MDI341 Data Challenge

Rapport

Introduction

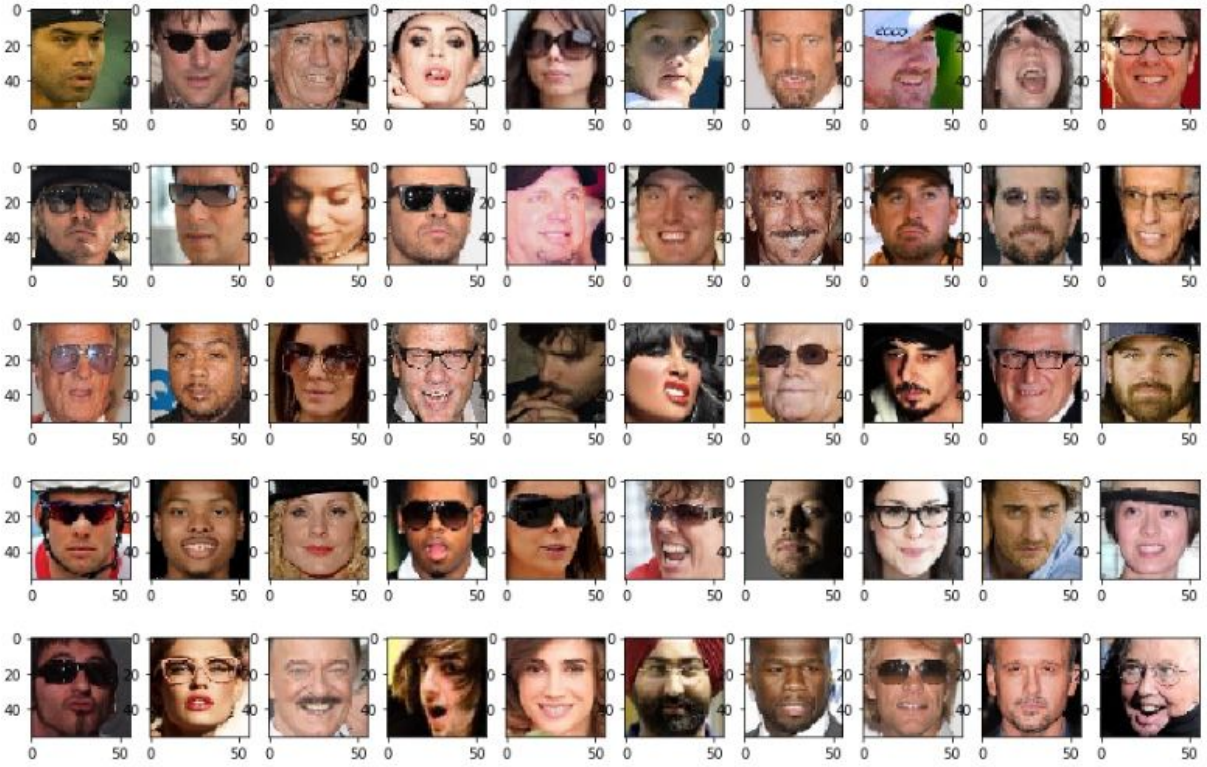
Tout comme pour le challenge précédent, ma démarche a débuté par un examen préalable des données et une recherche sur le web des ressources et/ou travaux relatifs à la problématique qui nous avait été initialement posée, à savoir la détection de moustaches sur des photos de visages. Dans ce cadre, j'avais trouvé un article proposant une démarche intéressante pour aborder ce problème :

- Jian-Gang Wang and Wei-Yun Yau, *"Real-time moustache detection by combining image decolorization and texture detection with applications to facial gender recognition"*, Machine Vision and Applications 25, 2014.

Malheureusement, le sujet ayant été modifié en cours de route, je n'ai pas pu exploiter la méthodologie proposée. Par cohérence dans ma démarche sur ce challenge, j'ai donc examiné plusieurs échantillons des photos qui nous avaient été fournies afin d'avoir une intuition sur la nouvelle problématique posée qui ne nous a pas été révélée.

