ENSEIRB-MATMECA

TinyML on Raspberry Pi

ETUDIANTS :

ALABD Koutaiba

ARDOUIN Victor

DAVID Noé

RESPONSABLE :

Patrice Kadionik

Table des matières

1 Introduction

Nous allons présenter notre projet TinyML sur Raspberry Pi. L'objectif de ce projet est d'implémenter une intelligence artificielle (IA) sur un microcontrôleur (de type Raspberry Pi). Nous allons ainsi utiliser Tensorflow Lite qui est une API permettant de manipuler des réseaux de neurones en réduisant leur format.

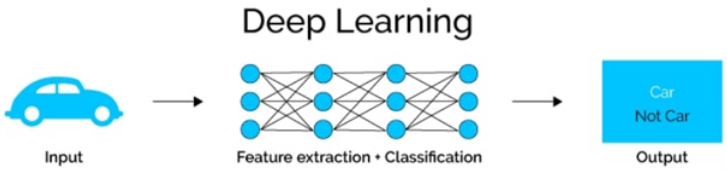
Dans ce contexte, notre objectif final est de créer une IA implémentée sur une Raspberry Pi 3 Modèle B, qui fait l'appel dans une classe grâce à de la reconnaissance faciale. Ceci étant possible grâce à la récolte préalable d'images de visage de tous les élèves, pour créer une base de donnée cohérente pour notre réseau de neurone.

2 Les principes de l'IA

L'intelligence artificielle (IA) est une vaste branche de l'informatique qui s'intéresse à la construction de machines intelligentes capables d'effectuer des tâches qui nécessitent généralement l'intelligence humaine. Le but principal de cette discipline est donc d'imiter l'intelligence humaine en construisant des réseaux de neurones qui ressemblent dans leur fonctionnement au cerveau humain. Les tâches que ces réseaux peuvent effectuer varient dans leur difficulté et complexité. Elles peuvent être simples, comme la détection de visages dans une image, mais aussi complexes comme trouver une équation particulière qui sépare un ensemble de données. Cependant, la construction des réseaux capables d'effectuer ce travail n'est pas simple ni trivial.

L'intelligence artificielle est un terme très large qui englobe plusieurs disciplines, mais celle la plus utilisée et la plus répandue est le Deep Learning. Dans celle-ci, des couches interconnectées de calculateur logicielles appelées «neurones» forment un réseau de neurones. L'idée est de reproduire une compréhension abstraite de la façon dont nous pensons que le cerveau humain pourrait traiter des informations similaires et apprendre de son environnement et de ses entrées sensorielles.

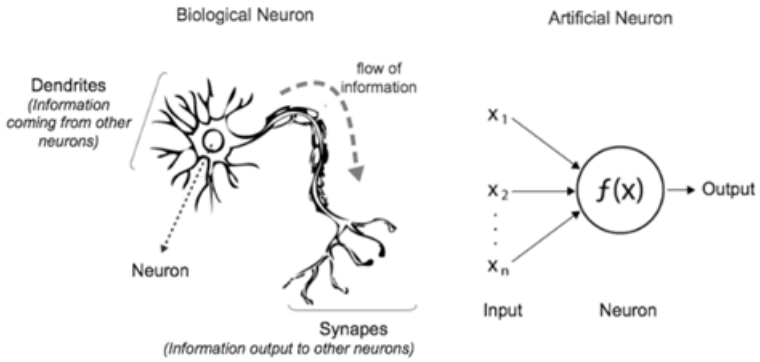
Le réseau ingère de grandes quantités de données d'entrée, les traitant à travers plusieurs couches de neurones qui apprennent des caractéristiques de plus en plus complexes des données à chaque couche :



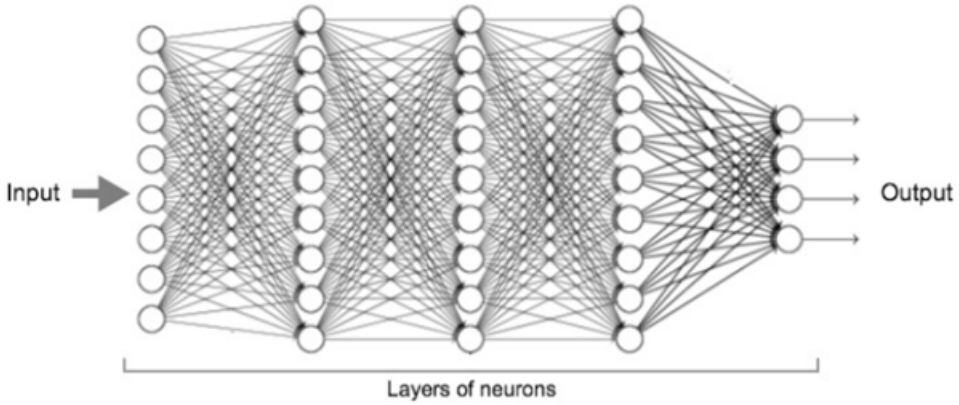
Pour vraiment comprendre les réseaux artificiels il faut apprendre les bases de l'intelligence artificielle en commençant par l'unité de base qui est le neurone. Dans la suite nous allons donc expliquer les briques de base de cette discipline et les principes utilisées pour construire des programmes intelligents.

