophytes de l'IA • Conventions • Beaucoup d'outils 	<ul style="list-style-type: none"> • Conventions, changer la manière dont Vertex fonctionne peut être problématique • Anciens codes souvent incompatibles
Scikit-Learn	<ul style="list-style-type: none"> • Facile à utiliser • Accessible • Grande communauté internationale 	<ul style="list-style-type: none"> • Inadapté à l'apprentissage profond • Moins efficace que TensorFlow

TABLE 3.1 – Comparaison des outils disponibles

Après comparaisons, nous avons choisi pour ce projet d'utiliser la bibliothèque **Tensorflow** en plus de **Keras**, (Keras se basant sur Tensorflow, les deux étant ainsi complémentaires).

3.2 Mais qu'est-ce qu'un Réseau de Neurones ?

Neural nets are an approach to data analysis unrelated to statistical methods. They use a mathematical pattern recognition paradigm to "learn" the complex interactions between input variables and outputs as it "trains" on a known set of data. It then uses these learned patterns to estimate the expected output when presented with new inputs. The neural net thus develops "common sense," as Minsky might put it, based on a historical data base. In medicine, we use the term "clinical judgment" for such a decision based on our prior learning and experience.

Un réseau de neurones est donc un outil informatique, auquel on passe un jeu de données ainsi que le label de ces données en entrée, qu'il va apprendre à reconnaître, et doit être par la suite en mesure de reconnaître et classifier des données similaires qu'il n'aura pas vu lors de sa phase d'entraînement.

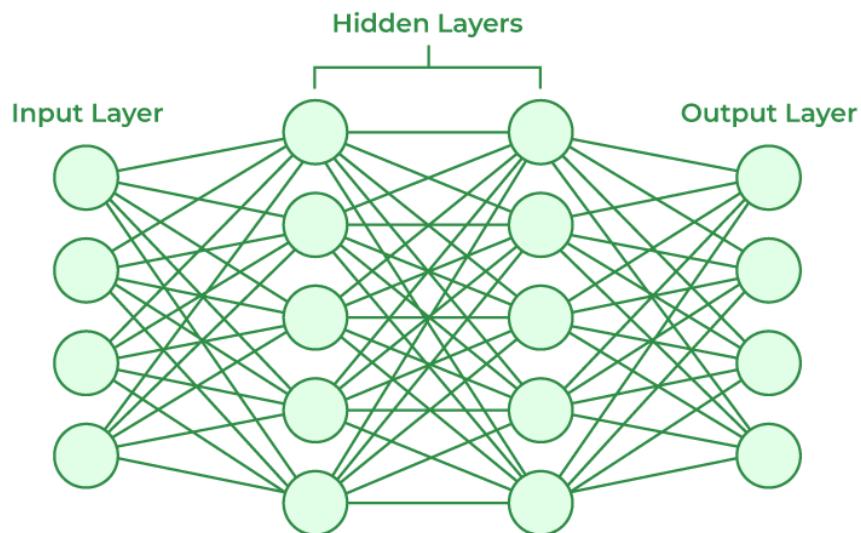


FIGURE 3.1 – Architecture d'un réseau de neurones

Les réseaux de neurones sont construit avec une succession de couches, elle mêmes constituées de neurones, chaque neurone d'une couche est relié à chaque neurone de la couche suivante, et chacun de ces liens est pondéré avec un poids, qui donnera plus ou moins d'importance à un neurone, selon la donnée passée dans la couche d'entrée (*input layer*).

Chapitre 4

CNN - Convolutional Neural Network

4.1 Comment est construit un CNN ?

On a défini les réseaux de neurones comme étant une succession de couches, constituées elles même de neurones. Dans cette optique, et avec tensorflow, un *CNN* se construira de la façon suivante :

- Une première couche convolutionnelle de neurones **Conv2D**, appelée *input layer*. Cette couche prendra en argument un nombre de filtres à appliquer, un noyau de convolution (taille des filtres), une fonction d'activation, ainsi que la taille des images à passer au réseau de neurones.
- Une couche **MaxPooling2D** servant à réduire la dimensionnalité des données afin de réduire la complexité et le temps de calcul sur les convolutions suivantes.

Les couches **Conv2D** et **MaxPool2D** peuvent être répétées.

- Une fois toutes les convolutions effectuées, on ajoute à notre réseau une couche **Flatten** afin de réduire notre image à un vecteur de dimension 1.
- Puis des couches **Dense**, couches "classiques" dans un réseau de neurones.
- La dernière couche **Dense** a une fonction d'activation **Softmax**, c'est cette couche qui nous permettra d'obtenir une prédiction sur nos images.