Images classées positives

Jean-Marc SEVIN - 02/05/2018



Images classées négatives

L'hypothèse la plus probable semble être la détection d'occlusions sur des visages (moustache, barbe, lunettes, lunettes de soleil, casquette, foulard…) dans le cadre de la reconnaissance faciale. J'ai donc effectué de nouvelles recherches sur le web. Voici la liste des sources qui m'ont été utiles :

1. Yizhang Xia, Bailing Zhang and Frans Coenen, *"Face Occlusion Based on Multi-task Convolutional Neural Network"*, Fuzzy Systems and Knowledge Discovery (FSKD), 2015.
2. Yizhang Xia, Bailing Zhang and Frans Coenen, *"Face Occlusion Detection Using Deep Convolutional Neural Networks"*, International Journal of Pattern Recognition and Artificial Intelligence 30, 2016

Ces dernières années, les publications en *computer vision* rapportent essentiellement des approches fondées sur l'apprentissage profond, utilisant particulièrement des *Convolutional Neural Networks* (CNN). Par conséquent, j'ai choisi de suivre cette tendance en testant plusieurs architectures de *deep learning* afin de trouver celle qui obtiendrait les meilleures performances.

Recherche de modèles

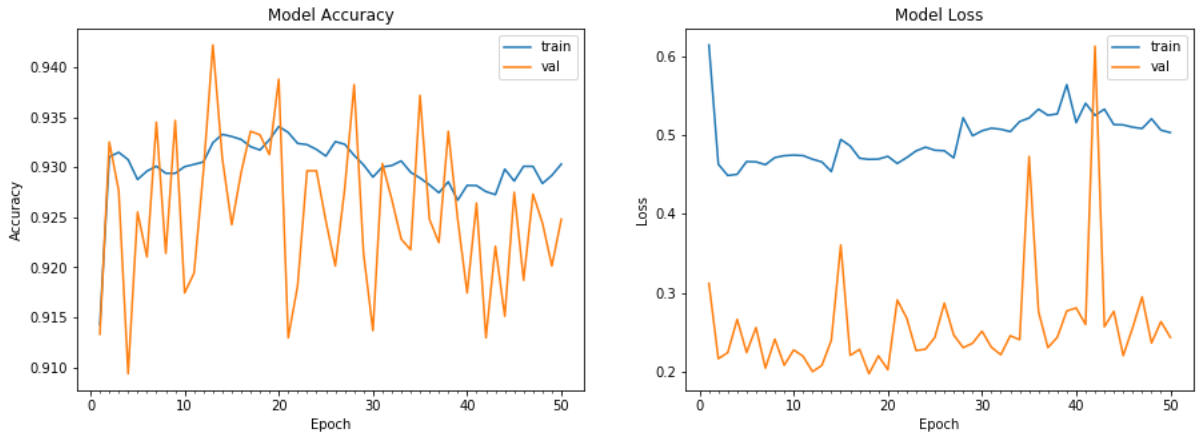
Notons tout d'abord que l'approche proposée dans l'énoncé, basée sur une régression logistique, obtenait un score de 0.6815 sur les données d'entraînement. C'est donc ce score qui nous servira de référence pour évaluer les différents modèles.

LeNet

Pour mon premier modèle, je me suis basé sur une architecture semblable à celles qui avaient proposées par Yann LeCun dans les années 90 et qui avaient permis d'obtenir d'excellents résultats sur la base de données MNIST.

Le réseau comprend deux couches de convolution 32x3x3 puis une couche de convolution 64x3x3, avec une fonction d'activation ReLU et un *pooling* de 2 sur chaque couche. Le bloc *Fully Connected* (FC) comporte une première couche de 64 neurones activés par une fonction ReLU, puis une couche de sortie avec un seul neurone activé par une fonction sigmoïde. La régularisation s'effectue à l'aide d'un *Dropout* de 0.5 sur ce bloc FC.

On optimise la fonction de coût *binary_crossentropy* à l'aide de *RMSProp*, sur 50 epochs, avec une taille de batch de 64. Les courbes d'apprentissage sont présentées ci-dessous :



En ce qui concerne le score, ce modèle obtient 0.7661 sur les données d'entraînement et 0.7779 sur le *leaderboard*.