2.1 Le neurone

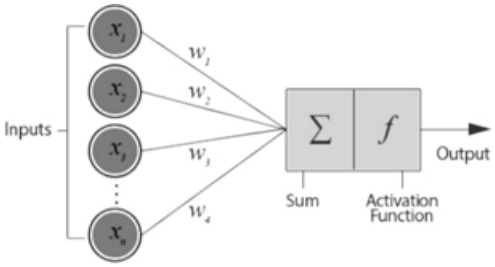
Pour imiter l'intelligence humaine et le cerveau (qui est la structure la plus complexe de l'univers), on fait l'analogie avec le neurone biologique qui a des entrées X1,X2,...,Xn (input) et une sortie vers d'autres neurones (output) :



Le comportement d'apprentissage des neurones biologiques a inspiré les scientifiques à créer un réseau de neurones qui sont reliés les uns aux autres. Imitant la façon dont l'information est traitée dans le cerveau humain, chaque neurone artificiel individuel enverra un signal à tous les neurones auxquels il est connecté quand suffisamment de ses signaux d'entrée sont activés. Ainsi, les neurones ont un mécanisme très simple sur le niveau individuel, mais quand on commence à avoir beaucoup, beaucoup (millions) de ces neurones empilés en couches et reliés entre eux, chacun de ces neurones sont connectés à des milliers d'autres neurones, cela donne un comportement d'apprentissage. Construisant un réseau neuronal multicouche s'appelle Deep Learning :



Le mécanisme de traitement des données dans un neurone est appelé l'activation, celle-ci dépend des entrées fournies au neurone et le poids associé à chaque entrée :

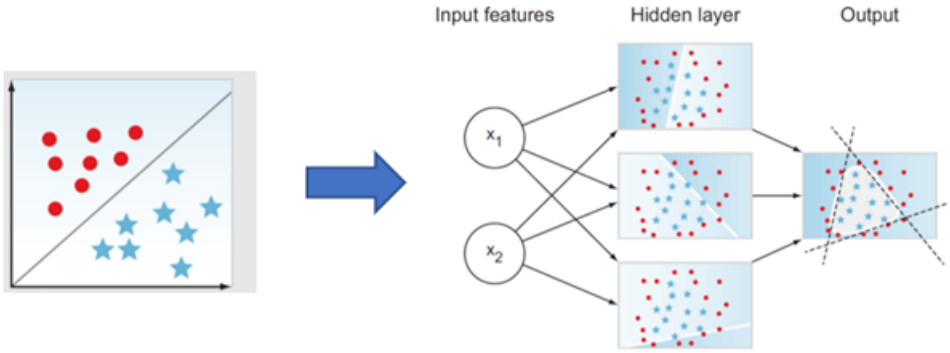


Le neurone effectue simplement une somme pondérée biaisé des entrées suivies par une fonction d'activation comme la fonction step (0 ou 1) ou une fonction linéaire. Le résultat ressemble donc à la fonction suivante :

$$\hat{y} = activation\ (\sum x_i . w_i + b)$$

Le neurone utilise la méthode d'essai et d'erreur pour apprendre de ses erreurs. Il utilise les poids comme boutons en ajustant leurs valeurs de haut en bas jusqu'à ce que le réseau soit entraîné.

D'ailleurs, on remarque que la prédiction d'un neurone est le résultat d'une fonction linéaire, ceci nous indique qu'un neurone ne peut traiter que des problèmes simples où les données peuvent être séparées par une seule ligne par exemple. Cependant, si le problème devient plus complexe, on a besoin de construire un vrai réseau en alliant plusieurs neurones dans des couches consécutives :

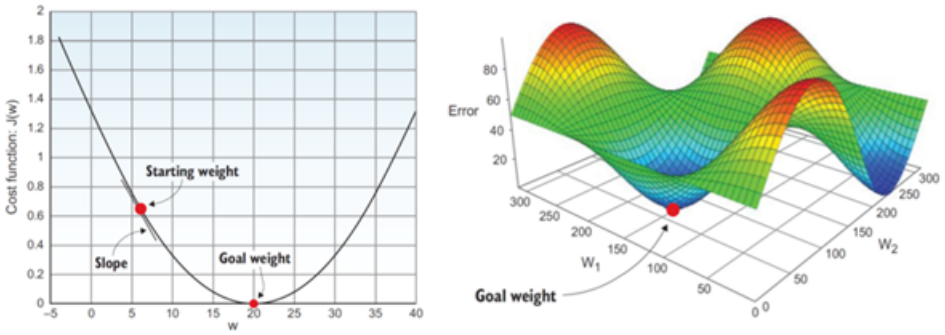


Le travail d'un ingénieur de machine learning est de construire un réseau qui n'est pas très simple (un neurone) ni très complexe (ce qui engendre un problème de sur-apprentissage). L'intuition et l'expérience sont des facteurs très importants pour la construction de réseaux de neurones.

2.2 L'entraînement des réseaux

Comme nous avons expliqué auparavant, l'entraînement d'un réseau consiste à chercher les valeurs optimales des poids pour que le réseau prédise de bonnes valeurs avec une erreur minimale. Dans le cas simple avec un seul neurone l'apprentissage consiste à : - calculer son "prédiction" $\hat{y}$ et le comparer avec la valeur correcte y qu'il doit retourner, - calculer l'erreur (error =y -$\hat{y}$), - mettre à jour les poids : si la prédiction est trop élevée, on ajustera les poids pour faire une prédiction plus faible la prochaine fois et vice versa, - On répète ces étapes jusqu'à avoir une erreur acceptable.