Voici donc un exemple de réseau de neurones construit de cette manière, et aussi le premier qu'on va utiliser :

```
model.add(tf.keras.layers.Conv2D(32, (3, 3), activation='relu',
    input_shape=(240, 320, 3)))
model.add(tf.keras.layers.MaxPooling2D((2, 2)))
model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D((2, 2)))

model.add(tf.keras.layers.Flatten())

model.add(tf.keras.layers.Dense(128, activation='relu'))
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(51, activation='softmax'))

model.compile(optimizer='adam',
    loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Pour réexpliquer chaque ligne :

La première ligne **Conv2D** appliquera sur nos images 32 filtres de taille 3×3 , avec une fonction d'activation **relu**, et une *input_shape* en $(240, 320, 3)$, ce qui signifie qu'on passe au réseau de neurones, des images de taille 240×320 avec les canaux *RGB* (soit une image en couleurs).

La ligne suivante **MaxPooling2D** réduira notre image actuelle en une image de taille $(120, 160, 3)$ (le nombre de canaux de couleurs reste inchangé).

Une deuxième couche **Conv2D** qui va à nouveau appliquer des filtres de taille 3×3 avec une fonction d'activation **relu**.

Une couche **MaxPooling2D** qui va à nouveau réduire nos images en une taille $(60, 80, 3)$.

Une couche **Flatten** pour réduire nos images à un vecteur à 1 dimension.

Deux couches **Dense** de respectivement 128 et 64 neurones, de fonction d'activation **relu**.

Une dernière couche **Dense**, de fonction d'activation **softmax**, qui nous renverra un vecteur de probabilité pour chaque classe. Ici avec 50 neurones $(51 - 1)$. On choisira alors la classe avec la plus grande probabilité.

Enfin, une ligne s'occupant de compiler le réseau de neurones avec plusieurs paramètres :

- Un **optimiseur**, ici **adam**, méthode d'optimisation par descente de gradient stochastique.
- Une fonction de perte (**loss**), ici **sparse_categorical_crossentropy**.
- Une métrique "**accuracy**", donnée qui sera suivie lors de l'entraînement.

4.2 Les filtres dans les CNN

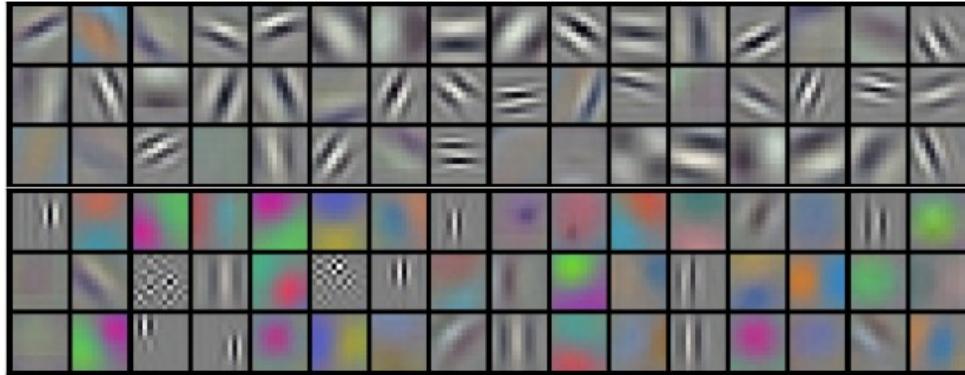


FIGURE 4.1 – Filtres de Krizhevsky

Les filtres utilisés dans des réseaux de neurones convolutionnels sont le plus souvent, initialisés à des valeurs aléatoires, et seront spécialisés dans la reconnaissance de **features** sur nos images. On définit une **feature** par une caractéristique particulière d'un numéro. Que ce soit des boucles, des bords horizontaux / verticaux, des angles...

