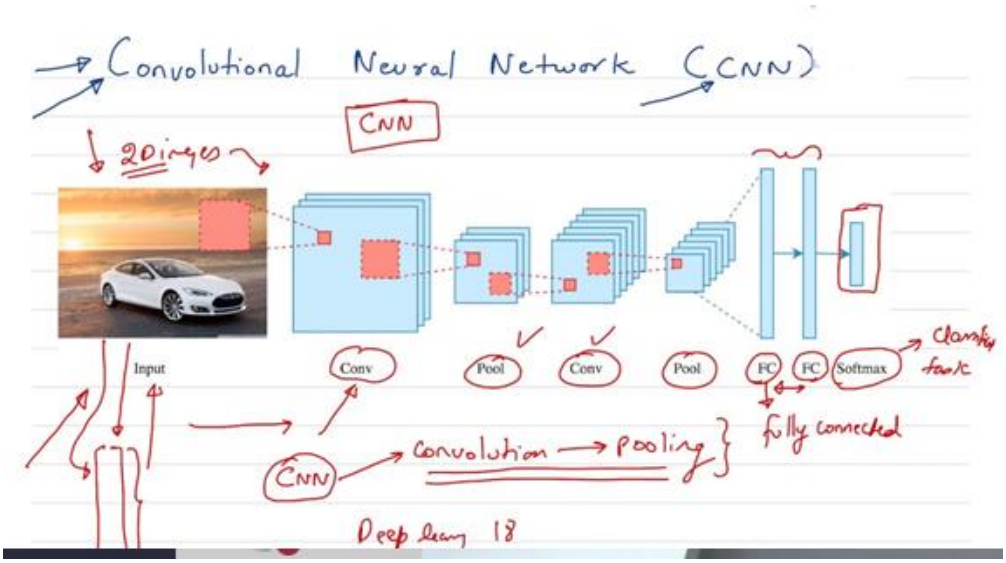


Rapport E2

ASSEMBLAGE D'UN MODELE D'APPRENTISSAGE PROFOND

Application des réseaux de neurones convolutifs (CNN) pour construire un modèle de détection des personnes portant des masques et celles n'en portant pas

DAO Ngoc Mai | Développeuse Data IA | Novembre 2020

1. Compréhension du besoin client

Comme nous le savons tous, le port d'un masque est l'une des mesures efficaces pour empêcher la propagation du COVID 19, mais s'assurer du bon respect de cette mesure n'est pas chose facile.

Pour cette raison, l'objectif de ce projet est d'utiliser un algorithme d'apprentissage en profondeur CNN avec différentes librairies Python (Tensorflow, Keras…) pour développer un modèle de détection de masques. J'espère qu'avec ce modèle, je pourrai être d'une certaine aide dans la lutte contre le COVID 19.

2. Traduction technique du projet

En deep learning, nous avons de nombreux algorithmes différents tels que réseau neuronal convolutif (CNN), réseau de neurones récurrents (RNN), réseaux de mémoire à long terme (LSTM)… Chaque algorithme a ses propres avantages et inconvénients.

Dans ce projet, j'ai choisi d'utiliser CNN, car c'est l'algorithme que je trouve le plus simple à comprendre parmi des algorithmes d'apprentissage profond, mais les résultats que j'ai obtenus sont bons. Je vais donc approfondir cet algorithme dans les sections suivantes. Je voudrais maintenant parler de deux autres algorithmes d'apprentissage profond qui sont RNN et LSTM.

Les RNNs sont une classe de réseaux de neurones artificiels qui peuvent traiter une séquence d'entrées en apprentissage profond et conserver son état tout en traitant la séquence d'entrées suivante.

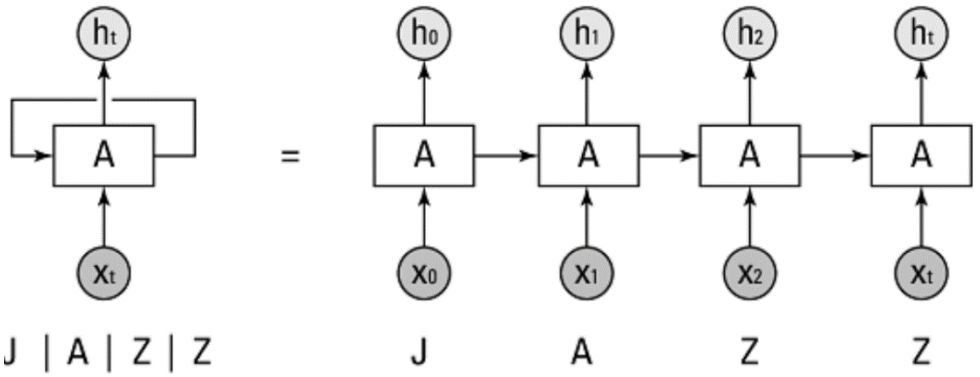


***Figure 1** : Réseaux de neurones récurrents*

Dans la figure 1, on voit que le réseau de neurones A prend une valeur « Xt » et produit une valeur « ht ». La boucle montre comment les informations sont transmises d'une étape à l'autre. Les entrées sont les lettres individuelles de « JAZZ » et chacune est transmise au réseau A dans le même ordre d'écriture (c'est-à-dire séquentiellement).