En considérant le cas où nous avons un seul poids ou deux poids à régler, on voit qu'on est dans une espace 2D ou 3D en fonction de l'erreur, la recherche des poids optimales dans ce cas est donc triviale :



Cependant, quand on a plus de 3 poids, le problème devient beaucoup plus complexe et on ne peut pas essayer toutes les valeurs de poids possibles. Par exemple, si on veut essayer 1000 valeurs de chaque poids pour un réseau qui contient 25 poids on a : 1000 * 1000 * ... * 1000 = $100^{25} = 10^{75}$ combinaisons.

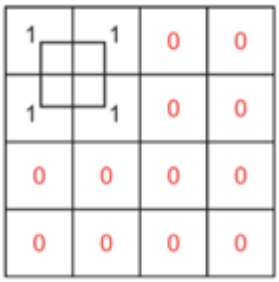
Pour tester toutes ces combinaisons en utilisant un super ordinateur, comme le Sunway Taihulight, qui fonctionne à une vitesse de 93 petaFLOPS, on aura besoin de $3.42 * 10^{50}$ années.

Pour arriver aux valeurs optimales des poids, on utilise des principes mathématiques simples qui utilisent la dérivée comme le décent de gradient et ses variants, on appelle ces derniers des optimiseurs. Le choix de la fonction qui caractérise l'erreur, la fonction d'optimisation et ses paramètres font aussi partie du travail de l'ingénieur de machine learning.

2.3 Les réseaux convolutifs

2.3.1 Les limites des réseaux conventionnels

Alors que les réseaux neuronaux comme nous avons vu sont très utilisés mais ils ont leurs limites, surtout quand il s'agit des images et vidéos où les caractéristiques spatiales sont importantes. Simplifions les choses pour comprendre plus sur l'importance des caractéristiques spatiales dans une image. Supposons que nous essayons d'apprendre à notre réseau de neurones à identifier la forme d'un carré. Dans l'image ci-dessous, considérez les valeurs de pixel 1 est blanc et 0 est noir. Ensuite, lorsque nous dessinons un carré blanc sur fond noir, la matrice ressemblera à ceci :



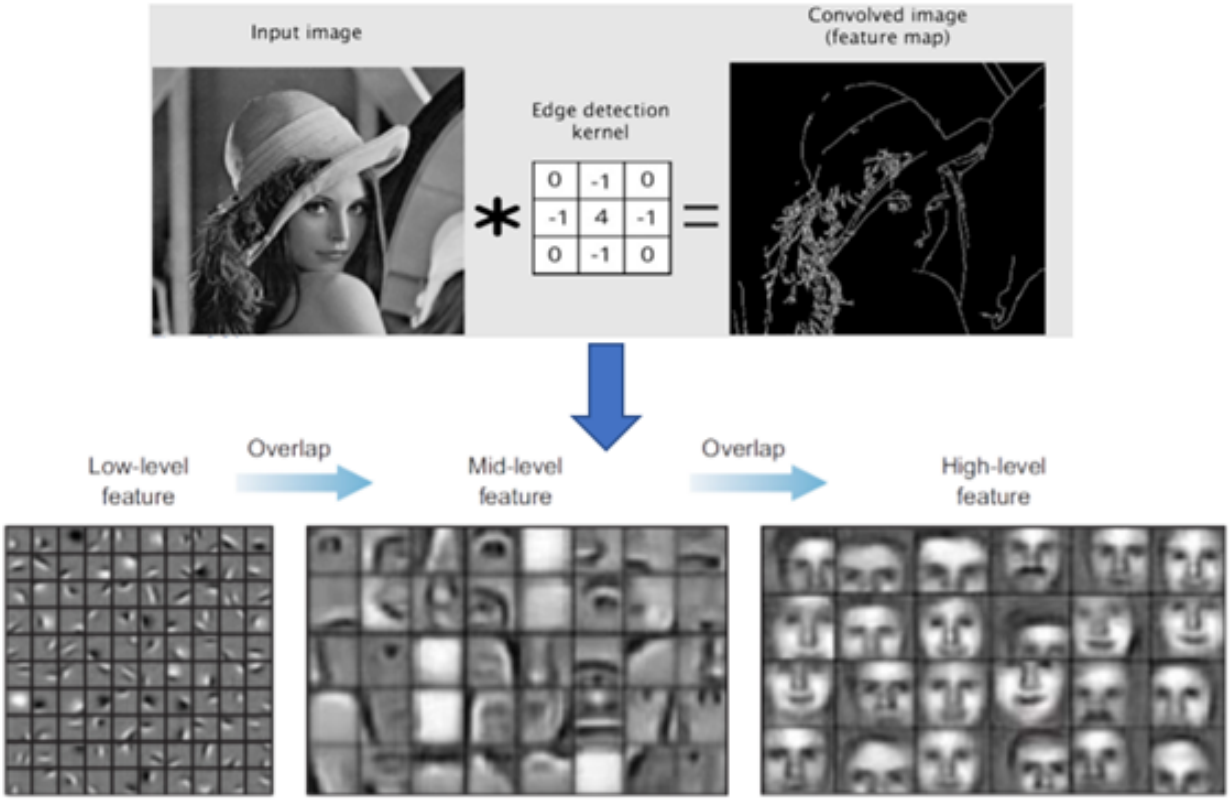
Étant donné que les réseaux conventionnels prennent le vecteur 1D en entrée, afin d'alimenter l'image au réseau, nous devons aplatir l'image 2D en un vecteur 1D. Le vecteur d'entrée de l'image ci-dessus ressemblera à ceci : Vecteur d'entrée = [1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Lorsque la formation est terminée, le réseau apprendra à identifier un carré uniquement lorsque les nœuds d'entrée x1, x2, x5 et x6 sont activés. Mais que se passe-t-il lorsque nous avons de nouvelles images avec des formes carrées mais situées dans des zones différentes de l'image ? Le réseau n'aura aucune idée qu'il s'agit de formes de carrés car le réseau n'a pas appris la forme carrée en tant que caractéristique. Au lieu de cela, il a appris les nœuds d'entrée qui, lorsqu'ils sont déclenchés, peuvent conduire à une carrée. Cela dit, pour enseigner les carrés à notre réseau, nous avons besoin de beaucoup de carrées situées partout dans l'image. Vous pouvez voir comment cette solution ne s'adaptera pas à des problèmes complexes, car pour enseigner les carrés à notre réseau, nous devons lui fournir des images de carrés situés partout dans l'image.

