



DETECTION DE VISAGES

Python - Keras – Tensorflow – Google Colab

Math Hugo | Collard Alexandre | 03/04/2020

<https://github.com/Mathugo/PiSecureHome>

https://github.com/Mathugo/Keras_CNN

https://github.com/Mathugo/Face_CNN

Sommaire

1.	Choix des technologies et du matériel utilisé	3
1.1	Software et matériel utilisés.....	3
2.	Objectifs du programme.....	4
2.1	Objectif général.....	4
2.2	Problématiques.....	4
3.	Création du dataset	5
3.1	Hugo	5
3.2	Inconu.....	6
3.3	Lancement du programme de création de dataset	6
3.4	Upload du dataset vers Google Drive.....	7
3.4.1	Problématiques	7
3.4.2	Upload asynchrone du dataset vers google drive.....	7
4.	Réseau de neurones convolutifs	8
4.1	Lecture asynchrone des images.....	8
4.2	Transformation des images.....	8
4.3	Data augmentation	8
4.4	Structure	9
4.5	Recherche des hyperparamètres par quadrillage	11
4.5.1	GridSearch ou RandomSearch	11
4.6	Entrainement.....	12
5.	Déploiement sur Raspberry PI.....	14
5.1	Librairies.....	14
5.2	Conversion du modèle.....	14
5.3	Déploiement.....	14
5.4	Résultats.....	15
5.5	Pistes d'amélioration	16

1. Choix des technologies et du matériel utilisé

1.1 Software et matériel utilisés

Nous avons choisi d'utiliser *Google Colab* et *Visual Studio Code* comme environnement de développement pour le Deep learning. *Google Colab* embarque *Python3.7*, *Keras*, *Tensorflow* et beaucoup d'autres outils nécessaires à la data science. Il permet d'entraîner efficacement des réseaux de neurones. Tout le code est expliqué sur Colab ainsi que les bibliothèques utilisées.

Lien du **Drive** : <https://drive.google.com/drive/folders/iYSkURYXmu-ULBaHuyMXpWtiuS7jcoAT?usp=sharing>

Tout le code est documenté et commenté dans le Google Colab, on ne reviendra pas sur le code dans le compte rendu.

Le programme final tournera sur une raspberry pi 4 doté d'une caméra 1.3mp filmant en 1080p.

2. Objectifs du programme

2.1 Objectif général

L'objectif du programme est de pouvoir détecter le visage d'une personne connu à l'aide d'un réseaux de neurones convolutifs pré-entraîné. Il y aura donc deux classes à savoir : Hugo et Inconnu. Le modèle pourra être déployé sur une **raspberry-pi 4** . L'idée serait de s'en servir pour créer une caméra de sécurité intelligente alertant l'occupant d'une maison qu'une personne inconnu se tient devant sa porte (avec des enceintes : **Google Home Mini**) .



Figure 1 Schéma de notre projet avec détection de visage dans notre cas.

2.2 Problématiques

Le programme devra être performant puisque la pi 4 devra analyser en continue les sorties vidéo de la caméra. Il faudra jouer entre fiabilité de détection et rapidité tout en trouvant des astuces d'optimisation (**TensorFlow Lite** etc..). Il faudra aussi trouver le bon modèle en jouant sur les hyperparamètres (epochs, learning rate, batch size, ...), la taille de l'image, les layers, la profondeur, la complexité du réseau de neurones et surtout le dataset.

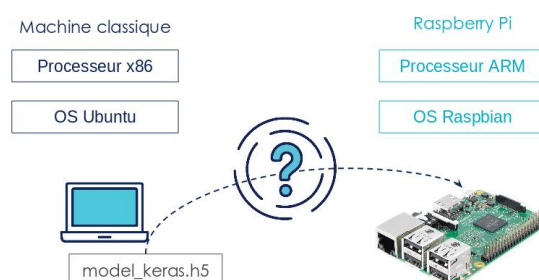


Figure 2 Entrainement sur machine puis déploiement sur pi

3. Création du dataset

https://github.com/Mathugo/Face_CNN

Afin d'obtenir un modèle fiable, il nous faudra beaucoup d'images différentes des deux classes (Hugo et Inconnu). L'objectif serait d'en avoir environ 30000 de chaque.

3.1 Hugo

Pour obtenir le visage d'une personne nous avons utilisé la librairie **MTCNN** de python qui affiche la meilleure fiabilité pour détecter des visages. Nous avons donc enregistré le visage de la personne Hugo avec une Webcam d'ordinateur en mode vidéo pendant environ 10min au total dans des environnements différents avec un éclairage et des postures différentes.

