

# Cursus Ingénieur Machine Learning

-

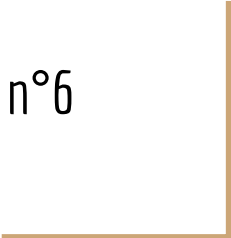
Vincent Jugé

-

Soutenance Projet n°6

-

07/2022





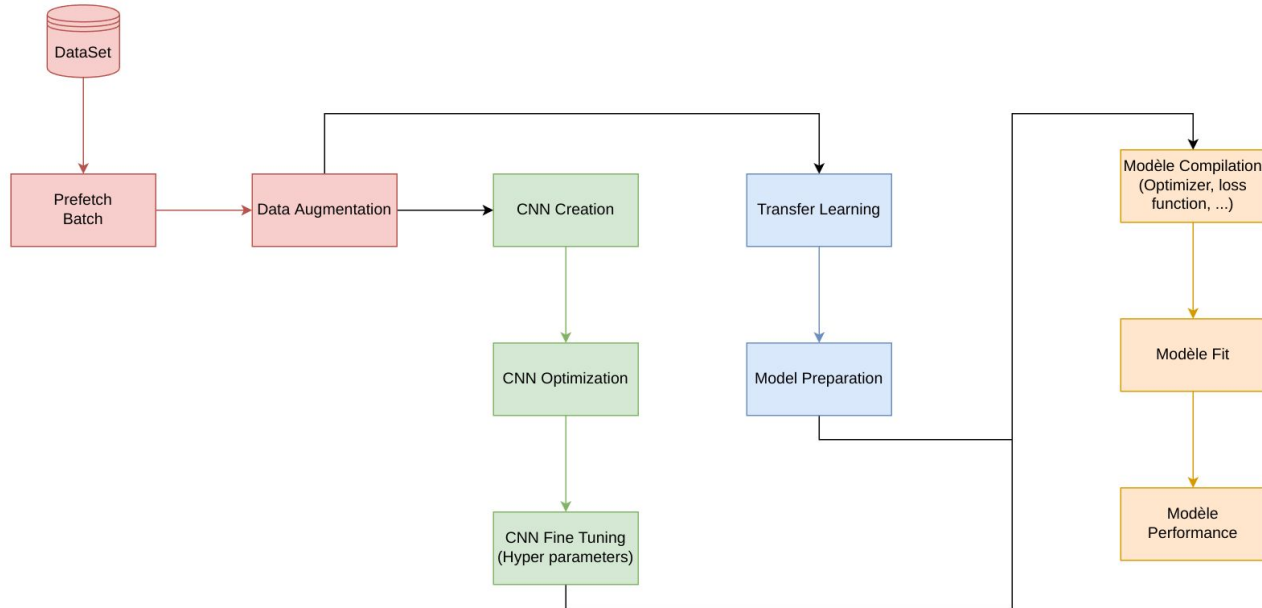
# Stackoverflow

Classez des images à l'aide d'algorithmes de Deep  
Learning



# Overview

Aperçu général des étapes:



# Environnement d'exécution

- On a besoin d'avoir un GPU compatible CUDA / TensorCores pour tirer pleinement parti de Tensorflow
- On utilise Google Collaboratory (plan payant avec des ressources GPU)

# Dataset

- On utilise le dataset “*Stanford Dogs*” qui est disponible via l’API `tensorflow_datasets`
  - [https://www.tensorflow.org/datasets/catalog/stanford\\_dogs](https://www.tensorflow.org/datasets/catalog/stanford_dogs)
  - Il contient l’ensemble des images, labels et permet de faire un train/test split facilement
  - Il ne nécessite pas de télécharger les archives des images
  - c’est une extension de la classe générique `DataSets`
  - cette API permet de batcher le dataset, lors de son utilisation par le modèle (phase d’apprentissage)
- On a 120 classes de chiens différentes
- Environ 20’000 samples