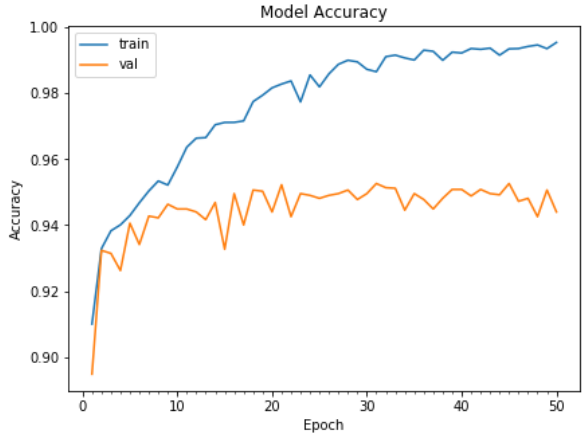
CNN 2015

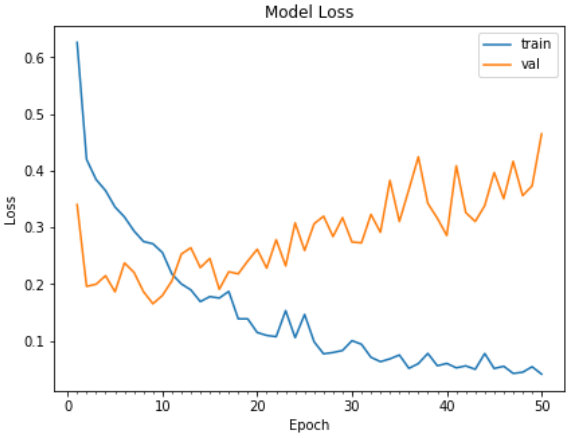
Pour le deuxième modèle, je me suis inspiré de la publication [1] datant de 2015. Celle-ci propose une architecture plus complexe, notamment pour le bloc FC.

Le réseau comprend cette fois une première couche de traitement qui transforme l'image RVB en niveaux de gris. Suivent une première couche de convolution 8x5x5, une deuxième couche de convolution 16x5x5 et une troisième couche de convolution 32x5x5, toutes les trois avec une fonction d'activation ReLU, une *Batch Normalization* et un *Dropout* de 0.25 ; la deuxième et la troisième couche comportent en plus un *pooling* de 2.

Le bloc FC quant à lui comporte cinq couches successives, chacune activée par une fonction ReLU : 1024 neurones, 128 neurones, 22 neurones, 128 neurones et 16 neurones. Enfin, la couche de sortie comporte deux neurones, chacun activé par une fonction softmax.

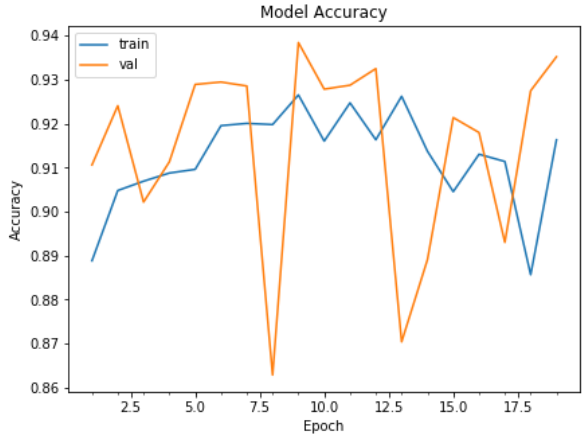
On optimise la fonction de coût *categorical_crossentropy* à l'aide d'*Adam*, sur 50 epochs, avec une taille de batch de 64. La validation s'effectue sur 5% des données et l'on indique au modèle que les classes sont déséquilibrées selon un ratio 14:86. Les courbes d'apprentissage sont présentées ci-dessous :

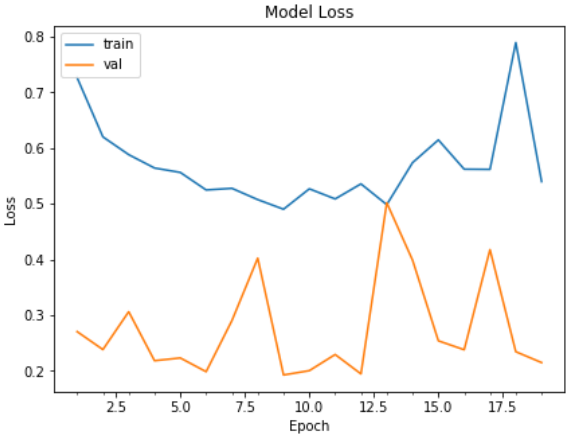




En ce qui concerne le score, ce modèle obtient 0.9812 sur les données d'entraînement, mais sur le leaderboard, on redescend à 0.8332 ; on se trouve clairement dans un cas de surapprentissage.