Ensuite, il nous a fallu analyser avec MTCNN frame par frame les vidéos pour en sortir le visage correspondant et les stocker dans le fichier <dataset\hugo>. Pour gagner de la mémoire nous avons modifier la dimensions des visages en 600x400. (17279 visages)

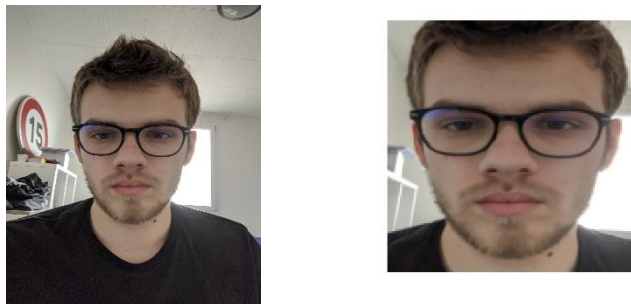
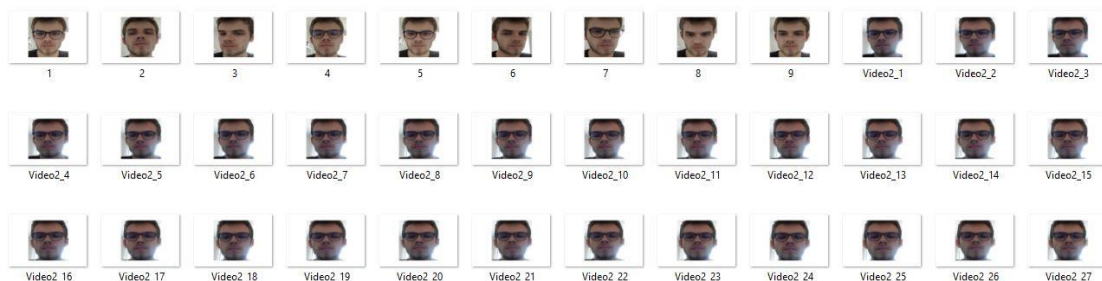
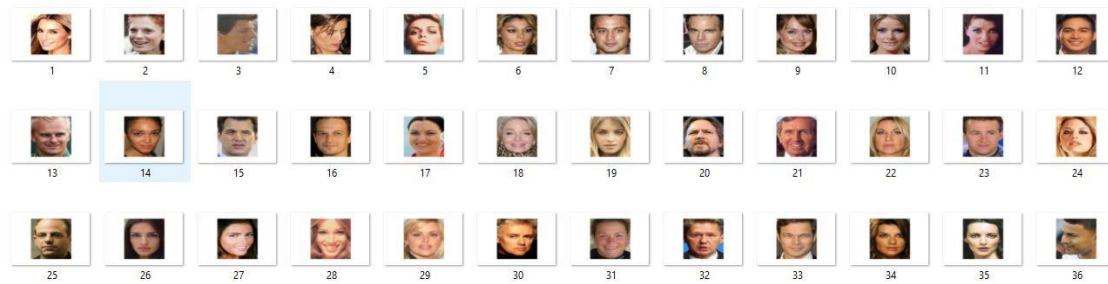


Figure 3 Image avant et après extraction du visage



3.2 Inconnu

Pour obtenir le visage d'un inconnu, nous avons téléchargé une base de donnée de visages de célébrités (1.4gb) (pris le quart car trop gros sinon) et ensuite on a réitéré le même principe à savoir détecter le visage de la photo correspondante à l'aide de la méthode MTCNN. On enregistre tout cela dans le dossier : <dataset\not_hugo> (17704 visages)



3.3 Lancement du programme de création de dataset

```
PS E:\ESIREM\Python\Face_CNN> python .\main.py
usage: main.py [-h] -d DATASET -o OUTPUT
main.py: error: the following arguments are required: -d/--dataset, -o/--output
PS E:\ESIREM\Python\Face_CNN> |
```

Dans notre cas on utilise les arguments : `python main.py -d dataset -o output`

Le programme va rechercher des vidéos et images dans les dossiers enfants de dataset représentant les classes (hugo, not_hugo), pour pouvoir ensuite extraire les visages de celles-ci vers le dossier output.