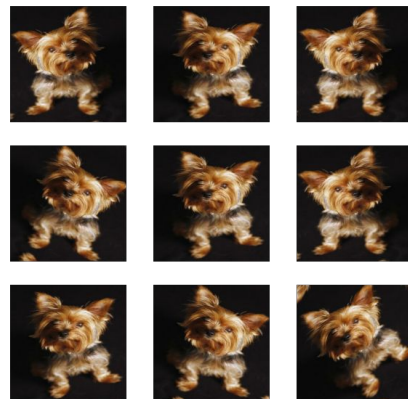
# Data Preprocessing & Augmentation

- Preprocessing

- Afin d'avoir des entrées de taille identique pour notre modèle, on effectue un resizing
- On effectue un One Hot encoding sur les labels

- Augmentation

- On ne dispose pas d'une grande variété de données, on doit donc en "créer" de nouvelles
- transformation des images par rotation et flip horizontal
- Pour une image du dataset, on en obtient neuf



# Création de modèles CNN

# Création de CNN - Fonctions communes

Pour la suite, on choisit des paramètres communs à tous nos modèles:

- optimizer : Adam
  - loss: CategoricalCrossEntropy
  - metrics: CategoricalAccuracy, Precision, Recall
- 
- Par défaut 20 epochs pour l'apprentissage
  - Early stopping par défaut à 8 epochs
  - Adaptation du learning rate: diminution au fur et à mesure de l'apprentissage



# CNN basique - cnn1

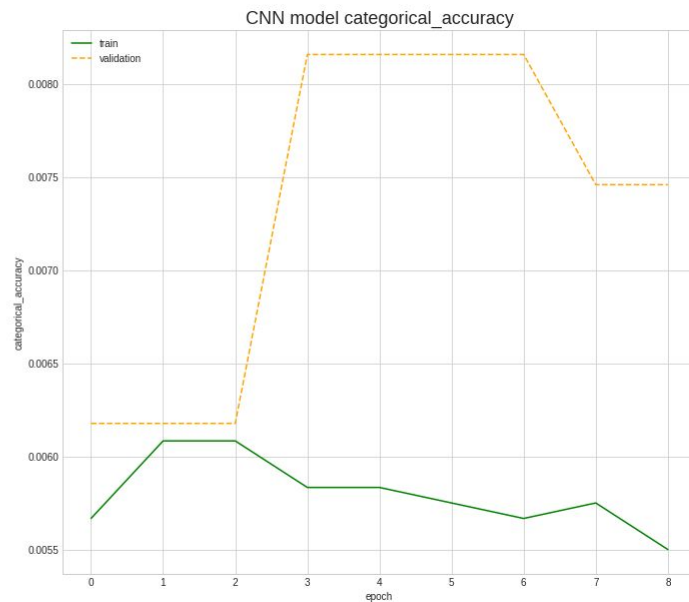
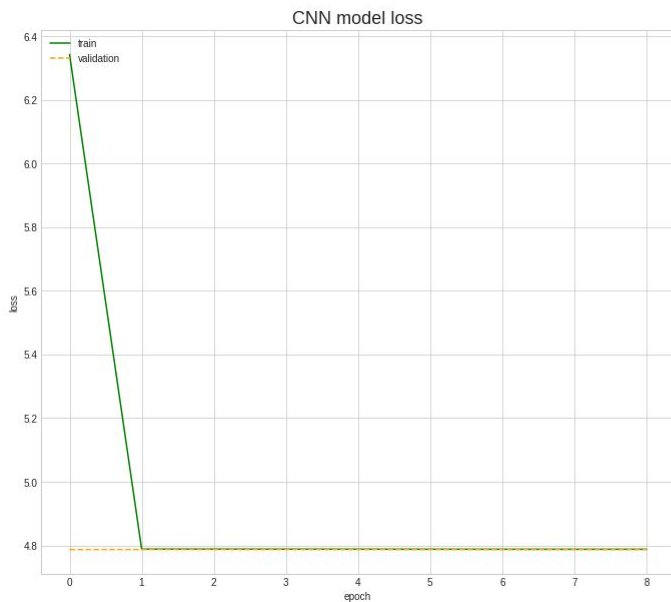
Pour commencer on crée un modèle CNN à une seule couche de convolution