Les réseaux de neurones récurrents peuvent être utilisés de différentes manières, telles que :

- Détection du mot / lettre suivant
- Prévision des prix des actifs financiers dans un espace temporel
- Modélisation d'action dans le sport (prédire la prochaine action dans un événement sportif comme le football, le rugby, le tennis, etc.)
- Composition musicale
- Génération d'images

Les LSTM sont un type spécial de RNN capables de gérer des dépendances à long terme sans être affectés par un gradient instable.

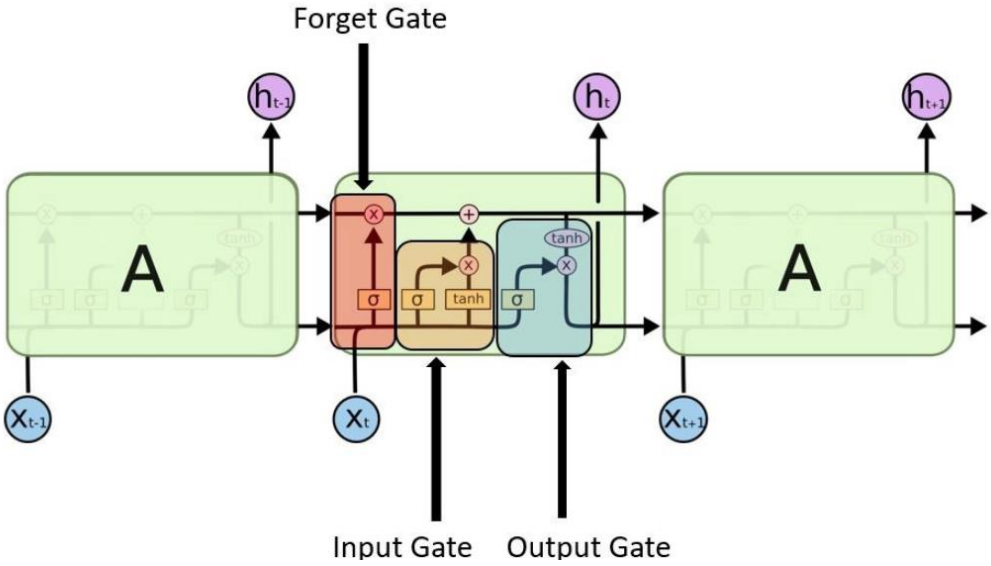


***Figure 2** : Réseaux de mémoire à long terme*

Le diagramme de la figure 2 est un RNN typique sauf que le module répétitif contient des couches supplémentaires qui se distinguent d'un RNN.

La différence ici est la ligne horizontale appelée «cell state» qui agit comme une bande transporteuse d'informations. Le LSTM supprimera ou ajoutera des informations à l'état de la cellule en utilisant les 3 portes comme dans la figure 2. Les portes sont composées d'une fonction sigmoïde et d'une opération de multiplication ponctuelle qui produit une valeur entre 1 et 0 qui nous permettra de décider si l'on met à jour la cellule ou non. Une valeur de 1 signifie laisser passer toutes les informations tandis que 0 signifie les ignorer complètement.

2. 1 Les réseaux de neurones convolutifs (CNN)

CNN est un algorithme de Deep Learning qui est utilisé pour tout usage autour de l'image ou de la vidéo comme la reconnaissance faciale ou encore la classification d'image.

Le nom réseau convolutif renvoie à un terme mathématique : le produit de convolution. En termes simples, l'idée est que l'on applique un filtre à l'image d'entrée, les paramètres du filtre nous permettrons d'apprendre les informations de l'image au fur et à mesure (figure 1).