3.4 Upload du dataset vers Google Drive

3.4.1 Problématiques

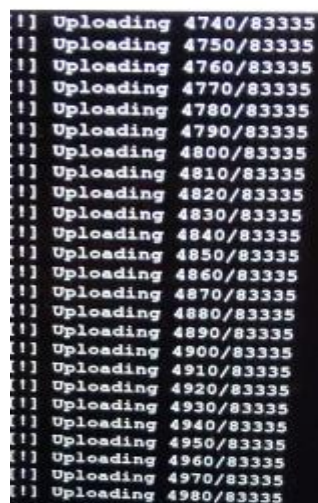
Dans l'objectif d'utiliser Google Colab il faut que les images transitent rapidement pour pouvoir les lire (60k images à charger). Google Colab permet d'upload directement un dossier dans l'environnement d'exécution mais c'est beaucoup trop long (environ 2h).

Google Storage service de google, permet de stocker de large dataset mais encore une fois celui-ci n'est pas assez rapide et il n'y a pas d'option pour lire directement et efficacement du contenu de google storage et de le récupérer vers Colab.

Notre choix se tourne vers Google Drive qui avec la bibliothèque python pydrive permet d'upload des fichiers et d'ensuite pouvoir monter google drive dans colab avec un temps réduit (environ 20-30min).

3.4.2 Upload asynchrone du dataset vers google drive

Le fichier **python_drive.py** permet d'upload dans un dossier drive en connaissant son ID des images. En exécutant une instruction à la suite l'upload est beaucoup trop long (1 fichier toute les 0.5 secondes). On utilisera donc une programmation asynchrone pour pouvoir réduire le temps d'upload. On fixera le nombre de threads à 10 dû aux limites de requêtes de l'API Google Drive. Après 8h on arrive à upload 30 000 images. Au final on aura 27 000 images dans le dossier **hugo** et 30 000 dans **not_hugo**



```
[1] Uploading 4740/83335
[1] Uploading 4750/83335
[1] Uploading 4760/83335
[1] Uploading 4770/83335
[1] Uploading 4780/83335
[1] Uploading 4790/83335
[1] Uploading 4800/83335
[1] Uploading 4810/83335
[1] Uploading 4820/83335
[1] Uploading 4830/83335
[1] Uploading 4840/83335
[1] Uploading 4850/83335
[1] Uploading 4860/83335
[1] Uploading 4870/83335
[1] Uploading 4880/83335
[1] Uploading 4890/83335
[1] Uploading 4900/83335
[1] Uploading 4910/83335
[1] Uploading 4920/83335
[1] Uploading 4930/83335
[1] Uploading 4940/83335
[1] Uploading 4950/83335
[1] Uploading 4960/83335
[1] Uploading 4970/83335
[1] Uploading 4980/83335
```

4. Réseau de neurones convolutifs

https://github.com/Mathugo/Keras_CNN

4.1 Lecture asynchrone des images

Depuis Google Colab on va monter Google Drive pour pouvoir lire le dataset. Mais là encore les images prennent beaucoup trop de temps à se charger. A l'aide de la classe **Read_Image(Thread)** dans le fichier **model.py** on va pouvoir charger par paquets de 10 les images en utilisant là aussi 10 Threads.

4.2 Transformation des images

Durant ce processus, les images sont redimensionner en 224x224, et les labels sont sous forme de booléens. 1 si Hugo et 0 sinon. Il va falloir ensuite transformer et « scaler » ces données pour pouvoir nourrir correctement le réseau de neurones. On utilise numpy et ses tableaux pour ceci. On split avec un ratio de 0.25 les données d'entrainements et le reste à savoir les données de test (avec lequel le modèle va s'entraîner et tester le résultat).

4.3 Data augmentation

Pour pouvoir avoir plus de données et une meilleure généralisation du modèle aux futurs données, on utilise la classe **ImageDataGenerator**. On génère des rotations (0.2 facteur), des modifications de largeurs et hauteur (width_shift_range=0.2, height_shift_range=0.2) ainsi que des zooms.