C'est une approche très basique, mais qui permet de voir les évolutions pour la suite

```
Model: "sequential_6"
-----
Layer (type)                 Output Shape              Param #
-----
rescaling_4 (Rescaling)      (None, 299, 299, 3)      0
conv2d_10 (Conv2D)           (None, 297, 297, 32)     896
max_pooling2d_7 (MaxPooling  (None, 148, 148, 32)     0
  2D)
dropout_7 (Dropout)          (None, 148, 148, 32)     0
flatten_2 (Flatten)          (None, 700928)           0
dense_8 (Dense)               (None, 128)              89718912
dense_9 (Dense)               (None, 120)              15480
-----
Total params: 89,735,288
Trainable params: 89,735,288
Non-trainable params: 0
```

# cnn1

Évidemment, le modèle est très peu performant et n'apprend pas grand chose



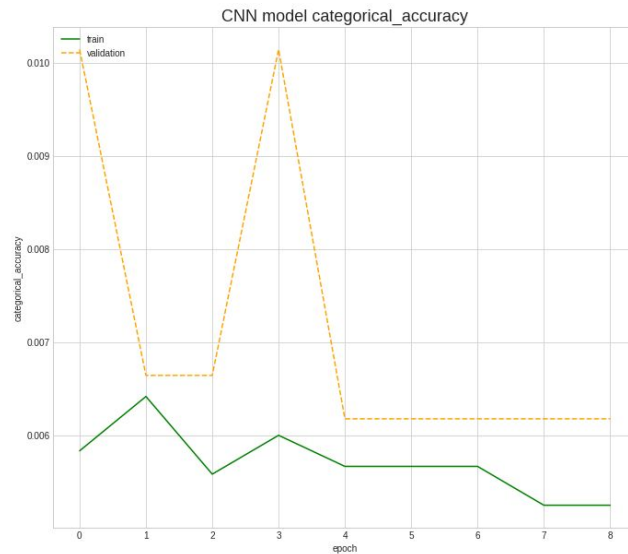
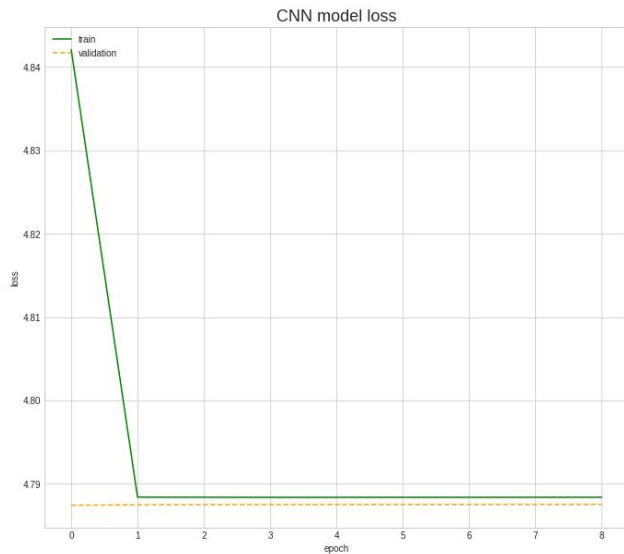
# CNN à 3 couches - cnn3

On ajoute quelques couches de convolution, pour voir la différence:

Layer (type)	Output Shape	Param #
rescaling_5 (Rescaling)	(None, 299, 299, 3)	0
conv2d_11 (Conv2D)	(None, 297, 297, 32)	896
max_pooling2d_8 (MaxPooling 2D)	(None, 148, 148, 32)	0
dropout_8 (Dropout)	(None, 148, 148, 32)	0
conv2d_12 (Conv2D)	(None, 146, 146, 64)	18496
max_pooling2d_9 (MaxPooling 2D)	(None, 73, 73, 64)	0
dropout_9 (Dropout)	(None, 73, 73, 64)	0
conv2d_13 (Conv2D)	(None, 71, 71, 128)	73856
dropout_10 (Dropout)	(None, 71, 71, 128)	0
flatten_3 (Flatten)	(None, 645248)	0
dense_10 (Dense)	(None, 128)	82591872
dropout_11 (Dropout)	(None, 128)	0
dense_11 (Dense)	(None, 120)	15480
=====		
Total params: 82,700,600		
Trainable params: 82,700,600		
Non-trainable params: 0		