4.3 Fonctionnement de notre code

Pour chacun de nos fichiers *test.py*, l'architecture de notre code reste plus au moins le même.

Nous importons dans un premier temps le CNN que nous allons utiliser pour le test, puis nous importons les images (telles quelles pour les brutes et générées pour les rognées). Des modifications ont été apportées avec la fonction *preprocessed* si nécessaire. Puis la fonction *evaluate* est appelée pour tester l'efficacité du réseau. Enfin, les résultats sont affichés pour voir visuellement l'issue du test.

Chapitre 5

Images Brutes

5.1 Images en couleur

Le premier réseau de neurones que nous avons essayé de faire est un réseau de neurones prenant en entrée les images brutes, non modifiées. Un exemple :



FIGURE 5.1 – Exemple d'image brute, ici un 7.

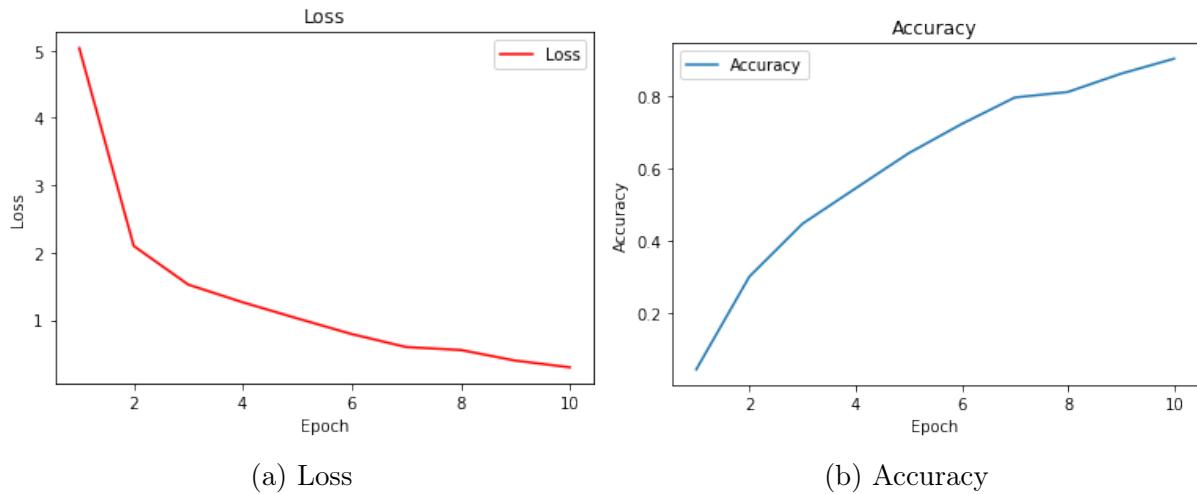
Cette image la à première vue pourrait être une bonne image pour l'apprentissage des images 7. Cependant, toutes les images ne sont pas aussi idéales, et certaines images seront par la suite bien plus problématiques pour faire de l'apprentissage.



FIGURE 5.2 – Exemple d'image brute, ici un 7.

On peut voir sur cette image là qu'il y'a trois 7, mais aucun de très viable, deux de ces 7 étant "couchés", et un au dessus, légèrement coupé et de travers.

Ainsi, sur les 10.429 images qui nous sont fournies, avec une répartition dans les ensembles d'entraînement et de test de 80% / 20% (soit environ 8343 images contre 2085), sachant que pas toutes ne sont utilisées, et sur 10 epochs d'entraînement, on obtient les courbes d'entraînement suivantes :



On obtient le résultat suivant :

13/13 6s 407ms/step - accuracy : 0.0658 - loss : 3876.1707. Une loss très élevée et une accuracy très faible. Les résultats visuels permettent de confirmer cela :

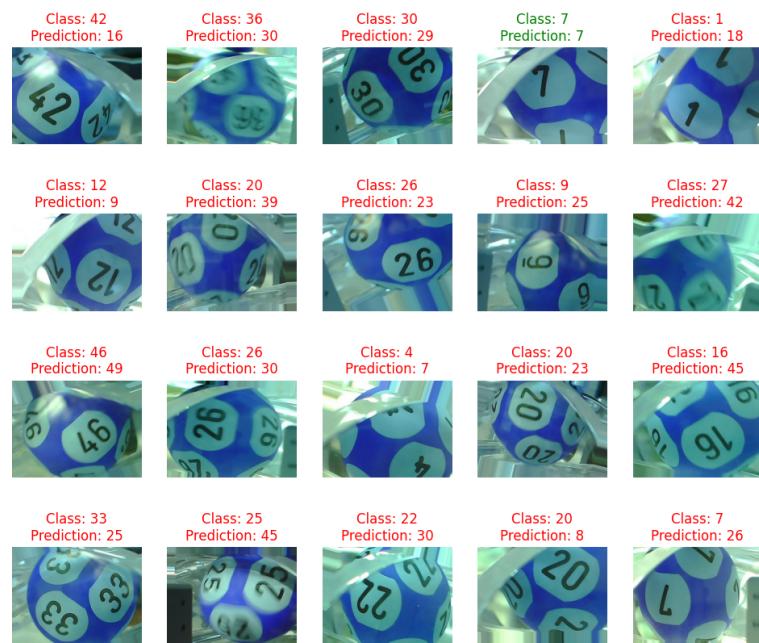


FIGURE 5.4 – Test 1

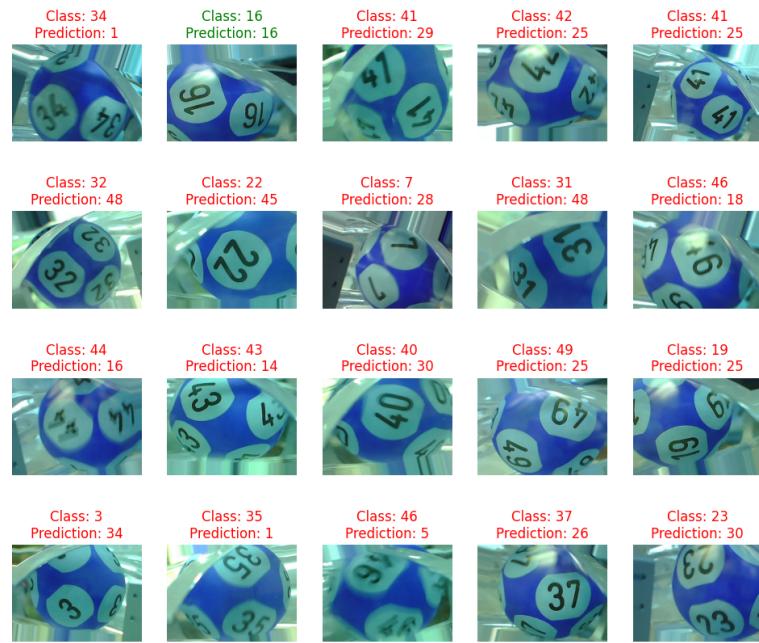


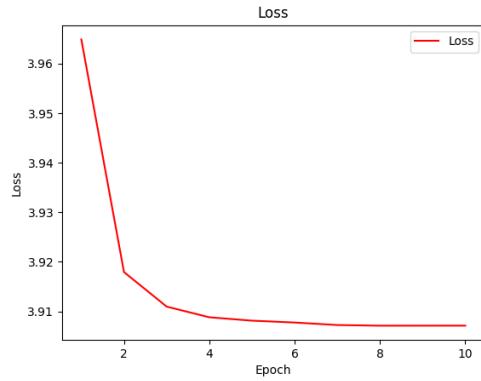
FIGURE 5.5 – Test 2

Sur 40 images données, seulement 2 ont été reconnues correctement. Cela n'est pas bien étonnant, c'est un résultat auquel nous nous attendions, et correspond d'avantage aux 6% indiqués ci-dessus qu'aux 90% indiqués par les courbes.

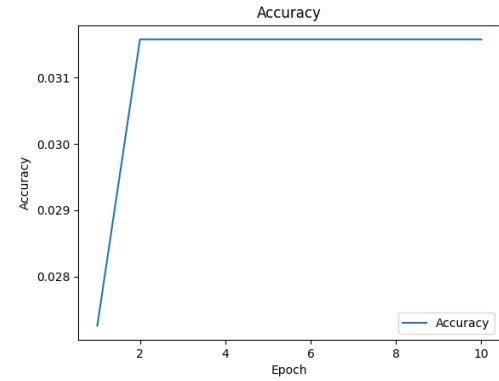
5.2 Images en noir et blanc