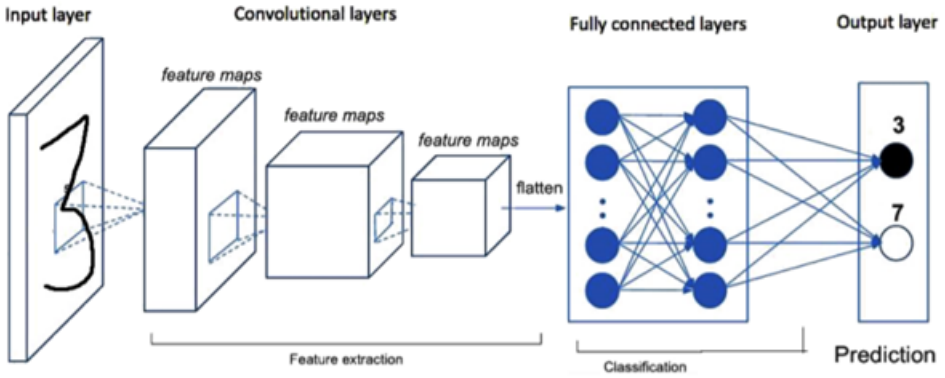
Autre exemple d'apprentissage des caractéristiques : si nous voulons apprendre au réseau à reconnaître les chats, idéalement, nous voulons que le réseau de neurones apprenne la forme complète des caractéristiques du chat, quel que soit l'endroit où elles apparaissent sur l'image (oreilles, nez, yeux, etc.) . Cela ne se produit que lorsque le réseau considère l'image comme un ensemble de pixels qui, lorsqu'ils sont proches les uns des autres, sont fortement liés. D'où l'utilisation du principe de la convolution pour créer des réseaux capables de distinguer les caractéristiques dans l'image, c'est ce qu'on appelle les réseaux convolutifs.

2.3.2 Principes des CNN

De leur nom, les réseaux convolutifs utilisent le principe de convolution pour extraire des caractéristiques à partir d'une image comme la détection des bords en utilisant des filtres de convolution (kernels). Ce principe utilisé dans la science du traitement d'images permet donc d'utiliser plusieurs filtres pour générer les caractéristiques différents contenus dans l'image (bords, courbures...). En organisant des filtres différents dans plusieurs couches, un réseau peut généraliser les caractéristiques simples comme les bords pour construire des caractéristiques de plus haut niveau :



L'ingénieur de machine learning doit donc choisir les tailles des filtres, le nombre de couches de convolution, le pas de convolution et d'autres paramètres du réseau. Les valeurs contenues dans les filtres deviennent donc les poids à optimiser pendant l'entraînement.