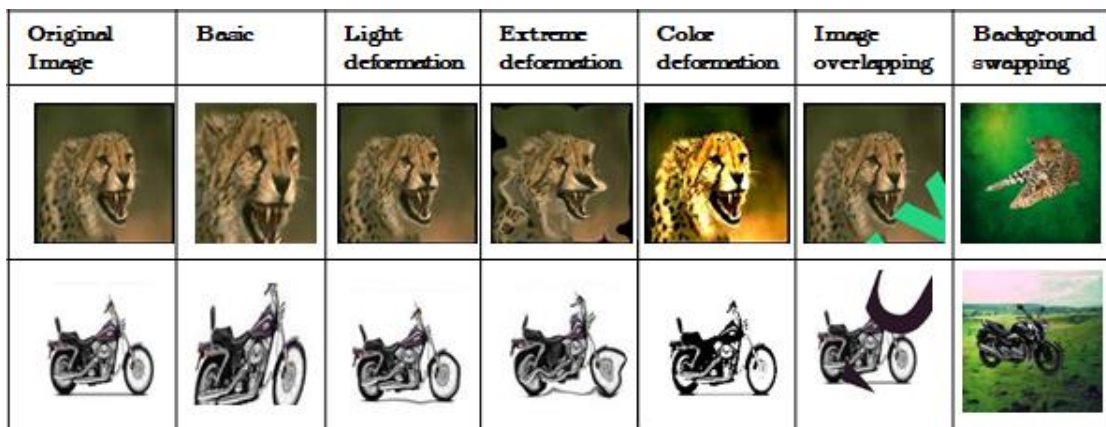


Figure 4 Représentation graphique de l'augmentation d'image

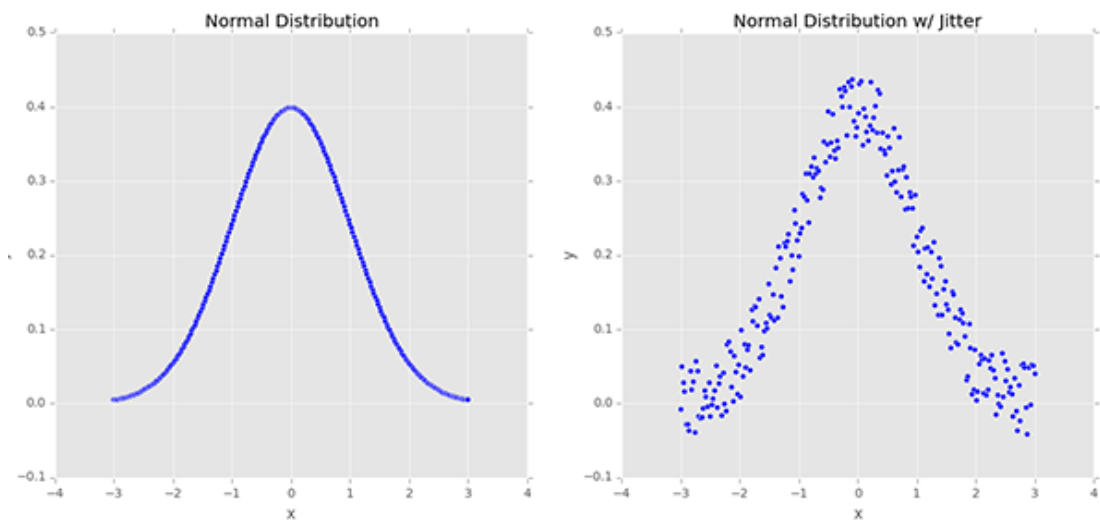


Figure 5 Distribution normal mais en rajoutant des points (des aléas, du bruit)

4.4 Structure

On a repris la structure du réseau de neurones convolutif VGG16 en diminuant les dimensions pour réduire le coût et le temps d'entraînement.

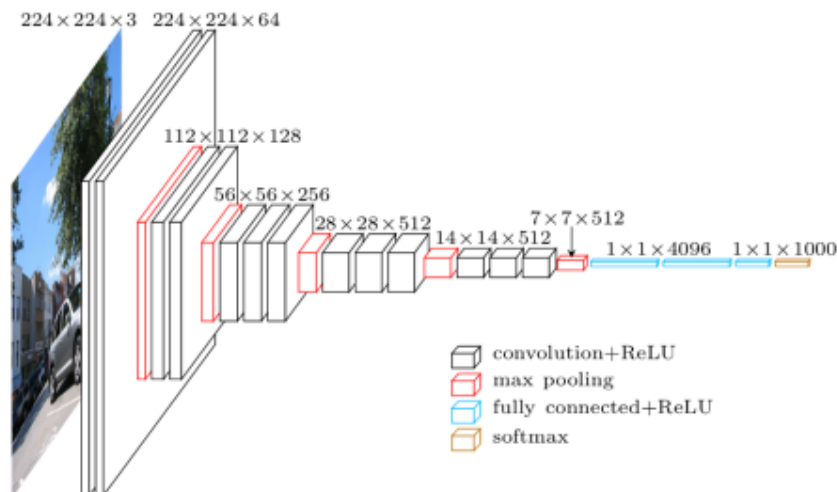


Figure 6 Structure du réseau de neurone VGG16

En effet la particularité de ce CNN c'est son nombre de paramètres. Avec une image de 224x224 on arrive à 134 millions de paramètres qui vont ensuite être trouvés à l'aide d'un algorithme d'optimisation (Adam, Stochastic gradient descent ..).

