ARN: Practical Work 4

Rémi Ançay & Lucas Charbonnier

Rapport du travail pratique

Partie 1

C'est un Multi-Layer Perceptron (MLP) avec 1 couche cachée et une couche de sortie « softmax ».

L'algorithme utilisé est une descente de gradient par batch et les paramètres sont la taille de batch, le nombre d'epochs ainsi que la proportion du dataset utilisé pour la validation. La loss function utilisée est une « categorical crossentropy ».

Partie 2

Raw Data

Layer (type)	Output Shape	Param #
dense_40 (Dense)	(None, 10)	7,850
dense_41 (Dense)	(None, 10)	110

Total params: 7,960 (31.09 KB)

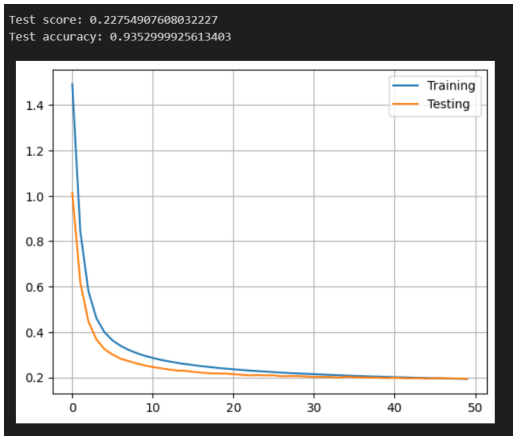
Trainable params: 7,960 (31.09 KB)

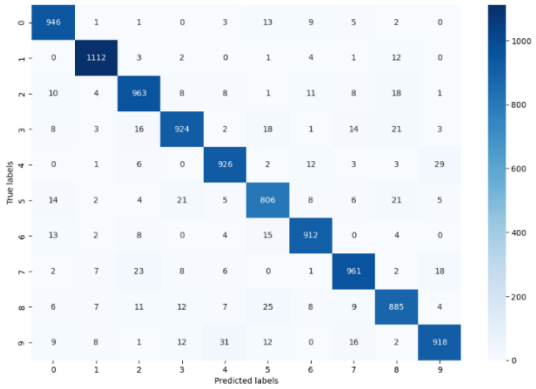
Non-trainable params: 0 (0.00 B)

Après plusieurs tests pour trouver les meilleurs paramètres, nous avons utilisé un MLP avec 1 couche cachée de 10 neurones et une couche finale à 10 sorties en « softmax ».

Comme nous avons 784 entrées connectés à 10 neurones, cela nous fait 7840 poids pour la première couche.
Nous avons ensuite cette couche cachée connectée directement aux 10 neurones de notre sortie. Cela nous fait 100 poids.
Nous ajoutons additionnons ces deux valeurs puis ajoutons encore les biais de chacun des neurones (20) pour un total de 7960

7840 + 10 + 100 + 10 = 7960 poids





Après entrainement du modèle, nous avons un score de 0.935.

Ce résultat est déjà bien. Nous voyons grâce à la matrice de confusion que la diagonale est très présente, donc les classes sont bien prédites.

HOG

Layer (type)	Output Shape	Param #
dense_6 (Dense)	(None, 10)	3,930
dense_7 (Dense)	(None, 10)	110

Total params: 4,040 (15.78 KB)

Trainable params: 4,040 (15.78 KB)

Non-trainable params: 0 (0.00 B)

```
n_orientations = 8
pix_p_cell = 4
hog_size = int(height * width * n_orientations / (pix_p_cell * pix_p_cell))
```