Du coup, afin que le modèle généralise mieux, j'ai procédé à une augmentation des données en réalisant sur les données d'entraînement une symétrie verticale, une translation verticale/horizontale de 10% et/ou un zoom/dézoom de 10%. Le modèle est ensuite entraîné en suivant sa progression, afin d'effectuer un *early stopping* si nécessaire, sauvegardant à chaque itération la version du modèle si celle-ci s'avère plus performante que les précédentes. Les courbes d'apprentissage sont présentées ci-dessous :





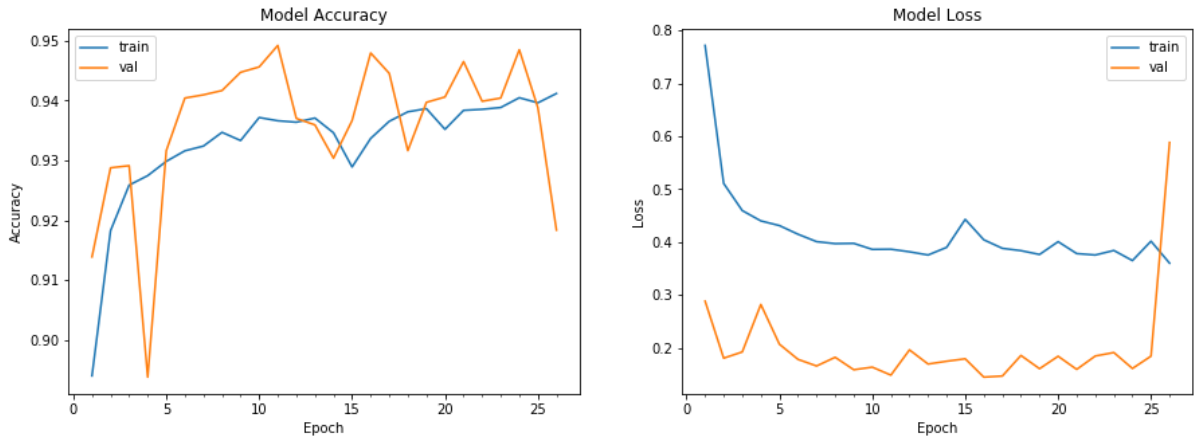
CNN 2016

Pour le troisième modèle, je me suis inspiré de la publication [2] datant de 2016. Celle-ci propose une architecture un peu plus légère que celle du deuxième modèle, mais elle se base cette fois sur des images RVB.

Le réseau comprend une première couche de convolution 8x5x5, une deuxième couche de convolution 16x5x5 et une troisième couche de convolution 32x5x5, toutes les trois avec une fonction d'activation ReLU, une *Batch Normalization* et un *Dropout* de 0.25 ; la deuxième et la troisième couche comportent en plus un *pooling* de 2.

Le bloc FC comporte quatre couches successives, chacune activée par une fonction ReLU : 1024 neurones, 128 neurones, 128 neurones et 16 neurones. La couche de sortie comporte deux neurones, chacun activé par une fonction softmax.

Tout comme pour le deuxième modèle, on optimise la fonction de coût *categorical_crossentropy* à l'aide d'*Adam*, sur 50 epochs, avec une taille de batch de 64. La validation s'effectue sur 5% des données et l'on indique toujours au modèle que les classes sont déséquilibrées selon un ratio 14:86. Cependant, on procède tout de suite à une augmentation des données selon la même méthode que celle employée pour le deuxième modèle, ainsi qu'à un *early stopping*. Les courbes d'apprentissage sont présentées ci-dessous :



En ce qui concerne le score, ce modèle obtient 0.8131 sur les données d'entraînement et 0.8145 sur le *leaderboard*.

Custom CNN

Sur la base de ces premières tentatives guidées, j'ai testé plusieurs variations personnelles.