Grâce au site <https://lutzroeder.github.io/netron/> on peut visualiser notre modèle

Après mis à jour, dans notre cas notre réseau donne ceci :

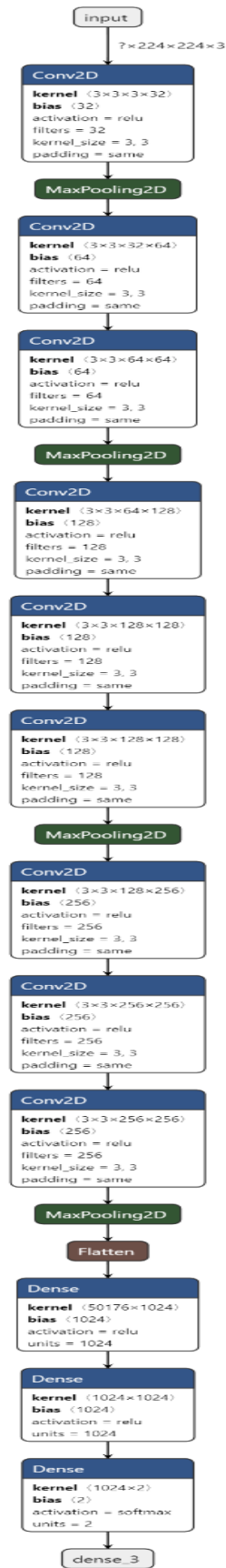


Figure 7 Notre réseau de neurone

4.5 Recherche des hyperparamètres par quadrillage

Pour avoir la meilleur précision possible, on peut utiliser les recherches par quadrillage de Keras : **GridSearchCV**, **RandomizedSearchCv**

Ces algorithmes marchent en définissant une intervalle bien choisi de recherche pour les hyperparamètres (**epochs**, **learning rate**, **batch size**)

Pour limiter le nombre d'itérations on utilisera des intervalles assez petite (manque de puissance de calcul). Tel que :

- Batch Size : $\{8, 16, \dots, 128\} \Rightarrow 2^k$ où k appartient $\in \mathbb{N}$ et $2 < k < 7$
- Learning rate : $\{10^{-4}, \dots, 10^{-1}\} \Rightarrow \log_{10} b = c$ où $c \in \mathbb{Z}$ et $c \in \{-4, \dots, -1\}$
- Epochs : $\{10, 50, 75\}$

4.5.1 GridSearch ou RandomSearch

Selon un papier de recherche de l'Université de Montréal, la recherche par quadrillage classique semble inefficace.

Source : <http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf>

En effet, les bons paramètres seront trouvés mais avec un coût très élevé et avec des dimensions très élevés, cela risque d'être compliqué. Alors qu'avec **Random Search**, on aura de meilleur résultats en moins d'itérations ce qui est crucial dans notre cas.

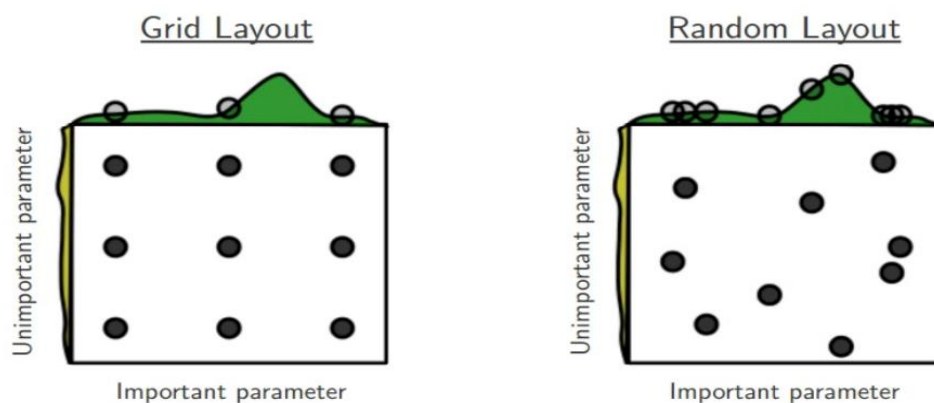


Image from <http://jmlr.csail.mit.edu/papers/volume13/bergstra12a/bergstra12a.pdf>

La différence réside dans les points choisi au début. Random Search les choisi aléatoirement. A la fin, on a « exploré l'espace » plus largement qu'avec un quadrillage classique donc exploré plus de possibilités en moins d'itérations.