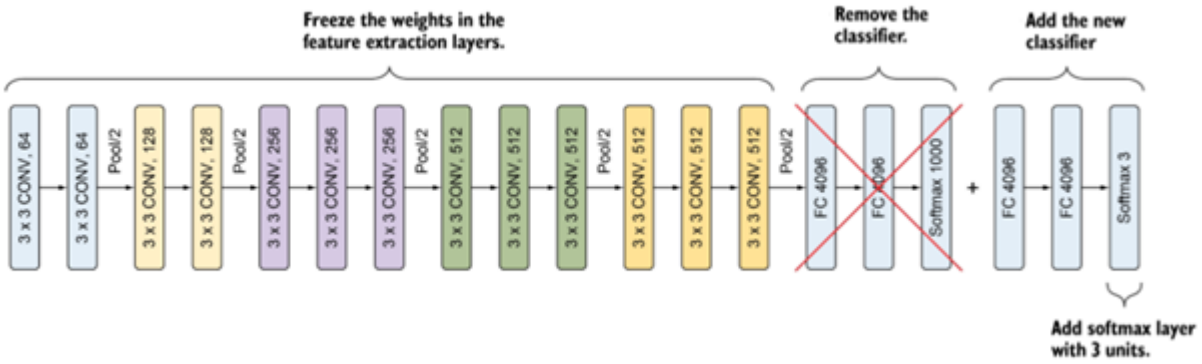


2.3.3 Transfer learning

Transfer learning est l'une des techniques les plus importantes de deep learning. Lors de la création d'un réseau pour résoudre un problème spécifique, on doit généralement collecter et étiqueter une énorme quantité de données pour entraîner ce réseau. Mais que se passerait-il si nous pouvions utiliser un réseau de neurones existant, que quelqu'un d'autre a réglé et formé, et l'utiliser comme point de départ pour notre nouvelle tâche ? Transfer learning nous permet de faire exactement cela. Nous pouvons télécharger un modèle open source que quelqu'un d'autre a déjà formé et ajusté pendant des semaines et utiliser leurs paramètres optimisés (les poids) comme point de départ pour former notre modèle un peu plus sur un ensemble de données plus petit que nous avons pour une tâche donnée . De cette façon, nous pouvons former notre réseau beaucoup plus rapidement et obtenir des résultats très élevés. L'idée est assez simple. Nous formons d'abord un réseau de neurones profonds sur une très grande quantité de données. Au cours du processus de formation, le réseau extraira une grande quantité de caractéristiques utiles qui peuvent être utilisées pour détecter des objets dans cet ensemble de données. Nous transférons ensuite ces caractéristiques extraites vers un nouveau réseau. Nous formons ensuite ce nouveau réseau sur notre nouveau jeu de données pour résoudre un problème différent. L'apprentissage par transfert est un excellent moyen de raccourcir le processus de collecte et de formation d'une énorme quantité de données simplement en réutilisant les poids du modèle à partir de modèles préformés qui ont été développés pour des

ensembles de données de référence de vision par ordinateur standard, tels que les tâches de reconnaissance d'image ImageNet.

L'implémentation de ce principe est très facile, il suffit de télécharger un réseau qui a été entraîné pour une tache semblable à la nôtre, récupérer les couches de convolution qui permettent d'extraire les caractéristiques, ajouter quelques couches de classification spécifiques pour notre problématique après les couches de convolution et faire l'entraînement avec notre dataset qu'on a construit et labélisé :

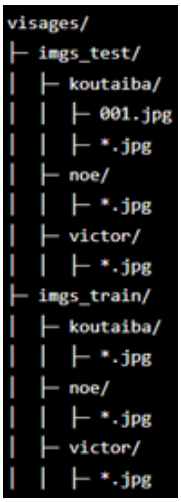


Dans notre projet, nous avons utilisé ce principe pour construire et entraîner notre réseau.

3 Traitement des données et entraînement du modèle

3.1 Traitement des données

Le projet consiste à détecter la présence des personnes dans une salle. Nous avons donc commencé par la détection des visages des étudiants qui travaillent sur ce projet : Koutaiba, Noé et Victor. Pour ce faire nous avons construit une base de données avec nos images. Nous avons commencé par l'enregistrement de quelques vidéos dans différents endroits avec des habits différents. Nous avons ensuite extrait des vidéos de multiples images puis nous avons utilisé OpenCV sous Python pour extraire les visages à partir de ces images. Le script d'extraction de visages sera fourni, ce script permet de traiter les images contenues dans un répertoire à la fois. Les images sont organisées dans des répertoires spécifiques en préparation pour la phase d'entraînement de la façon suivante :



On remarque que l'on a divisé les images en deux groupes, images de test et images d'entraînement. En effet, pendant l'entraînement, le réseau utilise les images d'entraînement pour mettre à jour les poids puis il utilise les images de test pour calculer des métriques de performance comme la précision et la valeur de l'erreur. Le but est d'avoir une bonne précision tout en ayant une erreur minimale.

De plus, le traitement de donnée est une étape importante pour l'entraînement du modèle. En effet, il faut standardiser les images en entrée du modèle pendant les phases d'entraînement et d'inférence en appliquant des transformations géométriques et des transformations sur les couleurs des images ainsi que les dimensions pour qu'ils soient uniformes à ce que le réseau

attend en entrée. Cette étape est cruciale pour la suite car c'est elle qui va permettre d'assurer que les images des datasets ont toutes le même format. Cela empêche notre modèle de se baser sur des détails tels que le format de l'image, sa luminosité et les méta-données.

3.2 Entraînement du modèle

Une fois que nos datasets étaient créés, nous avons pu passer à l'entraînement du modèle. Pour cela, nous avons choisi d'utiliser l'outil Google Colab qui permet d'utiliser les ressources de Google pour exécuter des programmes. Cette approche est très utilisée dans le domaine de l'IA, car la phase d'entraînement est très gourmande en calculs mais il faut aussi une RAM adaptée. Nous utilisons l'API Tensoflow Keras pour l'entraînement de notre intelligence artificielle supervisée. De plus, nous sommes partis du modèle existant MobileNetv2 qui est déjà entraîne pour l'extraction de caractéristiques propres aux visages. En utilisant ce modèle, nous avions seulement à modifier les dernières couches du réseau de neurones (en utilisant la méthode Dense) : les couches de décision. Le programme d'entraînement utilise des DataGenerator pour parcourir les datasets et envoyer le bon nombre d'images en fonction du batch size souhaité.

```
train_root = "/content/drive/MyDrive/ELEC3/projet_avance/visages/imgs_train"
test_root = "/content/drive/MyDrive/ELEC3/projet_avance/visages/imgs_test"

image_path = train_root + "/victor/0.jpg"
def image_load(image_path):
    loaded_image = image.load_img(image_path)
    image_rel = pathlib.Path(image_path).relative_to(train_root)
    print(image_rel)
    return loaded_image
image_load(image_path)

train_generator = ImageDataGenerator(rescale=1/255)
test_generator = ImageDataGenerator(rescale=1/255)

train_image_data0 = train_generator.flow_from_directory(str(train_root),target_size=(224,224))
test_image_data0 = test_generator.flow_from_directory(str(test_root), target_size=(224,224))

train_image_data = train_generator.apply_transform(train_image_data0, train_generator.get_random_transform(224,224))
test_image_data = test_generator.apply_transform(test_image_data0, test_generator.get_random_transform(224,224))

for image_batch, label_batch in train_image_data:
    print("Image-batch-shape:",image_batch.shape)
    print("Label-batch-shape:",label_batch.shape)
    break
for test_image_batch, test_label_batch in test_image_data:
    print("Image-batch-shape:",test_image_batch.shape)
    print("Label-batch-shape:",test_label_batch.shape)
    break
base_model = tf.keras.applications.MobileNetV2(input_shape=(224,224,3),
                                                include_top=False,
                                                weights='imagenet')

base_model.trainable = False
                       keras.layers.Dense(4, activation='softmax')])
x=base_model.output
x=GlobalAveragePooling2D()(x)
x=Dense(512,activation='relu')(x) #we add dense layers so that the model can learn more complex functions and classify for
better results.
preds=Dense(3,activation='softmax')(x) #final layer with softmax activation

model=Model(inputs=base_model.input,outputs=preds)
```