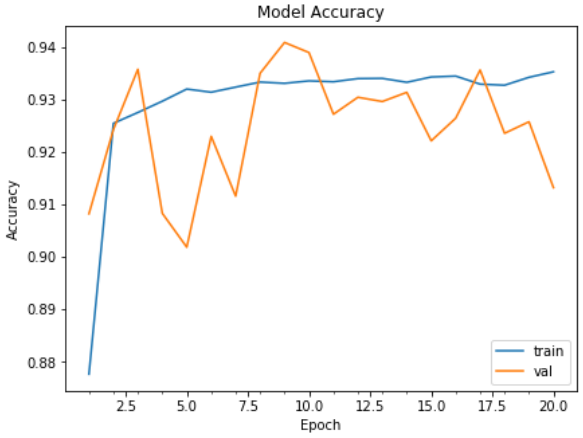
Custom CNN v1

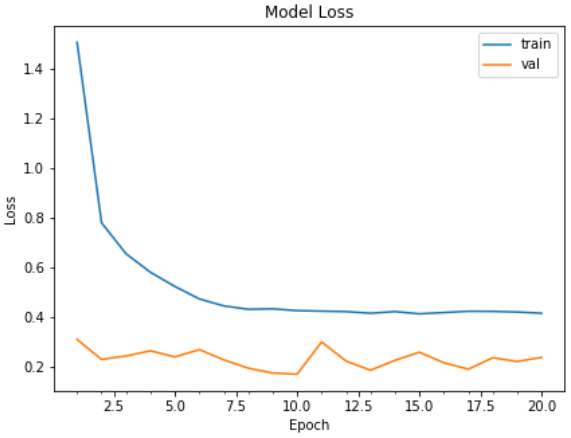
Ma première tentative se base sur un réseau comprenant trois blocs de convolution, chacun comprenant deux couches de convolution successives :

- CONV 8x3x3 → ReLU → CONV 8x3x3 → ReLU → Dropout 0.25
- CONV 16x3x3 → ReLU → CONV 16x3x3 → ReLU → MaxPool 2 → Dropout 0.25
- CONV 32x3x3 → ReLU → CONV 32x3x3 → ReLU → MaxPool 2 → Dropout 0.25

Le bloc FC quant à lui comporte trois couches successives, chacune activée par une fonction ReLU : 1024 neurones, 128 neurones et 16 neurones. Un *Dropout* de 0.5 et une couche de sortie comportant deux neurones activés par une fonction sigmoïde complète le réseau.

Cette fois, on optimise la fonction de coût *binary_crossentropy* à l'aide de *RMSProp*, sur 50 epochs, avec un *early stopping* et une taille de batch de 256. La validation s'effectue sur 10% des données et l'on indique toujours au modèle que les classes sont déséquilibrées selon un ratio 14:86. Les courbes d'apprentissage sont présentées ci-dessous :

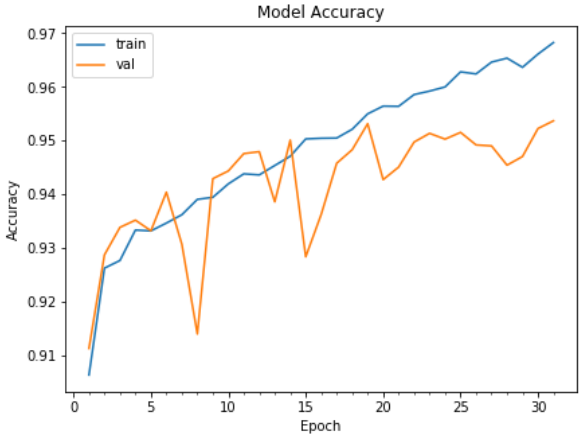


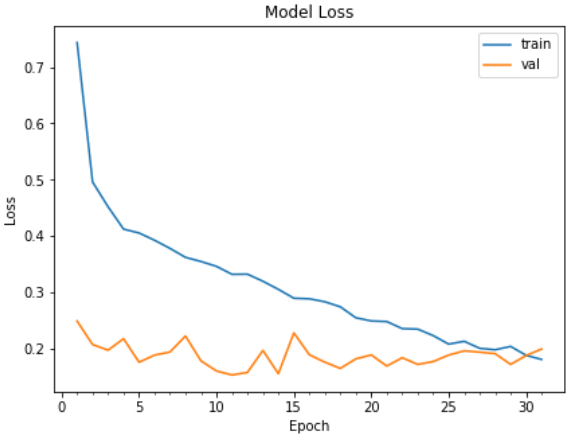


Ce modèle obtient un score de 0.7003 sur les données d'entraînement et 0.7042 sur le *leaderboard*.