On lance sur *Google Colab* la commande : `!python3 '/content/gdrive/My Drive/Colab/Notebooks/pi-secure-home/python_code/app.py' -rg 1 -n 16000`

Qui va lister les meilleurs hyperparamètres à considérer en chargeant 16 000 images.

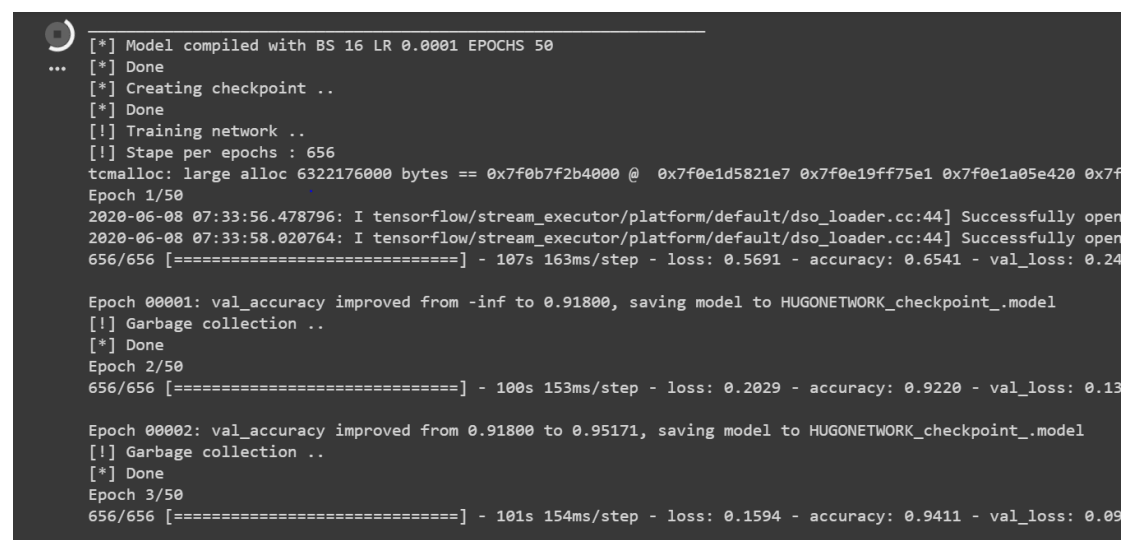
On obtient : bs = 16, Epochs = 100, LR=0.0001 avec N= 14 000

Avec Google Colab, en réduisant le format des variables (float16) et en utilisant des batch size très petit (8, 16, 32) on arrive à limiter l'usage de la RAM. Mais même avec ces astuces, on arrive qu'à charger 14 000 images sur les 55 000 qu'on a.

4.6 Entraînement

Après avoir eu les bons hyperparamètres, on se décide d'entraîner le modèle sur 14 000 images avec un batch size de 16, 50 Epochs et un learning rate de 10^{-4}

On pourrait réduire le batch size à 8 pour limiter l'usage de la ram et utiliser 500-1000 images de plus.



```
[*] Model compiled with BS 16 LR 0.0001 EPOCHS 50
...
[*] Done
[*] Creating checkpoint ..
[*] Done
[!] Training network ..
[!] Stape per epochs : 656
tcmmalloc: large alloc 6322176000 bytes == 0x7f0b7f2b4000 @ 0x7f0e1d5821e7 0x7f0e19ff75e1 0x7f0e1a05e420 0x7f
Epoch 1/50
2020-06-08 07:33:56.478796: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully open
2020-06-08 07:33:58.020764: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully open
656/656 [=====] - 107s 163ms/step - loss: 0.5691 - accuracy: 0.6541 - val_loss: 0.24

Epoch 00001: val_accuracy improved from -inf to 0.91800, saving model to HUGONETWORK_checkpoint_.model
[!] Garbage collection ..
[*] Done
Epoch 2/50
656/656 [=====] - 100s 153ms/step - loss: 0.2029 - accuracy: 0.9220 - val_loss: 0.13

Epoch 00002: val_accuracy improved from 0.91800 to 0.95171, saving model to HUGONETWORK_checkpoint_.model
[!] Garbage collection ..
[*] Done
Epoch 3/50
656/656 [=====] - 101s 154ms/step - loss: 0.1594 - accuracy: 0.9411 - val_loss: 0.09
```

Figure 8 Extrait d'un entraînement avec ces paramètres.