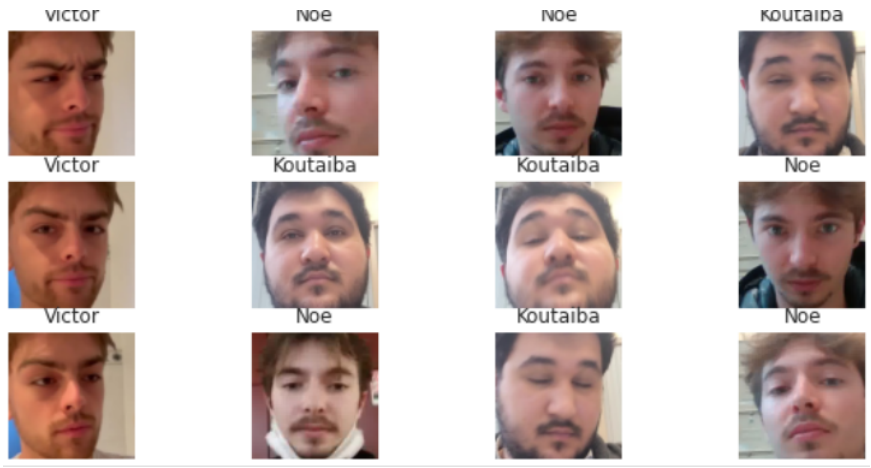
La phase d'entrainement est très empirique, en effet, après avoir écrit cet algorithme qui

va entrainer notre modèle à l'aide des fonctions de tensorflow.keras, nous devons essayer de jouer sur le nombre d'epochs, le batch size, l'optimizer ou encore les datasets pour améliorer notre modèle. Nous avions tout d'abord obtenu un entrainement qui semblait prometteur grâce à notre dataset d'environ 2000 photos de nos visages. Nous obtenions une précision très élevée selon la validation du modèle mais les images des deux datasets sont très proches donc l'entrainement est encore à améliorer.

Nous avons par la suite rajouté une classe en sortie pour indiquer qu'aucun des visages du dataset n'est reconnu. Pour cela, il suffisait de rajouter un dataset avec des visages de personnes aléatoires et d'augmenter la taille du vecteur de sortie de notre modèle.

```
preds=Dense(4,activation='softmax')(x) #final layer with softmax activation
```

Une fois notre entrainement terminé, nous vérifions que le modèle s'est bien entrainé en affichant les prédictions pour quelques images.



Ainsi nous vérifions après chaque entrainement que le modèle est suffisamment précis. Puis le modèle est sauvegardé à l'aide de la fonction suivante :

```
model.save('/content/drive/MyDrive/projet_avance/my_model_visages')
```

Cette fonction permet de sauvegarder le modèle de réseau de neurone. Un fichier contenant les premières couches pré-entrainées du modèle de base est créé ainsi qu'un autre fichier décrivant les dernières couches du réseau et le poids des variables de chacune de ces dernières.

4 Optimisation

4.1 Fonctionnement

Une fois la phase d'entraînement réalisée sur un ordinateur performant, un modèle est sauvegardé. Ce modèle peut ensuite être utilisé pour effectuer la phase d'inférence. Habituellement elle est aussi faite sur des systèmes très performants qui possèdent beaucoup d'espace mémoire. Cependant, dans le cadre de notre formation, nous travaillons avec de petits systèmes qui ont une puissance de calcul limitée.

Tensorflow propose une extension nommée Tensorflow Lite qui permet d'optimiser le modèle Tensorflow. Cette extension se caractérise par l'ajout de bibliothèques dans le fichier de conversion. Les fonctions utiles utilisées permettent de réduire la taille du modèle et de réduire la latence de l'inférence. En effet, le modèle utilisera moins de mémoire lors de l'exécution et occupera moins d'espace de stockage. Par ailleurs la latence est réduite, car Tensorflow Lite permet de réduire la quantité de calcul et la façon dont ils vont être fait.

Pour ce projet, nous avons tout d'abord testé l'optimisation automatique, puis nous avons tenté d'ajuster certains paramètres. La première étape a été de créer une base de donnée représentative. Cet ensemble de photos pris au hasard dans notre dataset d'entraînement permet au convertisseur de récupérer des informations lors du passage de ces images dans notre modèle. Il s'agit d'une fonction génératrice qui fournit un petit ensemble de données pour calibrer ou estimer la plage de valeur de tous les tableaux à virgule flottante (tenseurs) du modèle. Il récupère ainsi toutes les données à l'entrée du modèle, dans les couches intermédiaires et la sortie du modèle.