# cnn3

Là aussi, les performances ne sont pas bonnes, pratiquement identiques à cnn1



# Optimisation de cnn3 - cnn3v2

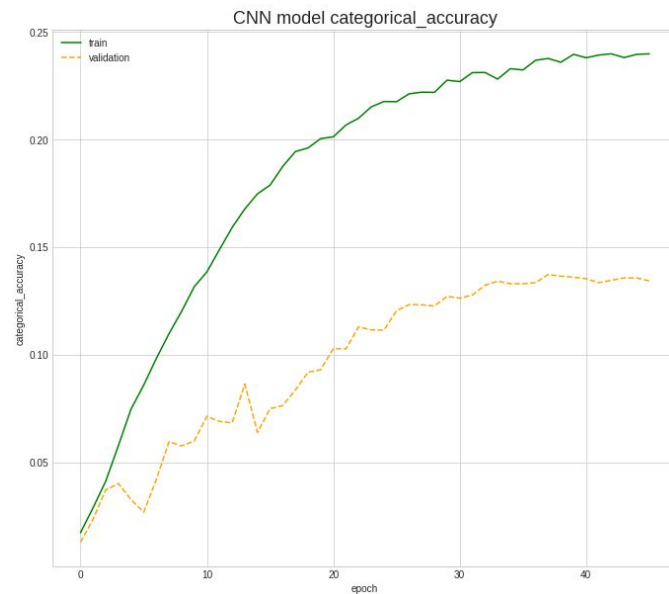
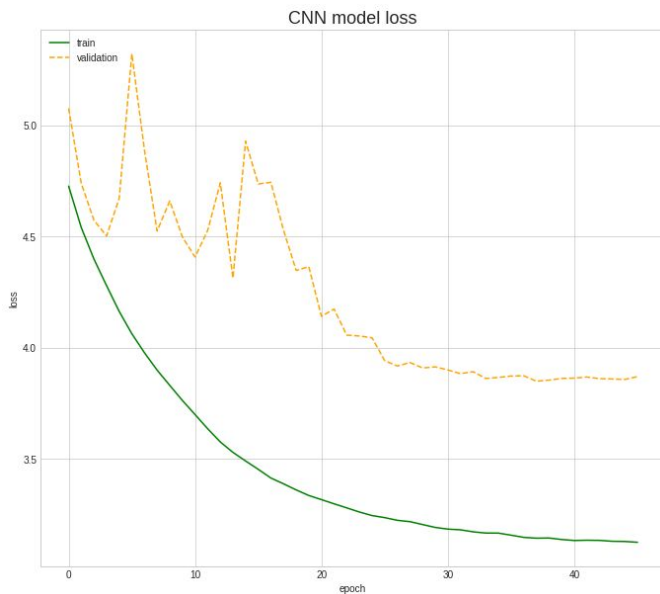
On optimize cnn3 en ajoutant des éléments:

- Dropout en sortie des couches de convolution : permet d'ajouter du 'bruit' et donc d'éviter l'over fitting
- Normalisation
- Couches d'activation en sortie des couches cachées
- Suppression de la couche de flattening

Layer (type)	Output Shape	Param #
rescaling_4 (Rescaling)	(None, 299, 299, 3)	0
conv2d_12 (Conv2D)	(None, 299, 299, 16)	448
batch_normalization_12 (Batch Normalization)	(None, 299, 299, 16)	48
activation_6 (Activation)	(None, 299, 299, 16)	0
max_pooling2d_8 (MaxPooling2D)	(None, 75, 75, 16)	0
dropout_6 (Dropout)	(None, 75, 75, 16)	0
conv2d_13 (Conv2D)	(None, 75, 75, 32)	4640
batch_normalization_13 (Batch Normalization)	(None, 75, 75, 32)	96
activation_7 (Activation)	(None, 75, 75, 32)	0
max_pooling2d_9 (MaxPooling2D)	(None, 37, 37, 32)	0
dropout_7 (Dropout)	(None, 37, 37, 32)	0
conv2d_14 (Conv2D)	(None, 37, 37, 64)	18496
batch_normalization_14 (Batch Normalization)	(None, 37, 37, 64)	192
global_average_pooling2d_4 (GlobalAveragePooling2D)	(None, 64)	0
dense_8 (Dense)	(None, 128)	8320
dense_9 (Dense)	(None, 120)	15480
Total params: 47,720		
Trainable params: 47,496		
Non-trainable params: 224		