Nous avons ensuite décidé de tester et d'adapter le code pour un traitement des images en noir et blanc.

On charge donc directement les images en niveau de gris, ce qui simplifie le traitement ultérieur. Le traitement reste le même que précédemment. On suppose que ces pré-conditions devraient améliorer la performance du modèle en fournissant des données plus nettes et uniformes.



(a) Loss



(b) Accuracy

Les résultats obtenus ne sont pas aussi bons que ce à quoi nous aurions pu nous attendre. En effet, sur les sorties des tests obtenues, il n'est capable de reconnaître pratiquement jamais les numéros.

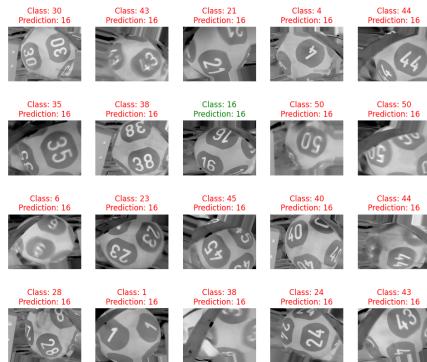


FIGURE 5.7 – Test 1

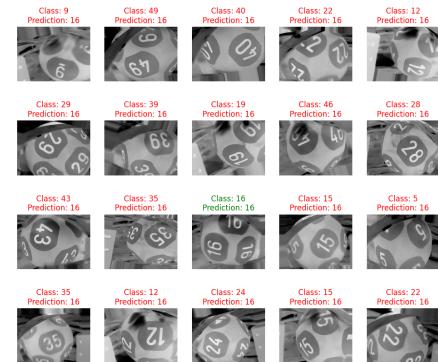


FIGURE 5.8 – Test 2

5.3 Images preprocessed

Au vu des résultats peu satisfaisants obtenus, nous avons ajouté l'utilisation d'une nouvelle fonction *preprocess_image*. Chaque image chargée passe par cette fonction de traitement. D'abord elle est convertie en niveau de gris, puis elle réduit le bruit via l'application d'un flou gaussien, ensuite elle effectue un seuillage pour séparer le premier plan et l'arrière plan et des opérations de nettoyage et s'adapte encore au niveau des couleurs.

On espère ici que ces changements permettent d'augmenter la précision du modèle grâce à un meilleur contraste, séparant bien les nombres des parties inutiles, et une réduction du bruit, tout en maintenant un traitement compatible avec un modèle conçu pour des images en couleur. Un exemple d'image pré-traitée :

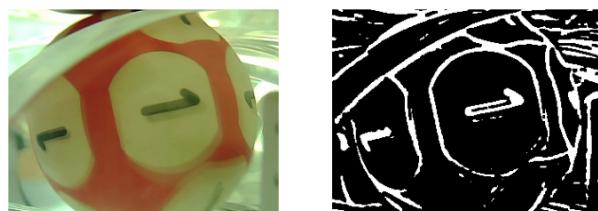
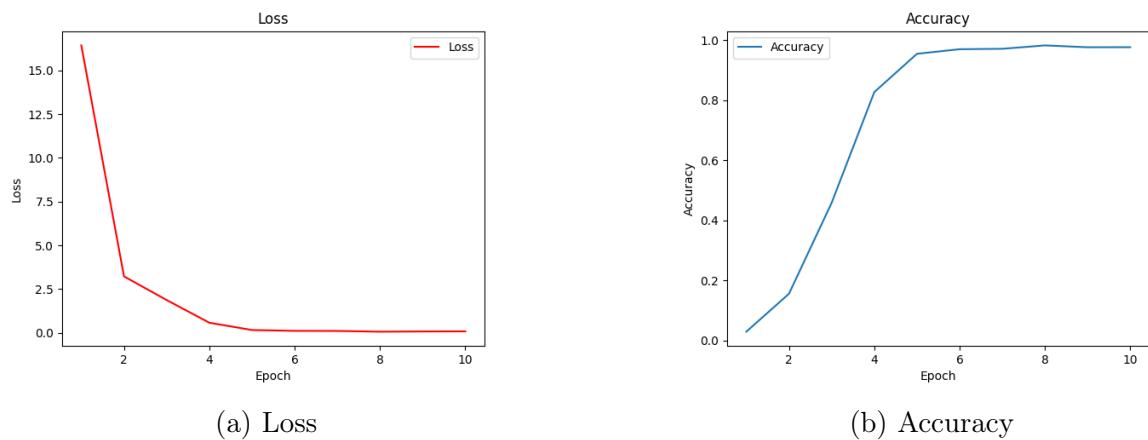


FIGURE 5.9 – Image normale et sa version prétraitée



On peut donc s'attendre à de bons résultats au vu des courbes.



FIGURE 5.11 – Test 1



FIGURE 5.12 – Test 2

Malheureusement, les résultats que nous obtenons ne sont pas ceux auxquels nous nous attendions. En effet, malgré tous ces pré-traitements, aucune image n'est reconnue à une exception près. Cependant, les prédictions ne sont pas toutes éloignées de la classe réelle. Par exemple, 44 a été reconnu en 42, 33 en 3, ce qui n'est pas aberrant.

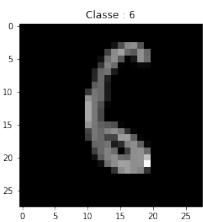
Chapitre 6

Images rognées

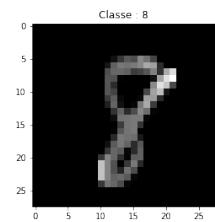
Puisque les différentes approches sur les images brutes ne nous ont pas menés à des résultats satisfaisants, nous avons décidé de procéder autrement et de désormais travailler sur des images rognées, ne laissant apparaître uniquement le nombre inscrit sur la boule de Loto.