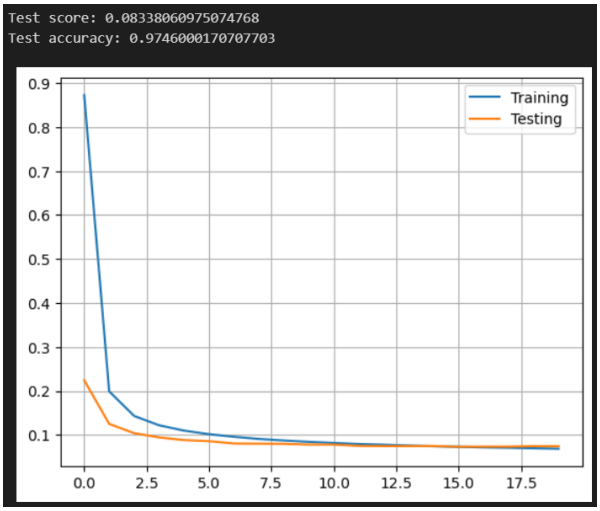
✓ 0.0s

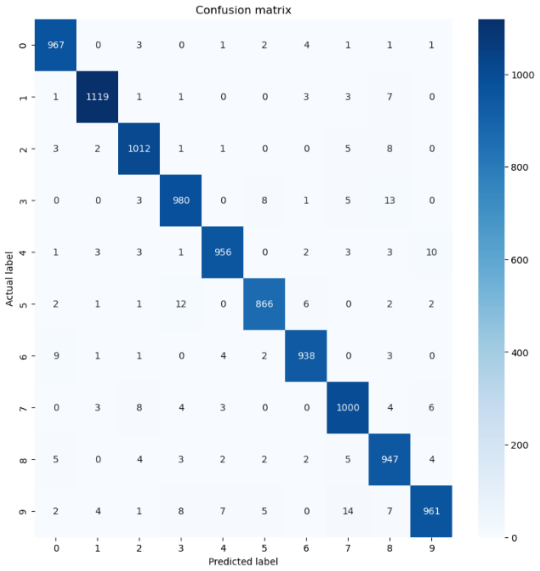
Comme pour le modèle précédent nous avons essayé plusieurs approches. Premièrement nous avons décidé de changer le nombre de neurone dans la couche cachée à 10. Nous avons ensuite joué avec les paramètres “n_orientation” et “pix_p_cell”. Leurs utilités étaient réduites, car ici nos images sont très petites. Dans notre cas elles n’ont pas apporté d’amélioration, donc nous les avons laissées à leur valeur initiale.

Avec le HOG, le nombre d'entrée est modifiée. Ici avec nos paramètres elle est de 392.
Le calcule est donc quasiment le même que pour la méthode précédente.

392 * 10 = 3920
10 * 10 = 100

3920 + 10 + 100 + 10 = 4040 poids





Après entrainement du modèle, nous avons un score de 0.975.

Ce résultat est mieux que celui de la méthode précédente. Nous pouvons donc voir que l'utilisation du HOG permet de réduire le nombre de poids tout en augmentant le résultat final.

CNN

Layer (type)	Output Shape	Param #
l0 (InputLayer)	(None, 28, 28, 1)	0
l1 (Conv2D)	(None, 28, 28, 10)	50
l1_mp (MaxPooling2D)	(None, 14, 14, 10)	0
l2 (Conv2D)	(None, 14, 14, 10)	410
l2_mp (MaxPooling2D)	(None, 7, 7, 10)	0
l3 (Conv2D)	(None, 7, 7, 10)	410
l3_mp (MaxPooling2D)	(None, 3, 3, 10)	0
flat (Flatten)	(None, 90)	0
l4 (Dense)	(None, 10)	910
l5 (Dense)	(None, 10)	110

Total params: 1,890 (7.38 KB)

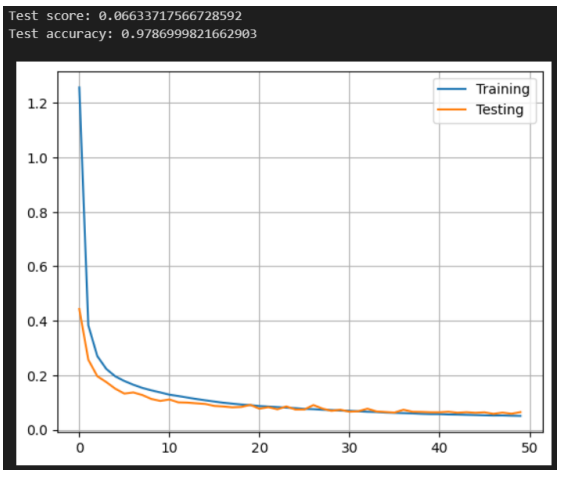
Trainable params: 1,890 (7.38 KB)