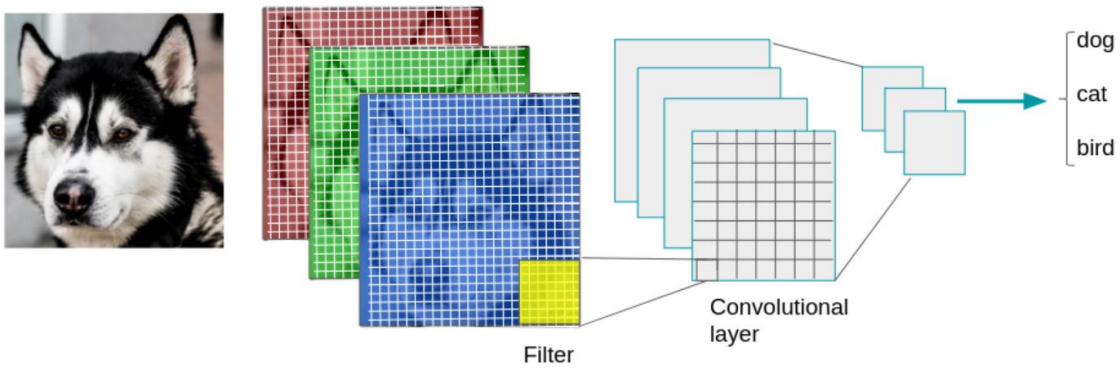


Figure 3 *: En Appliquant un filtre à l'image d'entrée, les paramètres du filtre nous permettre d'apprendre des informations de l'image*

2.2 Couche de convolution

La convolution est la première couche à extraire des entités d'une image d'entrée. La convolution préserve la relation entre les pixels en apprenant les caractéristiques de l'image à l'aide de petits carrés de données d'entrée.

Il y a deux formes de couches de convolution :

- Convolution simple (image noir et blanc)

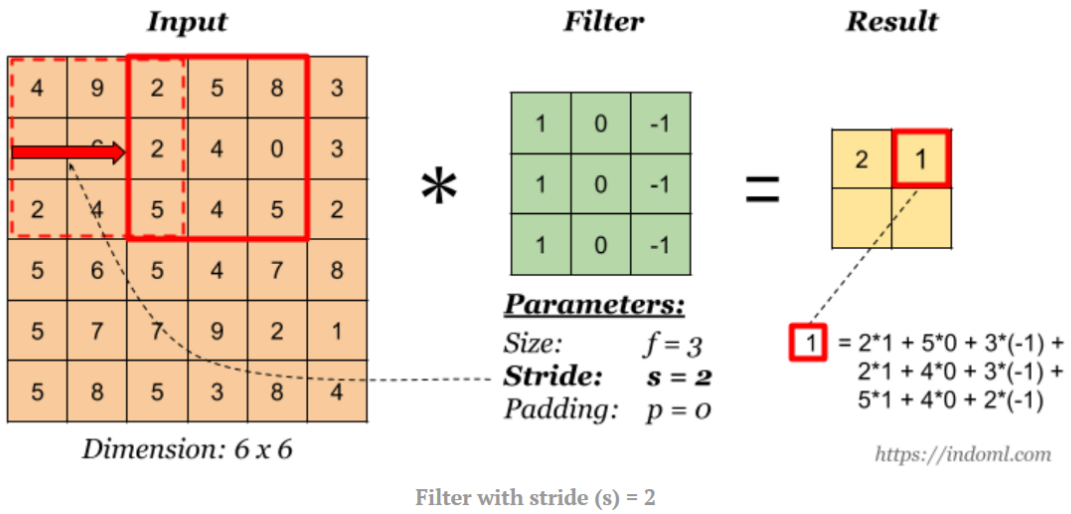


Figure 4 *: Convolution simple*

- Convolution 3D (image colorée)

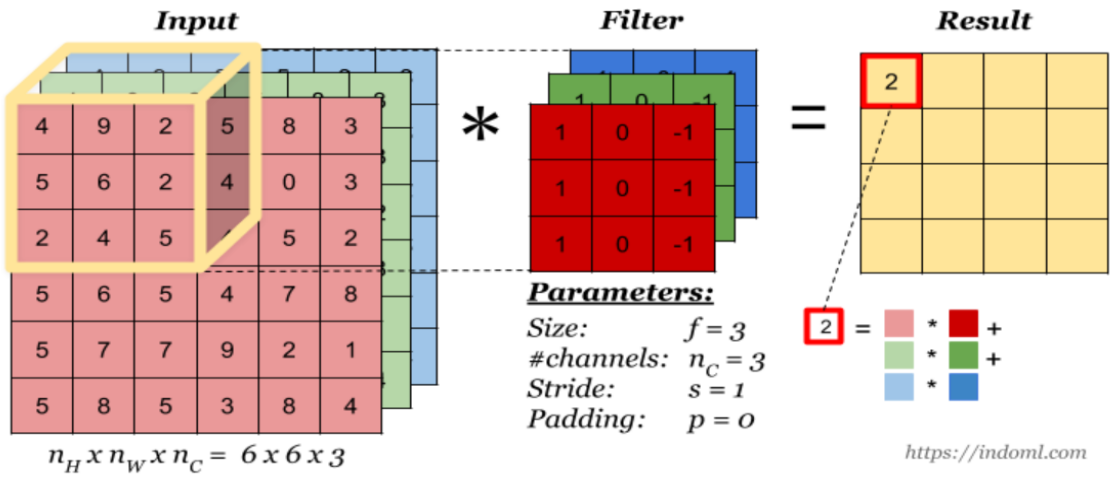


***Figure 5** : Convolution 3D*

2.3 Les paramètres dans un CNN

Comme le montrent les figures 4 et 5 de la section ci-dessus, nous pouvons voir de nombreux paramètres différents dans le réseau CNN, alors que sont-ils et que signifient-ils ?