Une fois cette partie passée, nous avons indiqué au convertisseur que nous voulions travailler avec des entiers signés sur un octet. Cette étape est la plus efficace car nous pouvons baisser de manière drastique l'empreinte mémoire du modèle et faciliter les calculs. Cependant, si l'on sous-dimensionne trop le modèle, les résultats que l'on va obtenir seront moins précis. En faisant cela Tensorflow nous assure que notre modèle peut être réduit de 75% sans engendrer une perte de précision trop importante.

Modèle	Taille Modèle (MB)
Sans Conversion	8,0
Conversion Automatique	4,2
Conversion et Optimisation	2,5

Au regard de ce tableau on s'aperçoit bien de l'efficacité de Tensorflow Lite.

4.2 Démonstration

Pour optimiser, il faut ajouter les options de conversion suivantes :

```
# Convert the model
converter = tf.lite.TFLiteConverter.from_saved_model('/content/drive/MyDrive/ELEC3/projet_avance/my_model_test')

#Passe en entier sur un octet les poids

converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8  # or tf.uint8
converter.inference_output_type = tf.int8

tflite_model = converter.convert()

# Save the model.
with open('/content/drive/MyDrive/ELEC3/projet_avance/tflite_models/model_nono.tflite', 'wb') as f:
  f.write(tflite_model)
```

Juste après avoir sauvé le modèle Tensorflow, il faut appliquer le mode optimisation, puis générer la representative dataset et enfin choisir le type de quantification que l'on veut. Il ne reste ensuite qu'à lancer la conversion et à enregistrer le modèle Lite. Il faudra tout de même veiller à bien créer la fonction "representative_dataset"

Si la logique et l'utilisation de l'outil ont été compris, nous nous sommes cependant confronté à un problème matériel. En effet, la conversion est très gourmande et nous consommons toutes les mémoires RAM mis à disposition par google Colab. Ainsi, nous n'avons pas pu faire toute les optimisations souhaitées. Pour répondre à ce problème, il faudrait soit acheter la licence Google Colab Pro, soit lancer la conversion de façon locale.

5 Inférence

Une fois que notre modèle a été entrainé puis optimisé on peut alors procéder à l'inférence de l'IA sur notre Raspberry Pi. Nous avons besoin pour cela, de configurer la carte Raspi en question puis de développer un algorithme capable d'utiliser notre modèle pour faire de la reconnaissance faciale.

5.1 Création de l'environnement virtuel

La première étape de l'inférence consiste à installer tous les packages et librairies nécessaires pour utiliser les fonctions de Tensorflow. De plus, on a besoin dans notre algorithme d'une fonction capable de détecter le contour des visages, on a choisi d'utiliser la bibliothèque OpenCV qui propose un modèle d'IA entrainé pour cette tâche en particulier. À l'aide du shell script pip install, nous parvenons à installer les librairies.

```
sudo apt install libatlas-base-dev
pip3 install tensorflow
```

Il faut ensuite créer un environnement virtuel que l'on nomme tensorflow et que l'on active avec la commande source. On utilise pour cela, cette suite de commandes.

```
# apt-get install python3-venv
# python3 -m venv ~/tensorflow
# source ~/tensorflow/bin/activate
```

Une fois que l'environnement virtuel est créé et que toutes les librairies y sont installées, on peut alors développer l'algorithme qui permet de faire l'appel dans la classe.

5.2 Algorithme utilisant l'IA entrainée

L'objectif de notre projet est d'utiliser notre IA pour faire l'appel dans la classe depuis la carte Raspberry Pi. Pour cela, nous utilisons une caméra connectée à la carte raspi, cette caméra va permettre de capturer une vidéo constituée des images de la classe. Sur chaque

image capturée par la caméra, nous appliquons d'abord l'algorithme d'OpenCV de détection de contours de visages, puis nous appliquons notre IA aux visages extraits par OpenCV.

Nous avons d'abord développé un algorithme qui détecte les visages présents en temps réel sur l'image de la caméra. Notre algorithme est décrit par le pseudo-code suivant :

```
While {
      Capture d'une image
      OpenCV détecte les contours des visages sur l'image
      Si (la taille du visage est supérieure à un certain nombre de pixel){
            Sauvegarde des coordonnées du visage
            Changement de format du visage sauvegardé (resize(224,224))
            Prédiction de l'appartenance du visage avec notre IA
      }
      Affichage des rectangles autour des visages et de la précision de notre modèle.
}
```

Pour utiliser le modèle d'IA entrainé pour la détection de visages d'OpenCV nous avons dû télécharger le modèle "haarcascade" qui est un modèle d'IA assez reconnu dans le domaine. En effet, l'IA haarcascade d'openCV est déjà spécialisée dans la détection de visages donc on a décidé de s'en servir pour ne pas avoir à faire une seconde IA.

```
# Load the cascade
face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
# To capture video from webcam.
cap = cv2.VideoCapture(0)
# To use a video file as input
# cap = cv2.VideoCapture('filename.mp4')
```

Cependant, cette IA fonctionnait plus ou moins, car elle détectait parfois des visages là où il n'y en avait pas. Pour éviter ce problème, nous avons ajouté une condition dans notre algorithme. Cette condition empêche que les visages trop petits détectés par OpenCV soient mis en entrée de notre modèle et entourés par un rectangle. Ainsi, avec cette condition, nous ne détectons plus que les vrais visages et le problème d'openCV est quasiment totalement résolu.

Une fois que chaque visage présent dans l'image est détecté puis sauvegardé, nous pouvons alors appliquer notre modèle sur chaque visage repéré. Ce modèle doit nous donner pour chaque visage, le nom de la personne auquel il appartient.