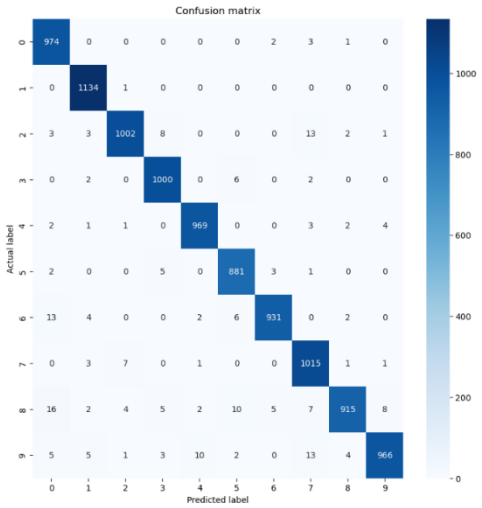
Non-trainable params: 0 (0.00 B)

Pour ce troisième essaie, nous utilisons un CNN. Nous avons donc décidé de le créer avec 3 couches de convolution avec chacune 10 neurones, puis de passer par une couche “Dense” de 10 neurones elle aussi.
Nous avons essayé d’ajouter une couche supplémentaire de convolution ou d’essayer plusieurs valeurs pour le nombre de neurones dans les couches cachées et nous somme finalement atterri sur ces valeurs.

L’image ci-dessus nous donne le nombre de poids nécessaire à cette architecture qui est de 1890.

C’est encore moins qu’avec les méthodes précédentes.





Grâce à cette méthode, nous obtenons un score de 0.979 ce qui est encore meilleur que précédemment.

Au regard du score obtenu et du nombre d'image dans le set de donnée, nous pouvons en déduire que cela sera difficile d'augmenter ce score. Cela s'explique peut-être par le fait que les images mal classifiées ne sont vraiment pas reconnaissables ou qu'elles viennent de système d'écriture différente.

Avec toutes les différentes matrices, nous pouvons voir que la plupart du temps ces chiffres sont confondu :

- Les 6 et les 0
- Les 8 et les 0
- Les 7 et les 2
- Les 7 et les 9
- Les 5 et les 8

Tous semblent plutôt logique.

Partie 3

Les réseaux de neurones convolutifs profonds ont plus de paramètres car ils ont plus de couches. Cela dépend bien évidemment de la taille de chaque couche mais, en principe, plus il y a de couches, plus il y a de paramètres.

Partie 4

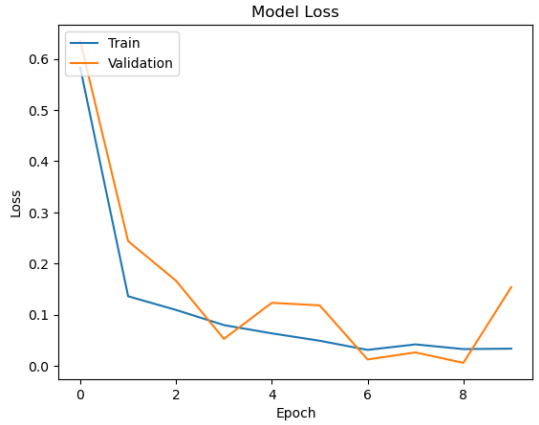
Voici le code permettant de créer l'architecture du réseau présenté dans la donnée :