# cnn3v2

Cette fois, notre modèle apprend, même si la performance est encore assez faible (25% d'accuracy)



# Optimisation des hyper parametres de cnn3v2

On va encore plus loin en optimisant certains parametres du modèle cnn3v2

- taille des filtres de convolution
- ajout ou non de Dropout

On utilise pour cela l'api `Keras Tuner`

- sur la phase de recherches des meilleurs hyper parametres, on utilise qu'une partie du dataset, afin de réduire le temps de calcul

# cnn3v2 optimisé - TensorBoard & Best Model

TensorBoard permet de voir les différents scénarios de keras tuner et leur impact

Le meilleur modèle trouvé est le suivant



Layer (type)	Output Shape	Param #
rescaling_2 (Rescaling)	(None, 299, 299, 3)	0
conv2d_6 (Conv2D)	(None, 299, 299, 64)	1792
batch_normalization_6 (Batch Normalization)	(None, 299, 299, 64)	192
activation_4 (Activation)	(None, 299, 299, 64)	0
max_pooling2d_4 (MaxPooling2D)	(None, 75, 75, 64)	0
dropout_4 (Dropout)	(None, 75, 75, 64)	0
conv2d_7 (Conv2D)	(None, 75, 75, 64)	36928
batch_normalization_7 (Batch Normalization)	(None, 75, 75, 64)	192
activation_5 (Activation)	(None, 75, 75, 64)	0
max_pooling2d_5 (MaxPooling2D)	(None, 37, 37, 64)	0
dropout_5 (Dropout)	(None, 37, 37, 64)	0
conv2d_8 (Conv2D)	(None, 37, 37, 64)	36928
batch_normalization_8 (Batch Normalization)	(None, 37, 37, 64)	192
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 64)	0
dense_4 (Dense)	(None, 128)	8320
dense_5 (Dense)	(None, 120)	15480

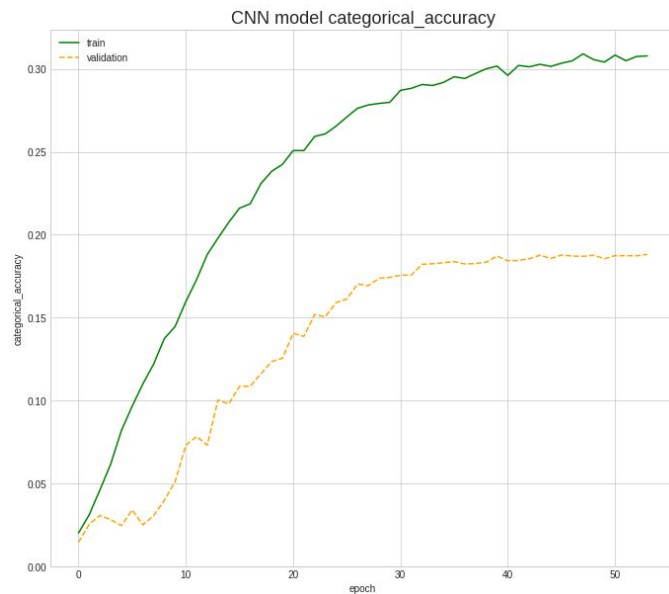
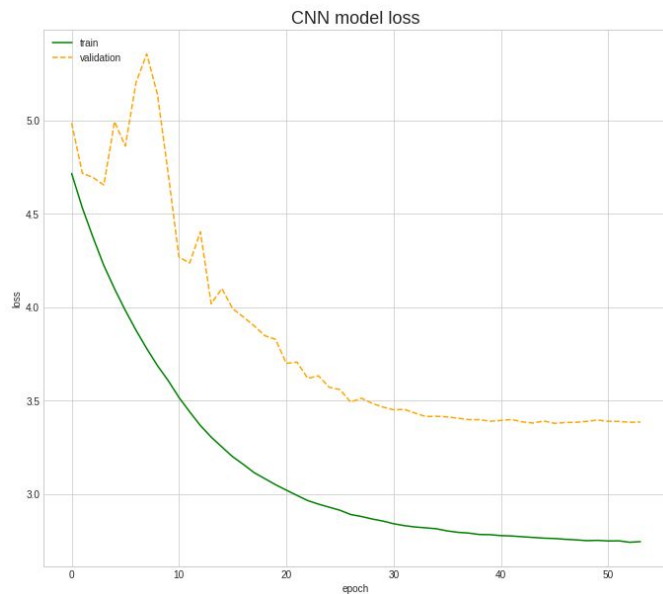
=====  
Total params: 100,024  
Trainable params: 99,640  
Non-trainable params: 384