Pour faire fonctionner notre modèle, il faut d'abord exporter les poids des filtres et le modèle de base dans la carte Raspberry. De plus, nous n'avions pas trouvé de moyen de sauvegarder le modèle au format TfLite à la fin de l'entraînement. Ainsi, nous sommes contraints

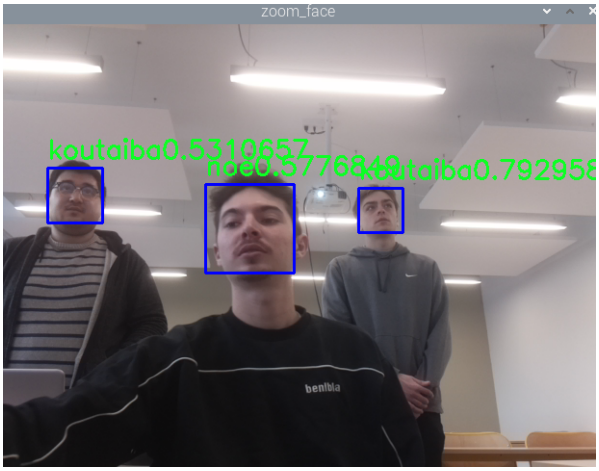
de sauvegarder et d'exporter le modèle au format tensorflow classique puis, une fois sur la carte Raspi, il faut que l'on convertisse le modèle en TensorflowLite à partir du modèle et des variables Tensorflow.

```
#load model
converter = tf.lite.TFLiteConverter.from_saved_model('my_model_test')

converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()

interpreter = tf.lite.Interpreter(model_content=tflite_model)
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Test the TensorFlow Lite model on random input data.
input_shape = input_details[0]['shape']
inputs, outputs = [], []
```

Ainsi, à chaque exécution de notre algorithme, il faut attendre cinq minutes pour que la conversion de Tensorflow vers TensorflowLite se fasse. Ensuite, le programme se lance et l'interface de la caméra apparait.

5.3 Résultats

Finalement, en se plaçant dans l'environnement virtuel (tensorflow) précédemment créé, nous pouvons exécuter notre algorithme. Contrairement aux résultats obtenus pendant l'entrainement, ceux de l'inférence n'étaient pas très concluants. En effet, avec notre premier modèle entrainé sur une large base de données, l'IA affichait "Koutaiba" pour presque tout le monde avec un pourcentage de précision élevé. Ainsi cela nous montrait que notre modèle n'avait pas été bien entrainé et qu'il fallait donc l'améliorer.



Nous avons ensuite décidé d'améliorer notre modèle, en rajoutant d'abord une classe en sortie pour indiquer qu'aucun de nous trois n'est reconnu. On a donc rajouté un dossier "nobody" dans chaque dataset. Les images de ce dossier proviennent du site internet https ://-thispersondoesnotexist.com/ car ce site WEB génère des visages humains qui n'appartiennent à personne grâce à une IA. Ainsi on a pu agrandir notre base de données.

Après ces améliorations notre modèle reconnaissant mieux certains visages comme ceux de Victor et Noé mais il donnait l'étiquette "nobody" la majeure partie du temps. Ainsi on a pu en déduire que notre modèle a été surentrainé et ne s'adapte pas au changement de contexte.

Pour faire l'appel dans la classe, il manque une dernière partie à notre algorithme. En effet, on pourrait effectuer cette tâche de plusieurs manières. La première solution serait de ne prendre qu'une photo et de détecter toutes les personnes présentes en une seule capture. Il suffit pour cela de vérifier quelles classes ont été détectées. Une autre solution serait de procéder de la même manière mais sur la totalité de la vidéo. En effet cette approche semble plus efficace car souvent OpenCV ne repère pas tous les visages présents dans une image. On pourrait aussi utiliser une approche statistique en regardant le nombre d'apparitions de chaque élève pour vérifier s'il dépasse un seuil fixé à l'avance. Cette méthode permettrait d'éviter les "faux présents" dans le cas où le modèle ferait quelques erreurs de reconnaissance faciale.

6 Conclusion

Le sujet abordé durant toutes ces séances nous a permis de découvrir de façon concrète l'intelligence artificielle. En effet, nous avions eu des cours d'introduction à l'IA mais le réel enjeu a été de faire fonctionner notre propre application. De plus nous avions la contrainte matérielle de la cible. La carte Raspberry avec une mémoire et une puissance de calcul limitée associée à une caméra pour faire tourner une IA de reconnaissance faciale.

Le projet dans son ensemble fut laborieux, car dans ce domaine il y a tout un concept et un environnement à maîtriser. Fort heureusement Tensorflow a une très bonne documentation et une assez grande communauté, ce qui nous a permit finalement de faire fonctionner notre application. Cependant elle ne répond pas totalement à nos attentes car notre IA ne nous reconnaît pas assez souvent.

Un des points d'amélioration de ce sujet pourrait être d'un peu plus guider le sujet et de créer un objectif à atteindre ; Taille modèle maximum, FPS minimum... ;

7 Bibliographie

https://github.com/EdjeElectronics/TensorFlow-Lite-Object-Detection-on-Android-and-Raspberry-Pi

https://github.com/tensorflow/examples/tree/master/lite/examples/object_detection/raspberry_pi#speed-up-model-inference-optional

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

https://www.tensorflow.org/model_optimization/guide/pruning/

https://magpi.raspberrypi.com/articles/tensorflow-ai-raspberry-pi