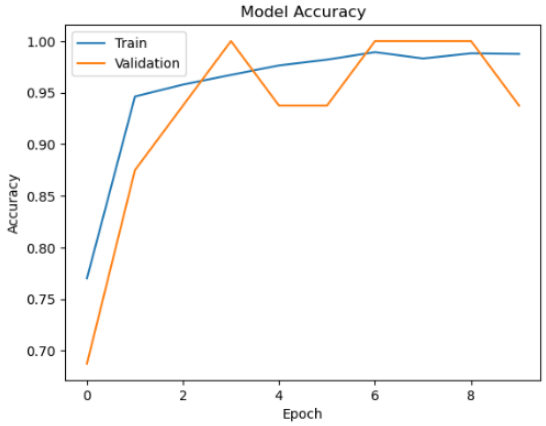
```
# Define CNN model
input_8 = layers.Input((IMG_HEIGHT, IMG_WIDTH, 1), name='input_8')

# note: use convolutions with relu and kernel size of 3.

conv_1 = Conv2D(8, (3, 3), padding='same', activation='relu', name='conv_1')(input_8)
max_pooling_1 = MaxPooling2D(pool_size=(2, 2), name='max_pooling_1')(conv_1)

conv_2 = Conv2D(16, (3, 3), padding='same', activation='relu', name='conv_2')(max_pooling_1)
max_pooling_2 = MaxPooling2D(pool_size=(2, 2), name='max_pooling_2')(conv_2)

conv_3 = Conv2D(32, (3, 3), padding='same', activation='relu', name='conv_3')(max_pooling_2)
max_pooling_3 = MaxPooling2D(pool_size=(2, 2), name='max_pooling_3')(conv_3)

conv_4 = Conv2D(64, (3, 3), padding='same', activation='relu', name='conv_4')(max_pooling_3)
max_pooling_4 = MaxPooling2D(pool_size=(2, 2), name='max_pooling_4')(conv_4)

conv_5 = Conv2D(128, (3, 3), padding='same', activation='relu', name='conv_5')(max_pooling_4)
max_pooling_5 = MaxPooling2D(pool_size=(2, 2), name='max_pooling_5')(conv_5)


flatten_7 = Flatten(name='flatten_7')(max_pooling_5)

dense_21 = Dense(32, activation='relu', name='dense_21')(flatten_7)
dense_22 = Dense(16, activation='relu', name='dense_22')(dense_21)


cnn_output = layers.Dense(1, activation='sigmoid', name='dense_23')(dense_22)
cnn = Model(inputs=input_8, outputs=cnn_output)
```

Nous entraînons ensuite le modèle avec les images du dataset :

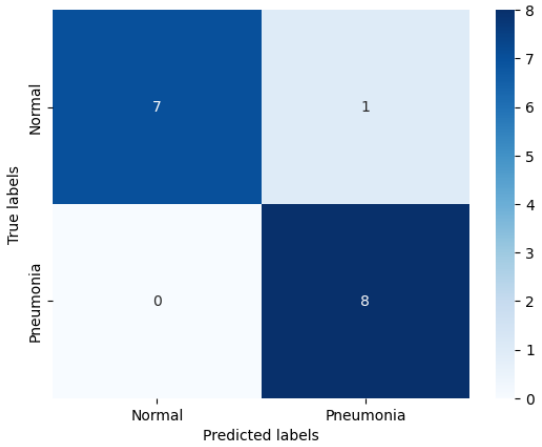
```
history = cnn.fit(
    train_ds,
    validation_data=val_ds,
    epochs=10,
    class_weight=class_weights
)
```

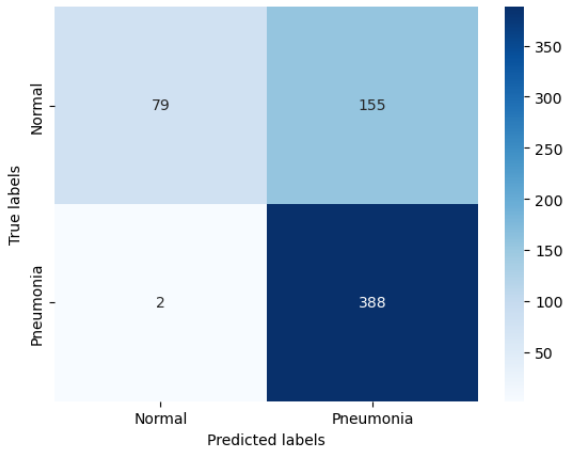
Voici le graphe de la performance du modèle pendant l'entraînement avec les données d'entraînement vs les données de validation :





Matrices de confusion pour les données de validation et de test :





Dataset de validation
Accuracy : 0.9375
F1-score : 0.9412

Dataset de test
Accuracy : 0.7484
F1-score : 0.8317

On peut voir qu'il y a une bien plus grande proportion de faux-positifs avec le dataset de test qu'avec le dataset de validation. Cela est principalement dû au fait que ces deux datasets ne contiennent pas des proportions similaires de positifs que de négatifs (pneumonies/normal).