# cnn3v2 optimisé

Le modèle est plus performant, on obtient un gain d'accuracy (30%).

Malgré tout, la structure restant très simple, les performances brutes sont assez faibles



# Transfer Learning

# Transfer Learning

Le Transfer Learning permet de réutiliser des réseaux profonds pré entraîné, et les adapter à un probleme spécifique.

Dans notre cas, on va réutiliser des modèles CNN entraînés sur le dataset ImageNet

L'API Keras propose un grand nombre de modèles, par la suite on choisi ResNet50V2 et Xception

# Transfer Learning

Le principe du Transfer Learning est

- utiliser un modèle
- remplacer la couche supérieure (TOP)
- ajouter éventuellement quelques couches de convolution
- modifier éventuellement l'input shape

# Résultats ResNet50V2

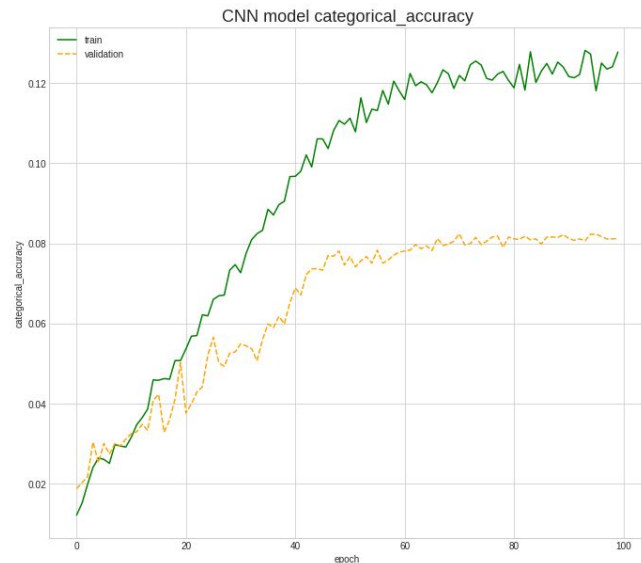
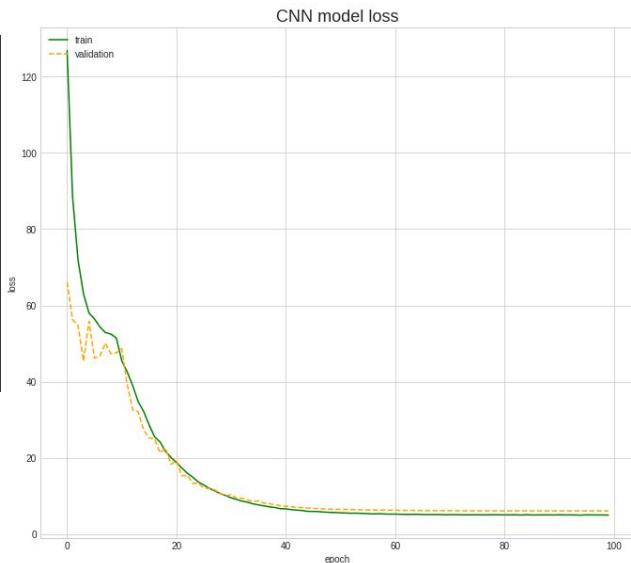
Le modèle arrête de progresser à l'époch 40 et n'est pas performant (12% accuracy)