6.1 MNIST

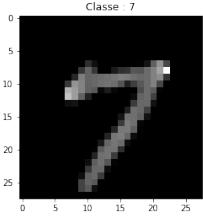
Cette idée nous est venue en découvrant l'existence de la base de données MNIST, cette base de données étant très utilisée dans l'apprentissage automatisé et le machine learning.



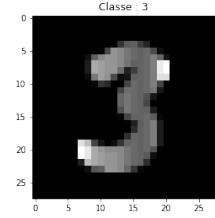
(a) Un 6 sur la base de données MNIST



(b) Un 8 sur la base de données MNIST



(c) Un 7 sur la base de données MNIST



(d) Un 3 sur la base de données MNIST

FIGURE 6.1 – Numéros écrits à la main dans la base de données MNIST

Cette base de donnée offre une multitude d'images de taille 28×28 écrit à la main. Elle est capable de reconnaître les chiffres de 1 à 9, comme le montre les résultats suivants :

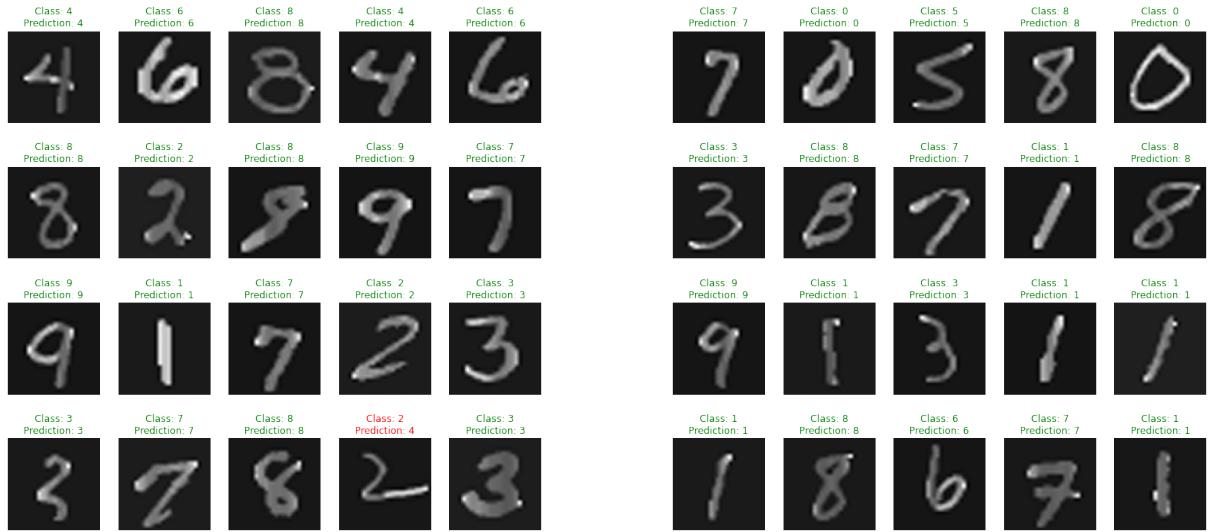


FIGURE 6.2 – Test 1

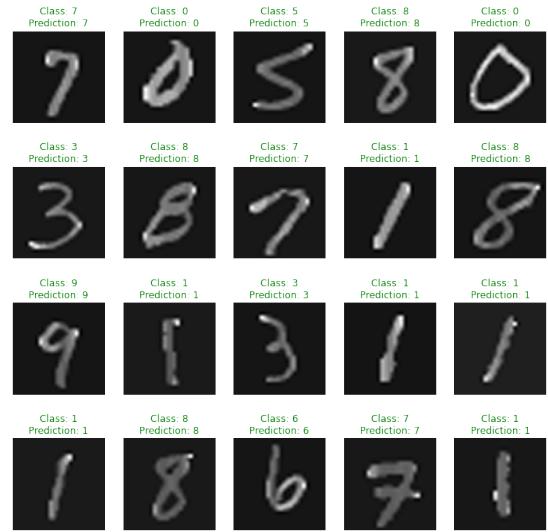


FIGURE 6.3 – Test 2



FIGURE 6.4 – Test 3

Avec ces résultats très corrects suite à l'entraînement sur MNIST et test sur MNIST, nous avons pensé à lancer les tests sur nos images du Loto. Cependant, un problème se pose ici, les images fournies dans la base MNIST ne contiennent que des chiffres de 0 à 9, et sont des images de taille 28×28 , tandis que nos images de loto ont des nombres compris entre 1 et 50, et de taille 240×320 . Les réseaux de neurones demandant des images de même taille, nous avons fait le choix de rogner des images à la main pour nos images de loto, de taille 70×70 , et nous avons agrandi les images MNIST par interpolation cubique, la librairie **opencv-python** est très bien pour ça :

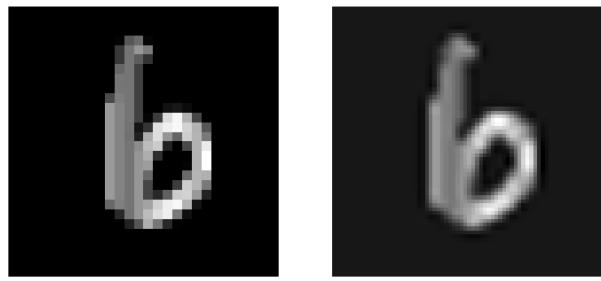


FIGURE 6.5 – Un 6 en 28×28 sur MNIST, et sa version agrandie en 70×70

Voici donc quelques résultats :



FIGURE 6.6 – Test 1



FIGURE 6.7 – Test 2

Toutefois, le succès des résultats obtenus ne correspond pas du tout à celui sur la base de données MNIST, bien que les images soient rognées et allant de 1 à 9 comme cette dernière.

6.2 Problème des images rognées

Un léger soucis se pose cependant sur les (254) images que nous avons rognées à la main (environ 5 par nombre de 1 à 50), c'est que 5 images par nombre est très loin d'être suffisant pour faire de l'apprentissage efficace. Pour remédier à cela, tensorflow nous fournit une méthode pour augmenter artificiellement notre base de données en appliquant des modifications sur nos images.