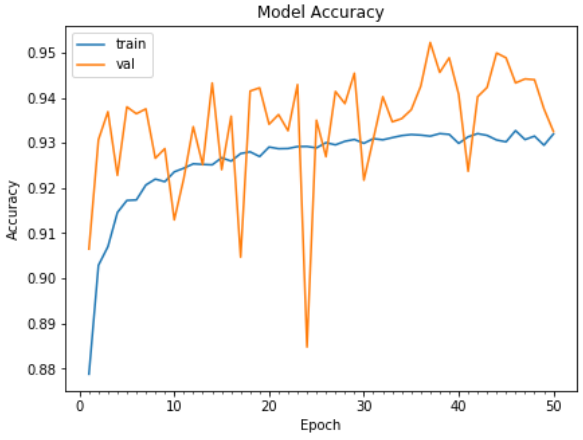
Custom CNN v2

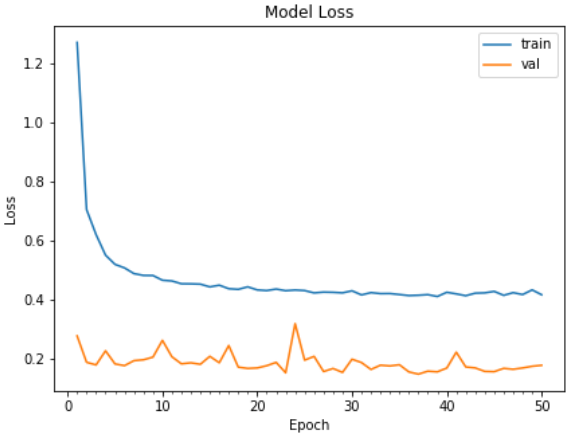
Ensuite, j'ai testé la même architecture en optimisant cette fois la fonction de coût *binary_crossentropy* à l'aide d'*Adam*, sur 50 epochs, avec un *early stopping* et une taille de batch de 64. La validation s'effectue sur 5% des données et l'on indique toujours au modèle que les classes sont déséquilibrées selon un ratio 14:86. Le modèle a été testé avec et sans augmentation des données. Les courbes d'apprentissage sont présentées ci-dessous :





Sans augmentation des données





Avec augmentation des données

Sans augmentation des données, ce modèle obtient un score de 0.8280 sur les données d'entraînement et 0.8806 sur le *leaderboard*, tandis qu'avec augmentation des données, il obtient un score de 0.7680 sur les données d'entraînement et 0.7764 sur le *leaderboard*.

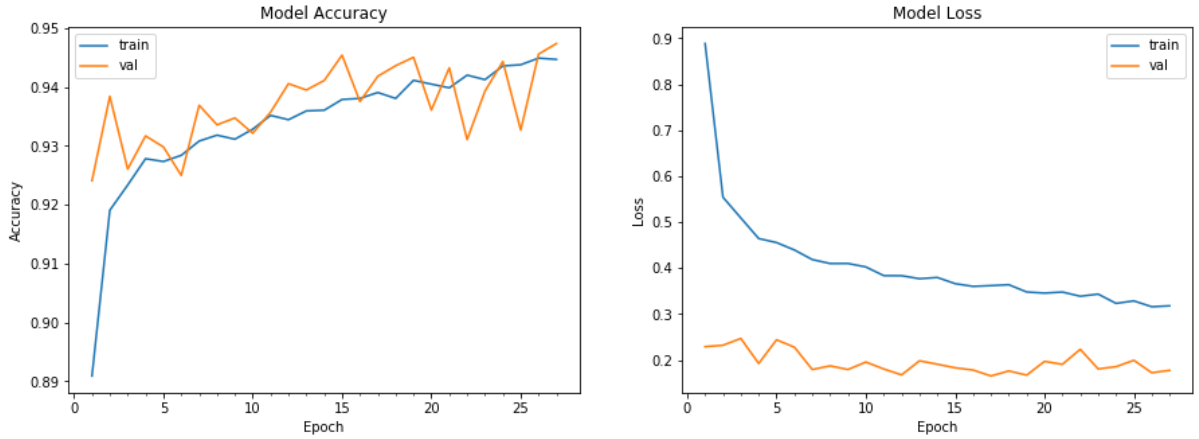
Custom CNN v3

Ma troisième tentative se base encore sur un réseau comprenant trois blocs de convolution, mais chacun comprend désormais trois couches de convolution successives :

- CONV 8x7x7 → ReLU → CONV 8x5x5 → ReLU → CONV 8x3x3 → ReLU→ Dropout 0.25
- CONV 16x7x7 → ReLU → CONV 16x5x5 → ReLU → CONV 16x3x3 → ReLU → MaxPool 2 → Dropout 0.25
- CONV 32x7x7 → ReLU → CONV 32x5x5 → ReLU → CONV 32x3x3 → ReLU → MaxPool 2 → Dropout 0.25

Le bloc FC reste identique aux tentatives précédentes.