Note : l'accuracy augmente très rapidement, au bout de seulement 3 epochs on atteint 0.95 accuracy. Cela veut dire que soit notre modèle répond très bien aux données soit le modèle ajuste trop et s'adapte bien aux données d'entraînements mais ne généralise pas. On verra cela avec les tests sur raspberry pi et les tests de validations.

En effet ce n'est pas normal d'avoir un modèle avec 0.91 d'accuracy au premier epochs.

Notre learning rate n'est donc pas bon. Cela provient de notre intervalle dans la **RandomSearch**. Notre LR évoluait de 0.1 à 0.001 or il pouvait très bien être à 10^{-7}

On décide donc de lancer une nouvelle recherche d'hyperparamètres. A la suite de plusieurs dizaine d'heure de training on obtient un LR de 10^{-7} . On réentraîne notre réseaux de neurones avec cette nouvelle valeur

```

[*] Model compiled with BS 16 LR 1e-07 EPOCHS 150
[*] Creating checkpoint ..
[*] Done
[!] Training network ..
[!] Stage per epochs : 632
tcmalloc: large alloc 6096388096 bytes == 0x7fd0c2a08000 @ 0x7fd354b451e7 0x7fd3515ba5e1 0x7fd351621420 0x7fd351621682 0x7fd351621b3e 0x7fd351621
Epoch 1/150
2020-06-11 11:58:44.290640: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library libcublas.so.10
2020-06-11 11:58:45.807464: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library libcudnn.so.7
632/632 [=====] - 117s 185ms/step - loss: 0.6922 - accuracy: 0.5017 - val_loss: 0.6902 - val_accuracy: 0.5384

Epoch 00001: val_accuracy improved from -inf to 0.53837, saving model to HUGONETWORK_checkpoint_.model
[!] Garbage collection ..
[*] Done
Epoch 2/150
632/632 [=====] - 110s 174ms/step - loss: 0.6909 - accuracy: 0.5613 - val_loss: 0.6886 - val_accuracy: 0.5576

Epoch 00002: val_accuracy improved from 0.53837 to 0.55763, saving model to HUGONETWORK_checkpoint_.model
[!] Garbage collection ..
[*] Done
Epoch 3/150
632/632 [=====] - 110s 174ms/step - loss: 0.6895 - accuracy: 0.5618 - val_loss: 0.6867 - val_accuracy: 0.6335

```

Figure 9 Training avec un LR de 10^{-7}

On remarque tout de suite que l'accuracy augmente « plus lentement » ce qui est normal.

A la fin du training on obtient une accuracy de 0.915

```

Epoch 00149: val_accuracy did not improve from 0.91526
[!] Garbage collection ..
[*] Done
Epoch 150/150
632/632 [=====] - 108s 171ms/step - loss: 0.2761 - accuracy: 0.8884 - val_loss: 0.24

```

Figure 10 Fin de Training



Figure 11 Evolution de la fonction d'erreur, accuracy au cours du temps (epochs)

5. Déploiement sur Raspberry PI

<https://github.com/Mathugo/PiSecureHome>

5.1 Librairies

Nous avons besoin d'installer *Tensorflow lite* pour gagner un maximum d'fps et alléger les calculs. On exécute la commande : **pip install tensorflow**

Et ensuite dans le fichier python on aura juste à ajouter au début :

```
from tf.lite_runtime.interpreter import Interpreter
```

5.2 Conversion du modèle

https://www.tensorflow.org/lite/convert/python_api

On a converti notre modèle construit avec Keras en modèle Tensorflow lite pour augmenter les performances.

```
import tensorflow as tf

def convert_to_lite(filename, output):

    model=tf.keras.models.load_model(filename, compile=False)
    # !pip install tf-nightly
    converter = tf.lite.TFLiteConverter.from_keras_model(model)
    tflite_model = converter.convert()
    with tf.io.gfile.GFile(output, 'wb') as f:
        f.write(tflite_model)
```

extrait du fichier **convert_to_lite.py** dans le répo KERAS_CNN

5.3 Déploiement

On exécute le programme python en ligne de commande depuis la Pi 4 en ssh.

```
raspberrypi:~/Bookshelf/Prog/PiSecureHome/pi4 $ python3 app.py -c data/haarcascade_frontalface_default.xml -m data/HUGONETWORK
4_224_75_1e-06_16_14000.tflite -d yes
```

Figure 2 Lancement du programme de détection sur la Pi

Nous avons 3 arguments dont 2 requis qui sont l'emplacement du fichier cascade d'opencv pour les visages et le modèle de deep learning tensorflow lite pour reconnaître les visages

5.4 Résultats

5.4.1 Fluidité et performance

On obtient de bons résultats avec une fluidité assez bonne. Pour le modèle Tensorflow lite on a environ 0.56 secondes d'exécution pour classer l'image, ce qui est très bon pour une raspberry pi.