- **Input** (H * W) : (hauteur x largeur) de l'image
- **Size** (f) : (hauteur x largeur) du filtre
- **Stride** (s) : de combien de cellules le filtre est déplacé sur les images l'entrée
- **Padding** (p) : Des valeurs « 0 » sont ajoutées autour des lignes et des colonnes d'entrée pour éviter de perdre des informations. Deux types de padding :
  - « Valide » : pas de padding
  - « Même » : la dimension de sortie est la même que l'entrée
- **Dimension** (N) : La forme de la sortie

$$N = [(H+ 2p-f) /s] + 1$$

- **Channels** (nc) : nombre de couches d'entrée (par exemple : une image RGB -> nc = 3

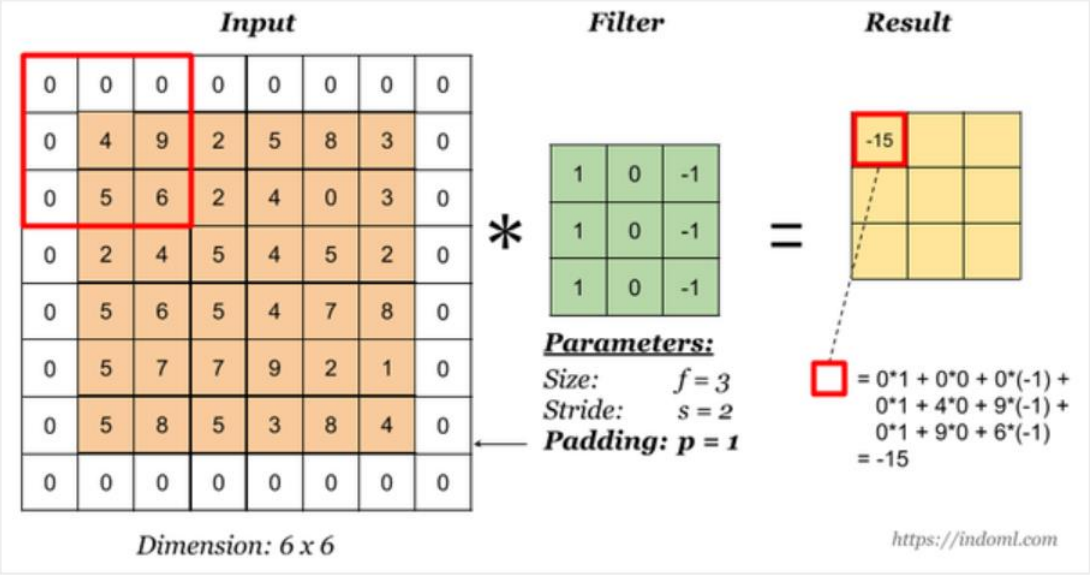


***Figure 6** : Avec padding : les « 0 » sont ajoutées aux lignes et colonnes de l'entrée*

2.4 Le principe du pooling

Le pooling est utilisé pour réduire la taille des représentations et accélérer les calculs.

Il y a deux principaux types de pooling : max pooling et average pooling (figure 7). Comme leur nom l'indique, max pooling prendra la plus grande valeur et average pooling fera une moyenne des valeurs.

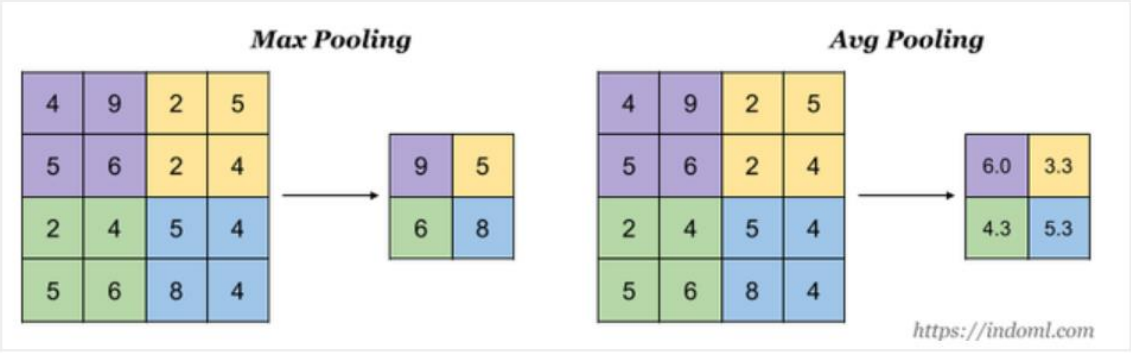


***Figure 7** : Deux types de pooling*

2.5 L'architecture d'un CNN

L'architecture d'un CNN est composée de plusieurs types de couches :