Cette fois, on optimise la fonction de coût *binary_crossentropy* à l'aide d'*Adam*, sur 50 epochs, avec un *early stopping* et une taille de batch de 128. La validation s'effectue sur 5% des données et l'on indique toujours au modèle que les classes sont déséquilibrées selon un ratio 14:86. Les courbes d'apprentissage sont présentées ci-dessous :



Ce modèle obtient un score de 0.8359 sur les données d'entraînement et 0.8317 sur le *leaderboard*.

Conclusion

Dressons maintenant un tableau comparatif de toutes ces expériences.

Architecture	Data augmentation	Loss	Optimizer	Batch size	Score (training)	Score (leaderboard)
LeNet	No	binary_crossentropy	RMSProp	64	0.7661	0.7779
CNN 2015	No	categorical_crossentropy	Adam	64	0.9812	0.8332
CNN 2015	Yes	categorical_crossentropy	Adam	64	0.7766	0.7810
CNN 2016	Yes	categorical_crossentropy	Adam	64	0.8131	0.8145
Custom CNN v1	No	binary_crossentropy	RMSProp	256	0.7003	0.7042
Custom CNN v2	**No**	**binary_crossentropy**	**Adam**	**64**	**0.8280**	**0.8806**
Custom CNN v2	Yes	binary_crossentropy	Adam	64	0.7680	0.7764
Custom CNN v3	No	binary_crossentropy	Adam	128	0.8359	0.8317

Le modèle Custom CNN v2 apparaît donc comme le plus prometteur, bien que l'écart entre son score sur les données d'entraînement et celui sur le *leaderboard* puisse être dû à une distribution plus favorable des données de validation.

Sur les données de test, les scores obtenus sont les suivants :

Modèle	Score
CNN 2015 (with data augmentation)	0.799401156421
CNN 2016 (with data augmentation)	0.82094886148
Custom CNN v1	0.79737044096
Custom CNN v2	0.817562467743
Custom CNN v2 (with data augmentation)	**0.825931730312**
Custom CNN v3	0.793055183875

N.B. Je tiens à préciser que j'ai envisagé d'utiliser des modèles plus complexes, via le *transfer learning* d'architectures connues comme VGG ou DenseNet, mais les versions pré-entraînées de ces modèles disponibles dans Keras nécessitaient en entrée des images de bien meilleure résolution que celles mises à notre disposition. N'ayant pas les ressources de calcul nécessaires, je ne pouvais pas réentraîner totalement ces modèles à partir des données du challenge.

Annexes

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 54, 54, 32)	896
activation_1 (Activation)	(None, 54, 54, 32)	0
max_pooling2d_1 (MaxPooling2	(None, 27, 27, 32)	0
conv2d_2 (Conv2D)	(None, 25, 25, 32)	9248
activation_2 (Activation)	(None, 25, 25, 32)	0
max_pooling2d_2 (MaxPooling2	(None, 12, 12, 32)	0
conv2d_3 (Conv2D)	(None, 10, 10, 64)	18496
activation_3 (Activation)	(None, 10, 10, 64)	0
max_pooling2d_3 (MaxPooling2	(None, 5, 5, 64)	0
flatten_1 (Flatten)	(None, 1600)	0
dense_1 (Dense)	(None, 64)	102464
activation_4 (Activation)	(None, 64)	0
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 1)	65
activation_5 (Activation)	(None, 1)	0

Total params: 131,169
Trainable params: 131,169
Non-trainable params: 0

Architecture LeNet

Layer (type)	Output Shape	Param #
lambda_1 (Lambda)	(None, 56, 56, 1)	0
conv2d_1 (Conv2D)	(None, 56, 56, 8)	208
batch_normalization_1 (Batch	(None, 56, 56, 8)	32
dropout_1 (Dropout)	(None, 56, 56, 8)	0
conv2d_2 (Conv2D)	(None, 56, 56, 16)	3216
batch_normalization_2 (Batch	(None, 56, 56, 16)	64
max_pooling2d_1 (MaxPooling2	(None, 28, 28, 16)	0
dropout_2 (Dropout)	(None, 28, 28, 16)	0
conv2d_3 (Conv2D)	(None, 28, 28, 32)	12832
batch_normalization_3 (Batch	(None, 28, 28, 32)	128
max_pooling2d_2 (MaxPooling2	(None, 14, 14, 32)	0
dropout_3 (Dropout)	(None, 14, 14, 32)	0
flatten_1 (Flatten)	(None, 6272)	0
dense_1 (Dense)	(None, 1024)	6423552
dense_2 (Dense)	(None, 128)	131200
dense_3 (Dense)	(None, 22)	2838
dense_4 (Dense)	(None, 128)	2944
dense_5 (Dense)	(None, 16)	2064
dense_6 (Dense)	(None, 2)	34