```
datagen = ImageDataGenerator(  
    rotation_range=180,          # Rotate the image  
    width_shift_range=0.1,       # Shift the image horizontally  
    height_shift_range=0.1,      # Shift the image vertically  
    zoom_range=0.1,             # Zoom in or out on the image  
    horizontal_flip=False,      # Flip the image horizontally  
    vertical_flip=False,        # Do not flip the image vertically  
    fill_mode='nearest'         # Fill in missing pixels with the nearest value  
)
```

Listing 6.1 – Générateur d'images

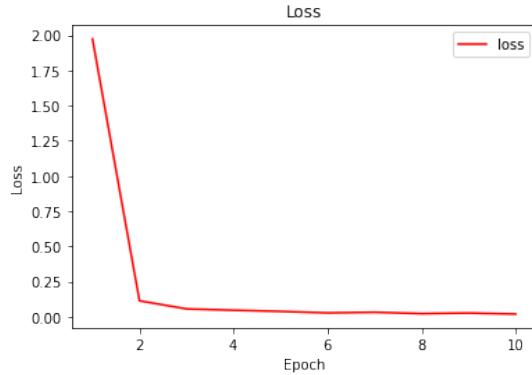
Ainsi, le code ci-dessus permet d'appliquer sur des images des rotations entre 1° et 180° , des décalages sur l'axe horizontal / vertical, ou encore des zooms. Un exemple :



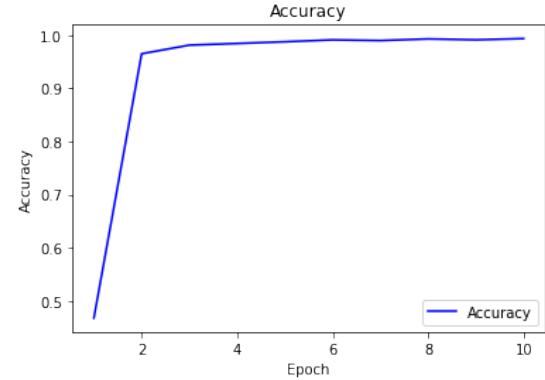
FIGURE 6.8 – Une image rognée par nous, et une image générée.

6.3 Images en couleur

Maintenant que nous avons notre base de donnée augmentée, nous pouvons lancer l'apprentissage d'un nouveau réseau de neurones, toujours sur 10 époques d'apprentissage.



(a) Loss



(b) Accuracy

Nous avons une très faible loss et une forte accuracy qui correspond aux résultats visuels :



FIGURE 6.10 – Test 1



FIGURE 6.11 – Test 2

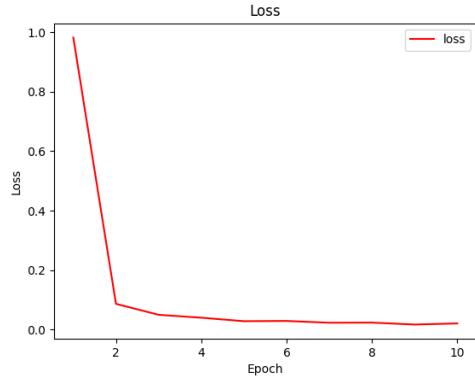


FIGURE 6.12 – Test 3

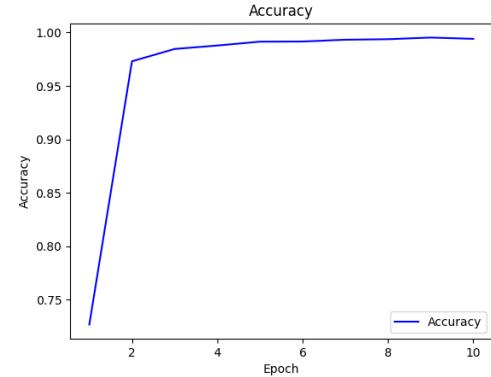
Toutes les images de nos tests ont été reconnues.

6.4 Images en noir et blanc

Nous avons ensuite relancé ce même test avec les images en noir et blanc.



(a) Loss



(b) Accuracy

Comme plus haut, nous avons une faible loss et forte accuracy, avec des résultats visuels correspondants :



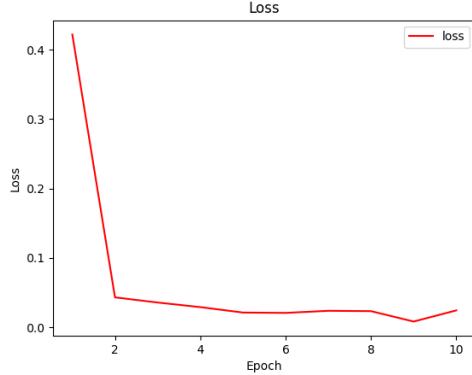
FIGURE 6.14 – Test 1



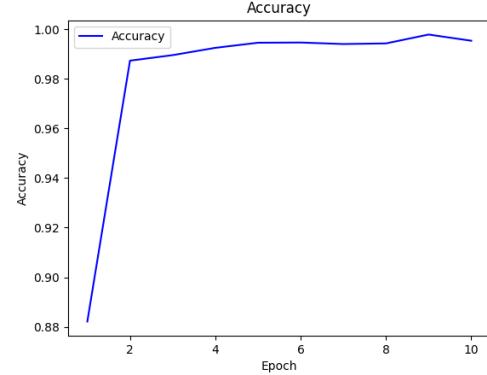
FIGURE 6.15 – Test 2

6.5 Images preprocessed

Enfin, nous avons fini par tester notre code sur les images passant par le pré-traitement de la fonction *preprocessed*.