- Couche de convolution : renvoi le résultat du produit de convolution pour chaque morceau de l'image, quelques pixels à la fois, jusqu'à avoir scanné toute l'image.
- Couche de pooling : réduit la quantité d'informations générées par la couche convolutive pour chaque entité et conserve les informations les plus essentielles

L'utilisation de réseau de neurones met en œuvre 3 couches supplémentaires :

- Couche d'entrée entièrement connectée (Fully Connected Input Layer). L'opération « flatten » permet de transformer la matrice issue des couches précédentes en un vecteur unique (vecteur = matrice linéaire) pouvant être utilisé comme entrée pour la couche suivante.
- Couche entièrement connectée (Fully Connected Layer) : applique des pondérations sur l'entrée générée par l'analyse d'entités pour prédire une étiquette précise.
- La couche de sortie entièrement connectée (Fully connected output layer) ： génère les probabilités finales pour déterminer une classe pour l'image.

Avec un réseau CNN, nous pouvons avoir plusieurs couches de convolution et plusieurs pooling. Le processus des couches convolutives et de pooling se répète généralement plusieurs fois (figure 8).

Plusieurs filtres peuvent être utilisés dans une couche de convolution pour détecter plusieurs entités. La sortie de la couche aura alors le même nombre de canaux que le nombre de filtres dans la couche.

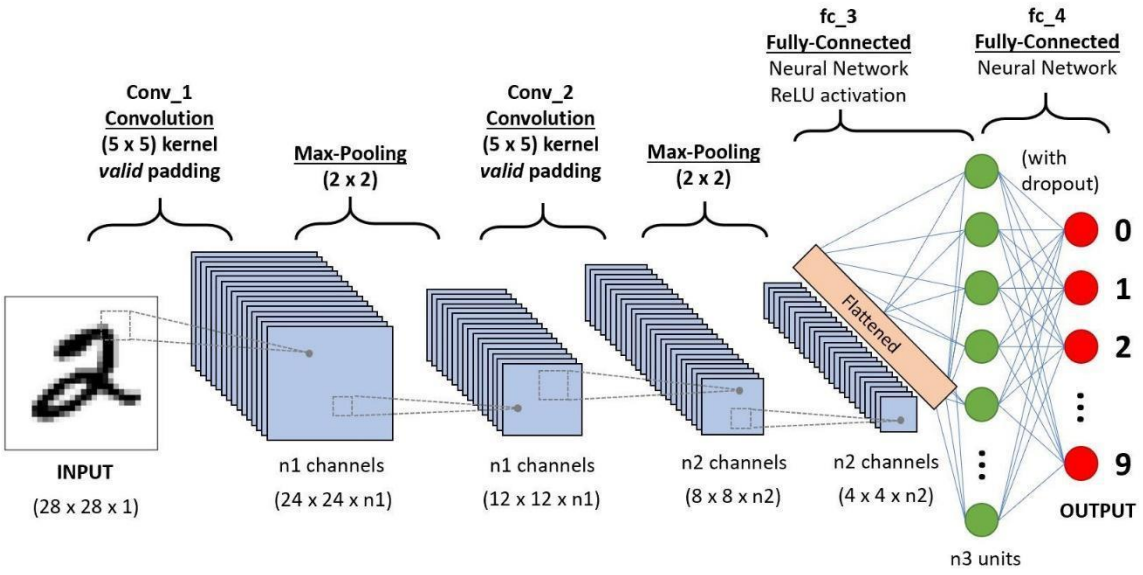


***Figure 8** : L'architecture d'un CNN*

3. La réponse technique mise en œuvre dans projet

3.1 Les données

Les données d'entrée sont des images avec une personne portant un masque ou non capturées par la caméra Vision AI Dev Kit.

Les données sont divisées en deux dossiers principaux : train (124 images) et test (16 images). Dans chaque dossier, il y a 2 sous-dossiers : masque et no-masque.