Discussion autour des résultats obtenus

Class weights

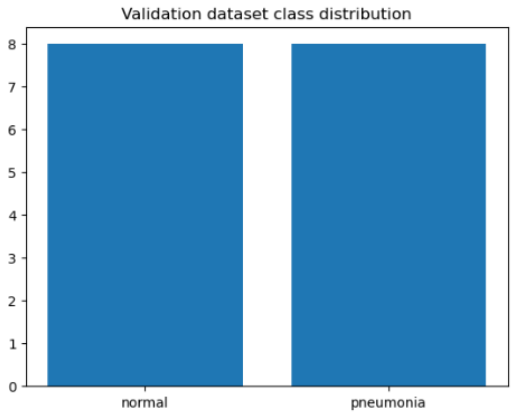
Il est important de noter l'utilisation de class weights pour cette expérience. Ces derniers permettent de « punir » plus fortement la présence de faux-négatifs dans les résultats que celle de faux-positifs. Cela a fortement influencé les résultats et on peut le voir sur la matrice de confusion finale ; il y a très peu de faux-négatifs.

Étant donné que cette expérience a un but médical, il est normal de vouloir éviter à tout prix les faux-négatifs. Si le modèle prédit une pneumonie alors qu'il n'y en a pas, cela pourrait donner lieu à des tests plus poussés qui permettraient de détecter ce faux-positif. Dans le cas inverse, c'est plus dangereux car un patient ne va pas forcément se refaire tester s'il pense qu'il n'a rien.

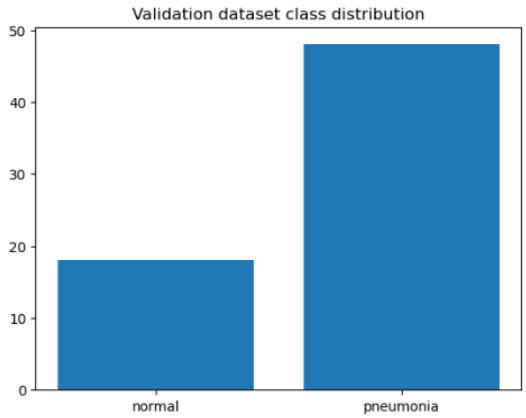
Ajustement du dataset de validation

Nous avons remarqué que le dataset de validation ne comportait que 8 données de chaque classe (comme montré dans la figure ci-dessous). Ce n'est pas beaucoup et, de notre point de vue, pas suffisant pour que le modèle puisse correctement évaluer sa propre performance.

Pour remédier à cela, nous avons déplacé quelques données provenant du dataset d'entraînement dans le dataset de validation afin que le modèle dispose de plus d'exemples pour s'évaluer.



Avant



Après

Nous avons ajouté majoritairement des exemples de pneumonie en plus car le plus important pour notre modèle est de pouvoir détecter ces cas de pneumonie. Comme dit plus haut, c'est plus important de détecter correctement une pneumonie plutôt que d'interpréter correctement l'absence de pneumonie.

Grâce à cela, nous avons pu légèrement améliorer les performances du modèle sur le dataset de test final. Nous sommes passé d'un f1-score de ~0,83 à ~0,86.

Étant donné que ce sont des datasets différents, cette amélioration n'est probablement pas due à de l'overfitting mais sûrement due au fait que le modèle a pu mieux s'évaluer pendant l'entraînement (et donc mieux s'ajuster). Cette amélioration peut aussi être attribuée au fait que, avec nos modifications, le dataset d'évaluation est plus ressemblant au dataset de test.