(a) Loss



(b) Accuracy

Là encore la loss est faible et l'accuracy très élevée. Nous obtenons visuellement du 100% de bonne interprétation.



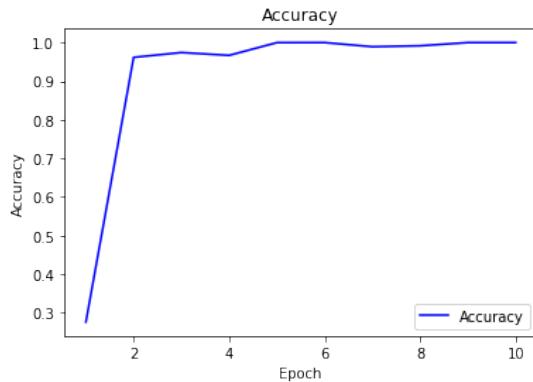
FIGURE 6.17 – Test 1



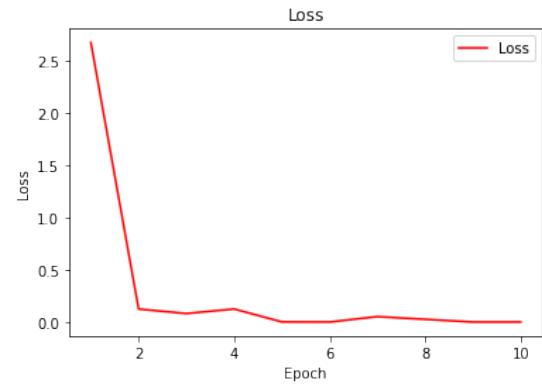
FIGURE 6.18 – Test 2

Les 3 méthodes semblent donner de très bons résultats ici (sur les images en couleurs, en niveaux de gris, et les images pré-traitées), mais on s'est rendus compte par la suite que les images se ressemblent parfois énormément, en témoigne les 2 images 46 de la Figure 6.17, les modifications apportées possibles étant peu nombreuses. On a alors refait un dernier test, sur les images en couleur de nouveau, et en autorisant les images à avoir cette fois ci des rotations allant jusqu'à 360°.

6.6 Images en couleurs, et rotation à 360°



(a) Loss



(b) Accuracy

Avec à nouveau, une très bonne accuracy très proche des 100%. Et sur des tests visuels :

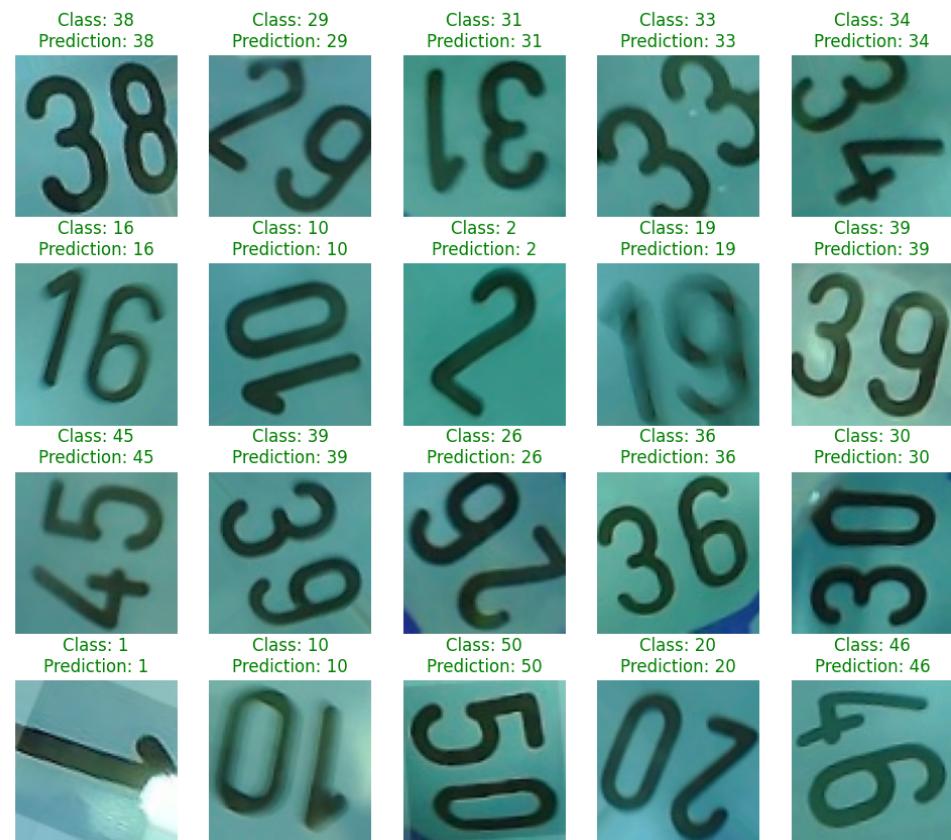


FIGURE 6.20 – Test 1



FIGURE 6.21 – Test 2

Chapitre 7

Conclusion

Ce projet nous a permis de découvrir ce qu'était un réseau de neurones/réseau de neurones convolutionnel, comment ça marche, l'apprentissage sur les images.

Nous avons d'abord testé notre code sur les images brutes fournies, mais nous nous sommes aperçus que ce n'était pas tellement productif.

Il nous a également permis d'essayer des fonctions de pré-traitement d'images sur différents réseaux de neurones, et leurs efficacités.

Après avoir découvert la base de données MNIST et son fonctionnement sur des images de 1 à 9, nous avons eu l'idée de réitérer nos tests sur nos images en ne considérant que les nombres inscrits sur les boules. C'est pourquoi nous avons rogné à la main quelques images et nous avons utilisé une fonction pour en générer aléatoirement afin d'en avoir assez pour la phase d'entraînement et de tests.