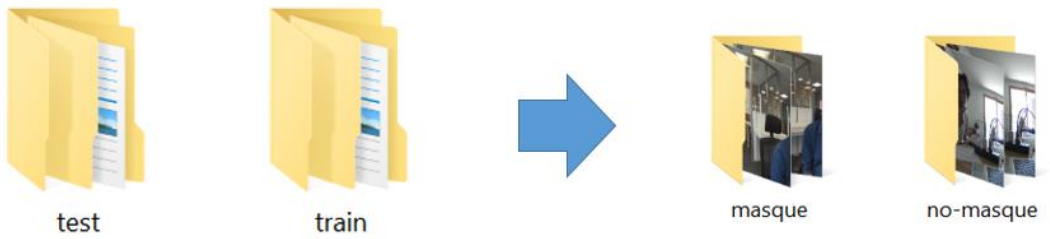
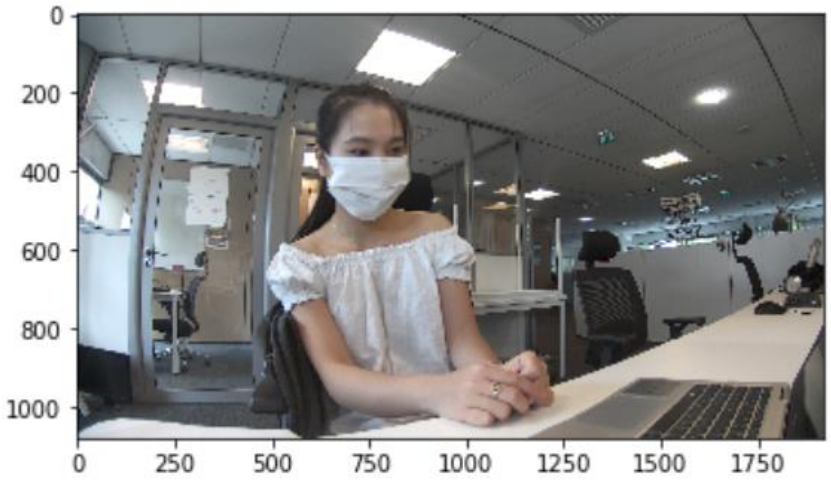


***Figure 9** : Stocké des données*

Nous allons maintenant regarder l'une des images et vérifier ses dimensions (Figure 10).

```
plt.imshow(masque_img)
```

```
<matplotlib.image.AxesImage at 0x1b4ab592978>
```




```
masque_img.shape
```

```
(1080, 1920, 3)
```

***Figure 10** : Une des images et ses dimensions*

Sur la figure 10, nous pouvons voir que les paramètres de l'image sont (1080, 1920, 3). Cela nous indique qu'il s'agit de données d'image couleur 3D (RGB) et que la taille de l'image est grande !

Si nous conservons cette taille pour l'apprentissage du modèle, la durée d'exécution du modèle sera très longue et l'ordinateur doit être suffisamment puissant, mais les résultats du modèle ne seront pas vraiment bons. C'est pourquoi je vais réduire la taille des images à (300, 500).

3.3 Construction du modèle

Tout d'abord, je vais importer les librairies :

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dropout, Flatten, Dense, Conv2D, MaxPooling2D
```

Le type de modèle que j'utiliserais est « Sequential ». Sequential est le moyen le plus simple de créer un modèle dans la libraire Keras. Il nous permet de construire un modèle couche par couche.

```
# Create a model
model = Sequential()

# 32 fileter with the size 3x3
model.add(Conv2D(filters=32, kernel_size=(3,3),input_shape=image_shape, activation='relu',))
# using max pooling
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3),input_shape=image_shape, activation='relu',))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3),input_shape=image_shape, activation='relu',))
model.add(MaxPooling2D(pool_size=(2, 2)))


model.add(Flatten())
```

J'utilise la fonction 'add ()' pour ajouter des couches à mon modèle.

Les 2 premières couches sont des couches Conv2D. Ce sont des couches de convolution qui traiteront les images d'entrée, qui sont considérées comme des matrices à 2 dimensions.

Les nombres 32, dans la première couche, et 64, dans la deuxième couche et la troisième couche, sont le nombre de "nodes" pour chaque couche. Ce nombre peut être ajusté, en fonction de la taille de l'ensemble de données. Après quelques tests, j'ai remarqué que les valeurs 64 et 32 fonctionnaient bien, j'ai donc gardé ces valeurs pour la suite du projet.

La taille du "kernel" est la taille de la matrice de filtre pour notre convolution. La taille de "kernel" choisi est (3,3), ce qui signifie que nous aurons une matrice de filtre 3x3.

L'activation est la fonction qui permet de démarrer le calcul sur la couche. La fonction d'activation que j'utilise pour mes couches est le ReLU. Cette fonction d'activation fonctionne bien dans les réseaux de neurones.

Le paramètre « input_shape » prend également une forme d'entrée. C'est la forme de chaque image d'entrée (300, 500, 3) comme vu précédemment, avec le 3 signifiant que les images sont en couleur.

Entre les couches Conv2D et la couche dense (vecteur unique), il y a une couche « Flatten ». Flatten sert de connexion entre la convolution et les couches denses.

« Dense » est un type de couche standard qui est utilisé dans de nombreux cas pour les réseaux de neurones et que j'ai utilisé pour ce projet.