Total params: 6,579,112
Trainable params: 6,579,000
Non-trainable params: 112

Architecture CNN 2015

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 56, 56, 8)	608
batch_normalization_1 (Batch	(None, 56, 56, 8)	32
dropout_1 (Dropout)	(None, 56, 56, 8)	0
conv2d_2 (Conv2D)	(None, 56, 56, 16)	3216
batch_normalization_2 (Batch	(None, 56, 56, 16)	64
max_pooling2d_1 (MaxPooling2	(None, 28, 28, 16)	0
dropout_2 (Dropout)	(None, 28, 28, 16)	0
conv2d_3 (Conv2D)	(None, 28, 28, 32)	12832
batch_normalization_3 (Batch	(None, 28, 28, 32)	128
max_pooling2d_2 (MaxPooling2	(None, 14, 14, 32)	0
dropout_3 (Dropout)	(None, 14, 14, 32)	0
flatten_1 (Flatten)	(None, 6272)	0
dense_1 (Dense)	(None, 1024)	6423552
dense_2 (Dense)	(None, 128)	131200
dense_3 (Dense)	(None, 128)	16512
dense_4 (Dense)	(None, 16)	2064
dense_5 (Dense)	(None, 2)	34

Total params: 6,590,242
Trainable params: 6,590,130
Non-trainable params: 112

Architecture CNN 2016

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 56, 56, 8)	224
conv2d_2 (Conv2D)	(None, 54, 54, 8)	584
dropout_1 (Dropout)	(None, 54, 54, 8)	0
conv2d_3 (Conv2D)	(None, 54, 54, 16)	1168
conv2d_4 (Conv2D)	(None, 52, 52, 16)	2320
max_pooling2d_1 (MaxPooling2	(None, 26, 26, 16)	0
dropout_2 (Dropout)	(None, 26, 26, 16)	0
conv2d_5 (Conv2D)	(None, 26, 26, 32)	4640
conv2d_6 (Conv2D)	(None, 24, 24, 32)	9248
max_pooling2d_2 (MaxPooling2	(None, 12, 12, 32)	0
dropout_3 (Dropout)	(None, 12, 12, 32)	0
flatten_1 (Flatten)	(None, 4608)	0
dense_1 (Dense)	(None, 1024)	4719616
dense_2 (Dense)	(None, 128)	131200
dense_3 (Dense)	(None, 16)	2064
dropout_4 (Dropout)	(None, 16)	0
dense_4 (Dense)	(None, 2)	34

Total params: 4,871,098
Trainable params: 4,871,098
Non-trainable params: 0

Architecture Custom CNN v1 & v2

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 56, 56, 8)	1184
conv2d_2 (Conv2D)	(None, 52, 52, 8)	1608
conv2d_3 (Conv2D)	(None, 50, 50, 8)	584
dropout_1 (Dropout)	(None, 50, 50, 8)	0
conv2d_4 (Conv2D)	(None, 50, 50, 16)	6288
conv2d_5 (Conv2D)	(None, 50, 50, 16)	6416
conv2d_6 (Conv2D)	(None, 48, 48, 16)	2320
max_pooling2d_1 (MaxPooling2	(None, 24, 24, 16)	0
dropout_2 (Dropout)	(None, 24, 24, 16)	0
conv2d_7 (Conv2D)	(None, 24, 24, 32)	25120
conv2d_8 (Conv2D)	(None, 24, 24, 32)	25632
conv2d_9 (Conv2D)	(None, 22, 22, 32)	9248
max_pooling2d_2 (MaxPooling2	(None, 11, 11, 32)	0
dropout_3 (Dropout)	(None, 11, 11, 32)	0
flatten_1 (Flatten)	(None, 3872)	0
dense_1 (Dense)	(None, 1024)	3965952
dense_2 (Dense)	(None, 128)	131200
dense_3 (Dense)	(None, 16)	2064
dropout_4 (Dropout)	(None, 16)	0
dense_4 (Dense)	(None, 2)	34

Total params: 4,177,650
Trainable params: 4,177,650
Non-trainable params: 0

Architecture Custom CNN v3