En comparaison, on avait essayé de reconnaître des visages avec **face_recognition** en python mais les fps étaient très bas (1 images toutes les 4 secondes). Ici dès qu'opencv repère un visage, on a une chute d'fps et on arrive à 2 -3 fps en détection. En faisant notre propre réseau de neurones le temps de détection à été réduit par 10.

5.4.2 Fiabilité et précision

Du coté de la fiabilité de la détection, on remarque que la diminution de la luminosité ainsi que la présence de contraste, contre jour entraine une chute de la fiabilité. C'est à cause du pauvre nombres d'images qu'on a pu charger dans notre environnement Colab (14 000). Pour la détection d'images, on a besoin de plusieurs dizaines de milliers d'images par classe. Ici on en avait que 7000 par classe.

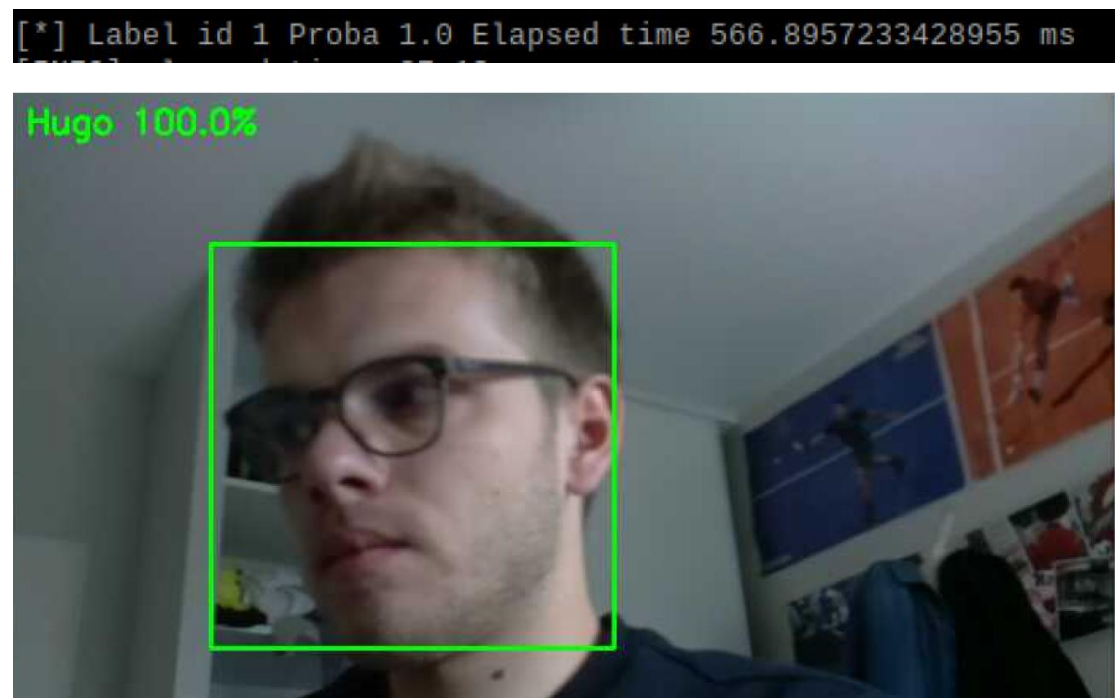


Figure 9 Extrait d'un flux vidéo de la Pi

5.5 Pistes d'amélioration

5.5.1 Fluidité et performance

Pour augmenter les performances de détection on aurait pu acheter un petit **TPU usb accelerator**. Cela aurait considérablement réduit le temps de détection du modèle tensorflow lite.



Figure 9 Petit TPU Coral pour de l'IA Embarqué

5.5.2 Fiabilité et précision

Si on avait disposé d'un pc avec beaucoup de ram (32gb minimum) et un puissant GPU, on aurait pu charger toutes nos images et augmenter grandement la fiabilité et précision de notre modèle.

On aurait pu aussi augmenter le nombre de classe, par exemple rajouter la classe Alexandre. Il nous aurait fallu changer 1 layer dans le réseau de neurones pour pouvoir retourner 3 probabilité correspondant aux classes. Le réseau VGG16 est capable de supporter plusieurs centaines de classes. La difficulté ici était d'avoir assez d'images.