```
# Output layer, its binary -> use sigmoid
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

J'utilise « sigmoid » pour activer la couche de sortie, car j'ai besoin d'un seul résultat soit 0 - masque, soit 1 – no-masque.

3.4 Compiler le modèle

La compilation du modèle prend en compte trois paramètres : la perte, l'optimiseur et les métriques.

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

Pour chacun de ses paramètres, il y a plusieurs choix possibles qui impact la précision des pondérations et le temp de calcul. Des pondérations plus précises entraineront des temps de calcul beaucoup plus long.

J'utilise «binary_crossentropy» pour la fonction de perte. Il s'agit du choix le plus courant pour une sortie binaire 0 ou 1. Plus la valeur du résultat sera basse et plus le taux de perte sera faible.

L'optimiseur contrôle le taux d'apprentissage. J'ai choisi "adam" comme optimiseur. Adam est généralement un bon optimiseur à utiliser dans de nombreux cas. L'optimiseur « adam » ajuste le taux d'apprentissage tout au long de la formation.

La métrique «accuracy» retenue permet de voir le score de précision sur l'ensemble de la validation lorsque j'entraîne le modèle.

3.5 Entraîner le modèle

```
batch_size = 16
```

```
train_image_gen = image_gen.flow_from_directory(train_path,
                                                target_size=image_shape[:2],
                                                 color_mode='rgb',
                                                batch_size=batch_size,
                                                class_mode='binary')
```

```
Found 124 images belonging to 2 classes.
```

J'utilise la fonction « flow_from_directory () » pour entraîner le modèle et les attributs les plus utilisés avec cette méthode :

- **target_size** : la taille des images d'entrée, chaque image sera redimensionnée à cette taille.
- **color_mode** : si l'image est soit en noir et blanc, soit en niveaux de gris, on définit la valeur à «grayscale», si l'image a trois canaux de couleur, on définit la valeur à «rgb».
- **batch_size** : nombre d'images à générer par batch.
- **class_mode** : Définit à «binary» si on n'a que deux classes à prédire, sinon définit sur «categorical»
- **shuffle** : défini à "True" si on veut mélanger l'ordre de l'image qui est renvoyée, sinon à "False". Default : True

Pour l'entraînement, j'utilise la fonction 'fit ()' sur mon modèle avec les paramètres suivants : données d'entraînement (train), données de validation (test), et nombre d'époques.

```
results = model.fit_generator(train_image_gen,epochs=20,
                              validation_data=test_image_gen,
                              callbacks=[early_stop])
```

Le nombre d'époques correspond au nombre de fois où le modèle parcourra les données. Plus nous aurons d'époques, plus le modèle s'améliorera, jusqu'à un certain point. Après ce point, le modèle cessera de s'améliorer à chaque époque. Pour mon modèle, j'ai choisi de fixer arbitrairement le nombre d'époques à 20.

Ici, nous voyons le paramètre « Early_stop » qui fait arrêter tôt le modèle si le modèle ne s'améliore plus afin de réduire le temps d'exécution du modèle.

4. Le bilan du projet et les améliorations possibles

```
Epoch 1/20
1/1 [==============================] - 4s 4s/step - loss: 0.6879 - acc: 0.5625
8/8 [==============================] - 39s 5s/step - loss: 2.0555 - acc: 0.5726 - val_loss: 0.6879 - val_acc: 0.5625
Epoch 2/20
1/1 [==============================] - 4s 4s/step - loss: 0.6814 - acc: 0.7500
8/8 [==============================] - 35s 4s/step - loss: 0.7287 - acc: 0.5000 - val_loss: 0.6814 - val_acc: 0.7500
Epoch 3/20
1/1 [==============================] - 4s 4s/step - loss: 0.6772 - acc: 0.6875
8/8 [==============================] - 36s 4s/step - loss: 0.6860 - acc: 0.6290 - val_loss: 0.6772 - val_acc: 0.6875
Epoch 4/20
1/1 [==============================] - 4s 4s/step - loss: 0.6263 - acc: 0.6250
8/8 [==============================] - 37s 5s/step - loss: 0.6684 - acc: 0.5806 - val_loss: 0.6263 - val_acc: 0.6250
Epoch 5/20
1/1 [==============================] - 4s 4s/step - loss: 0.6876 - acc: 0.5625
8/8 [==============================] - 37s 5s/step - loss: 0.6406 - acc: 0.6774 - val_loss: 0.6876 - val_acc: 0.5625
Epoch 6/20
1/1 [==============================] - 4s 4s/step - loss: 0.6556 - acc: 0.6250
8/8 [==============================] - 37s 5s/step - loss: 0.5899 - acc: 0.7016 - val_loss: 0.6556 - val_acc: 0.6250
```