Ce processus a porté ses fruits et nous a permis d'obtenir des résultats pratiquement totalement satisfaisants.

Cependant, les bons résultats ayant été obtenus sur les images rognées, nous pourrions améliorer notre code en ajoutant une fonction permettant de rogner les images automatiquement, et ce manière efficiente. Pour ce faire, il serait également nécessaire de prendre en compte que l'image à traiter ait un nombre centré afin de sélectionner une zone par défaut à extraire.

Annexe A

Cahier des charges

Rappel du cahier des charges.

Annexe B

Manuel utilisateur

Le programme se lance de la manière suivante :

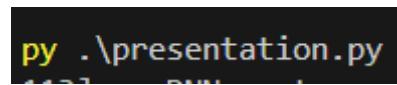


FIGURE B.1 – Lancer le programme

Une fenêtre apparaît alors, il faut par la suite cliquer sur **File** en haut à gauche, puis sur **Open**.

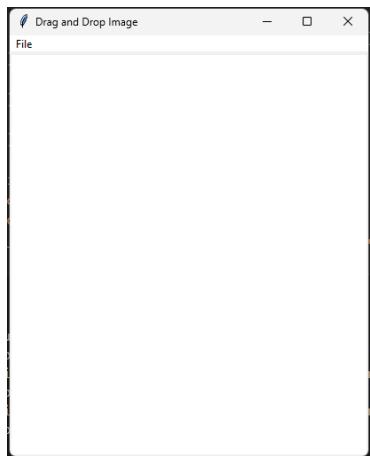


FIGURE B.2 – Fenêtre



FIGURE B.3 – File

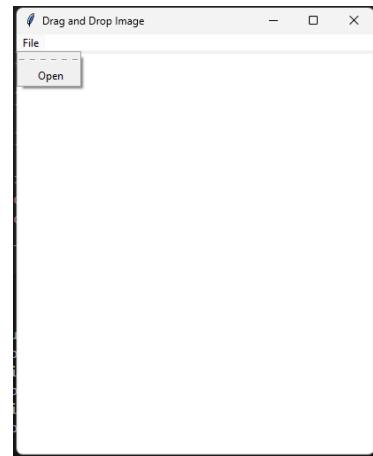


FIGURE B.4 – Open

L'explorateur de fichier s'ouvre alors, il faut alors choisir une photo dont on veut prédire le résultat.

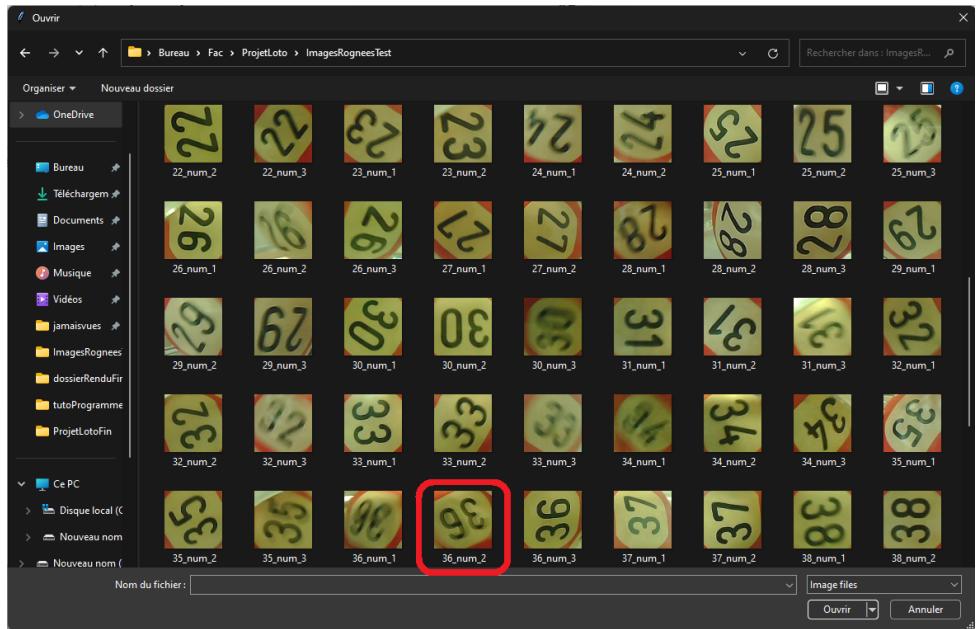


FIGURE B.5 – Choix de la photo à prédire.

Le programme prend alors la photo, l'affiche et en fait une prédiction.



FIGURE B.6 – Bonne prédiction.



FIGURE B.7 – Bonne prédiction.

Attention cependant, si la photo est trop floue, il se peut que les prédictions ne soient pas parfaites !

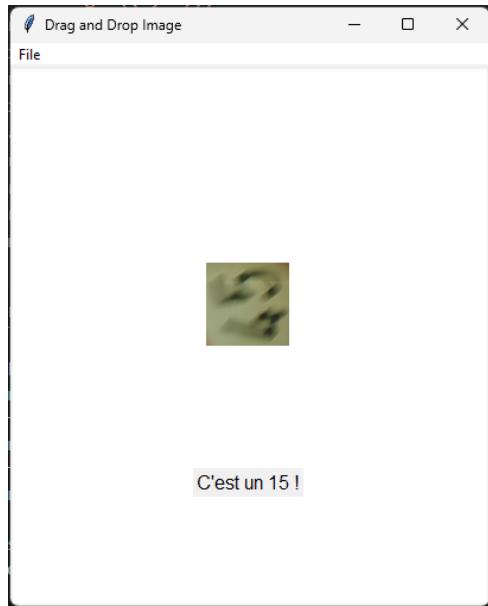


FIGURE B.8 – Mauvaise prédiction, image floue.

Ou pour le 6 et le 9, si le point (dans le cas du 6), ou la petite barre (dans le cas du 9) n'est pas détectée, il peut y avoir une prédiction inverse.

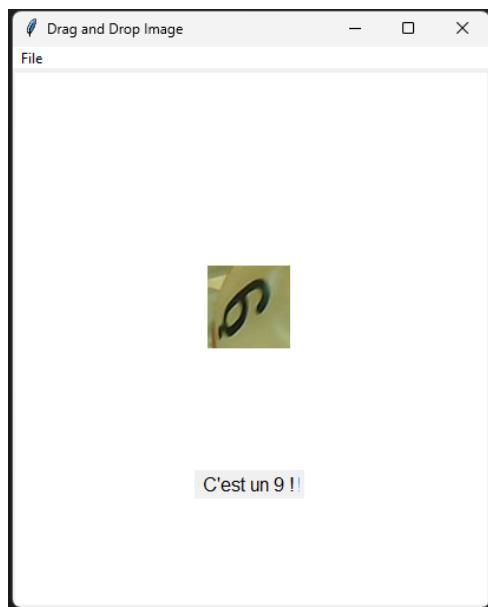


FIGURE B.9 – Mauvaise prédiction, 6 confondu avec 9.

Annexe C

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/all_symbols

[www.youtube.com/watch ?v=bte8Er0QhDg](http://www.youtube.com/watch?v=bte8Er0QhDg)