Il n'est pas très performant, peut être parce que:

- l'input shape serait à revoir
- les couches ajoutées ne sont pas pertinentes

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[(None, 299, 299, 3)]	0
sequential (Sequential)	(None, 299, 299, 3)	0
resnet50v2 (Functional)	(None, 10, 10, 2048)	23564800
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 2048)	0
dropout_1 (Dropout)	(None, 2048)	0
dense_2 (Dense)	(None, 120)	245880

=====  
Total params: 23,810,680  
Trainable params: 245,880  
Non-trainable params: 23,564,800

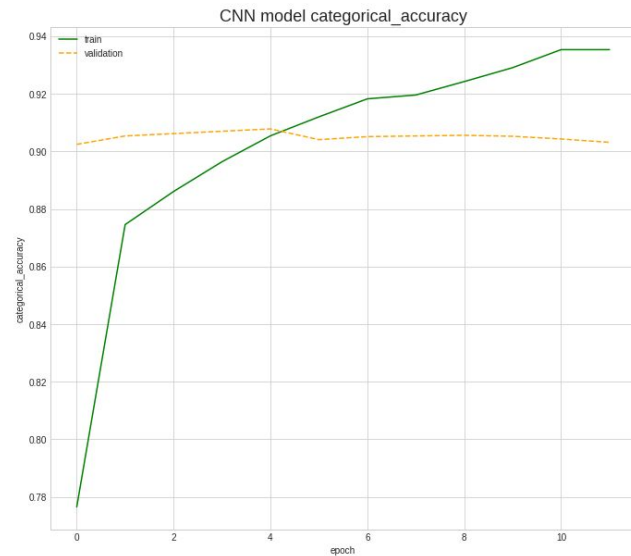
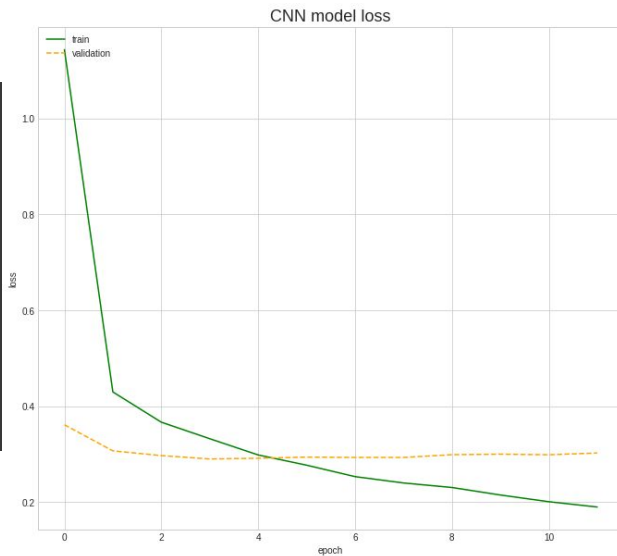


# Résultats Xception

Ce modèle est capable d'apprendre très rapidement et obtient un très bon niveau de performance (> 93% accuracy)

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 299, 299, 3)]	0
sequential (Sequential)	(None, 299, 299, 3)	0
rescaling (Rescaling)	(None, 299, 299, 3)	0
xception (Functional)	(None, 10, 10, 2048)	20861480
global average pooling2d (GlobalAveragePooling2D)	(None, 2048)	0
dropout (Dropout)	(None, 2048)	0
dense (Dense)	(None, 120)	245880

=====  
Total params: 21,107,360  
Trainable params: 245,880  
Non-trainable params: 20,861,480



# Vérification Xception

Certaines prédictions sont fausses, en vérifiant les résultats on comprend facilement que la distinction est ténue.

Par exemple le modèle 'confonds' parfois ces couples de races:





# Application



# Application

On utilise Gradio, qui permet de rapidement proposer une interface