taux de perte — precision

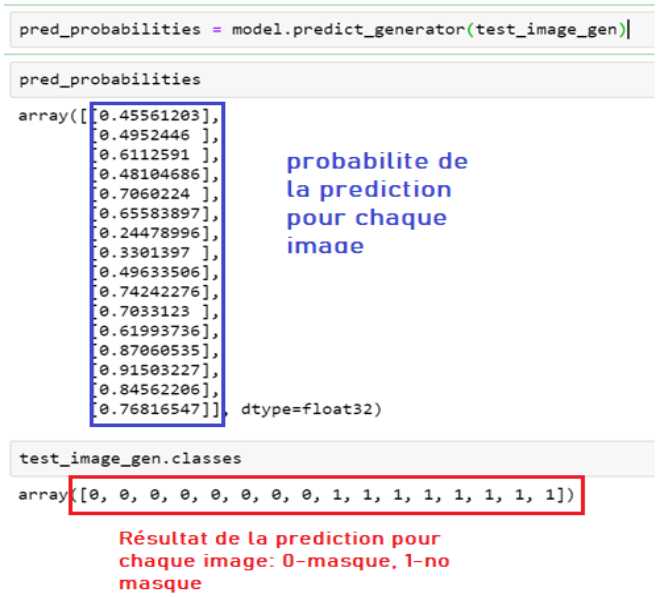
Après 6 époques, le modèle ne s'améliore plus et avec le paramètre « Early_stop », il s'arrête tôt même si on fixe 20 époques en total. On a obtenu une précision de 62,5%.

J'utilise ensuite la fonction « evaluer_generator() » pour évaluer le modèle, le résultat est 56.2% pour « masque », 68.8 % pour « no-masque », en moyenne, on a 62.5 % pour la précision du modèle.

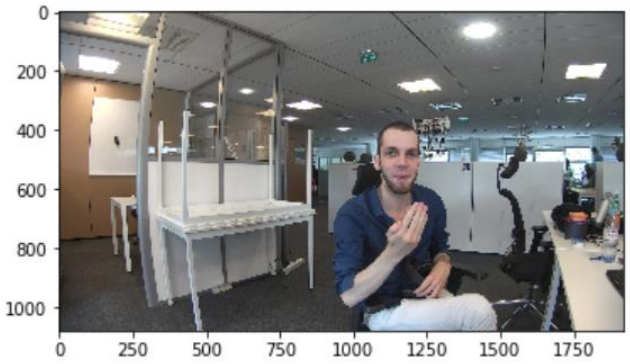
```
model.evaluate_generator(test_image_gen)
```

```
[0.562466025352478, 0.6875]
```

Pour voir les prédictions réelles que le modèle a faites pour les données de test je vais utiliser la fonction de prédiction : « predict_generator() » (voir ci-dessous). Le résultat est un tableau qui représente les probabilités que l'image testée soit avec un masque ou non. En dessous, on peut voir la prédiction sur chaque image comme ayant le masque ou non, 0 - masque et 1 – no-masque.



Je fais une prédiction avec une certaine image, l'exemple ci-dessous est avec une image sans masque, mais la distance du visage pour l'identification est assez éloignée.



Le modèle résultant a prédit que la personne sur l'image portait un masque (0 pour masque et 1 pour no-masque) alors que ce n'est pas le cas.

```
model.predict(my_image)
```

```
array([[0.]], dtype=float32)
```

J'ai réessayé avec une autre photo plus proche :

Ceci est l'image d'une personne portant un masque, et le résultat de prédiction du modèle est correct :

```
model.predict(my_image2)
```

```
array([[0.]], dtype=float32)
```

5. Conclusion

La détection des personnes portant ou non un masque avec ce modèle est correct sur les images avec sujets rapprochés mais est non satisfaisante pour images avec des sujets éloignés.

La phase d'entrainement de ce modèle c'est fait avec pas beaucoup de photos (problème de puissance de calcul de mon ordinateur). Ceci peut expliquer les résultats obtenus. De plus, le nombre d'images avec des sujets éloignés étaient aussi trop restreint.

Par contre, la puissance du modèle (nombre de paramétrage possible, etc…) devrait permettre d'obtenir de meilleur résultat à condition d'avoir la puissance de calcul nécessaire.