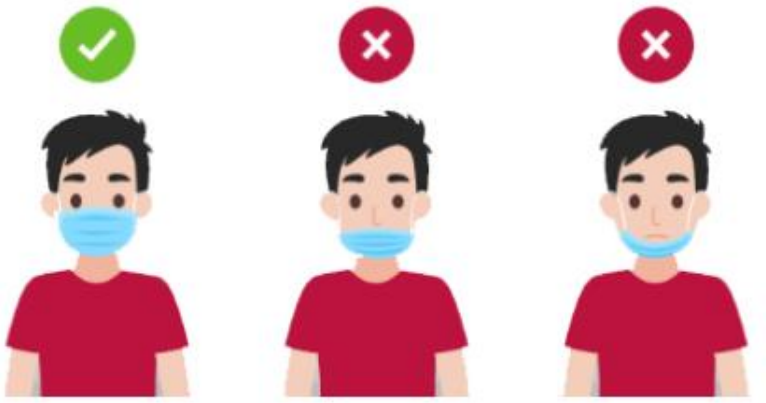
Boersma Hélène
Courtois Quentin
Jaouen Guillaume

Deep Learning

Détection du port du masque sur le visage



Introduction & Contexte

Avec la pandémie de COVID-19, la plupart des pays du monde ont exigé à la population le port du masque chrirgical dans les transports ou dans la rue. Cependant, de nombreuses personnes ne le portent pas bien. Un article français évoquait la possibilité que la mairie de Paris mette en place un système de vidéo surveillance dans les transports publics pour avertir les personnes qui ne respectent pas les consignes sanitaires (vous pouvez trouver l'article ici).
Notre projet consiste à analyser des centaines de photos de personnes avec des personnes portant le masque. Nous avons trois catégories :

- Les personnes qui portent bien le masque
- Les personnes qui ne portent pas de masque
- Les personnes qui ne portent pas le masque correctement

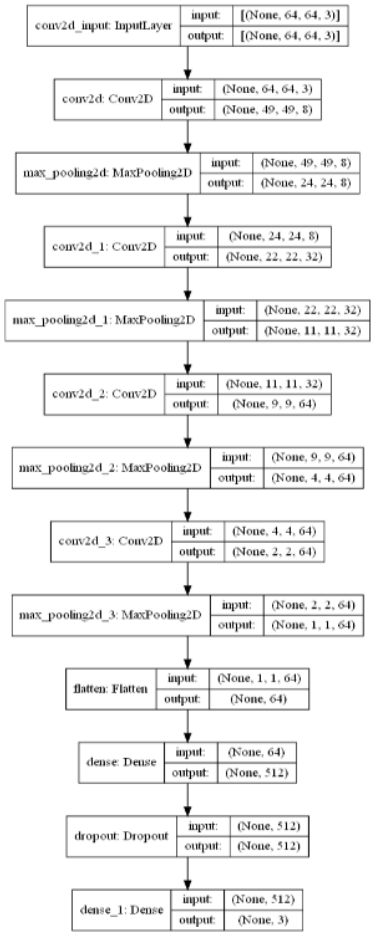
Nous avons un dataset avec plusieurs photos sur lesquelles peuvent figurer une ou plusieurs personnes. A partir des fichiers xml, nous avons utilisé la classe nommée "bndbox», qui nous a permis de recadrer la tête des personnes sur une seule image et avec des dimensions choisies.
En ce qui concerne les images du dataset, nous disposons d'un jeu de données de plus de 4000 images, dont 3000 images avec un bon port du masque, 700 sans masque et 150 images où le masque n'est pas bien posé sur le visage.

Nous avons conscience que nous aurons probablement des valeurs aberrantes en raison de l'inégalité du nombre d'images appartenant aux trois catégories. Une option à explorer pourrait être de réduire cette inégalité en ajustant le nombre d'images aux catégories, par exemple en augmentant les données pour certaines catégories (c'est-à-dire un masque mal porté).

Expérience

- Notre modèle

Concernant notre modèle, nous avons utilisé un réseau neuronal convolutif de base décrit comme tel:



Notre modèle initial et se composait de 4 blocs convolutifs avec un nombre croissant de filtres suivis d'une couche de Max Pooling réduisant la taille de l'image ou de la feature map par 2 à chaque fois.

Une fois que nos données ont traversé l'entièreté des couches convolutionelles, on aplatit les données de la dernière feature map pour la faire rentrer dans un réseau de neurones classiques avec comme entrée 1164 neurones. Nous rajoutons aussi des couches de Dropout pour réduire l'overfitting avant de prédire les différentes catégories voulues.

En raison des limitations de temps, nous n'avons pas pu tester toutes les combinaisons possibles. Ainsi, pour les expériences générales telles que les blocs convolutifs et les formes d'images, nous avons choisi d'utiliser l'optimiseur le plus basique, à savoir SGD avec des paramètres par défaut. Pour les optimiseurs, nous avons testé avec des formes de (64, 64, 3). Pour Adam, nous avons laissé la valeur par défaut $\beta\beta 2$ = 0.999 pour toutes les expériences.

Pour agrémenter ce projet Kaggle, nous avons décidé de garder toujours le même modèle lors de nos expériences mais en essayant de fine-tuner les différents algorithmes d'optimisation comme vous allez le voir par la suite

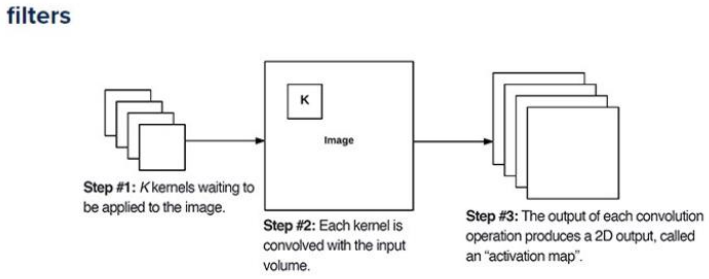


Figure 1: The Keras Conv2D parameter, `filters` determines the number of kernels to convolve with the input volume. Each of these operations produces a 2D activation map.

Cette figure montre comment fonctionne une convolution CONV2D dans Keras

- ### Blocs convolutifs

Pour déterminer combien de blocs nous devrions avoir, nous avons entraîné notre modèle avec des blocs conv 1, 2 et 4. Les différences étaient si faibles que nous avons décidé de garder le modèle avec la quantité minimale de paramètres d'entraînement. Il s'agit donc du modèle avec le plus de blocs, qui est notre modèle initial (90 000 paramètres contre plus de 2 millions pour un seul bloc).

- ### Shape des images

Pour la shape 32, nous avons dû réduire le nombre de blocs convolutifs car l'image ne peut pas être réduite par 2 si elle est déjà une image 1 par 1. Ainsi, pour 32,32,3, notre modèle ne comportait que 2 blocs.
Globalement, les résultats étaient très similaires et pour des raisons de calcul concernant les shapes 128,128,3 qui augmentent le nombre de paramètres (12 millions) et le temps de calcul des blocs convolutifs (cela double le temps), nous avons décidé de changer notre modèle pour n'avoir qu'un seul bloc convolutif et une shape de 64,64,3.

- ### Optimizers

Pour chaque optimizer, nous avons essayé de réduire au maximum le nombre de sessions d'entraînements afin de ne garder que les résultats les plus intéressants pour l'analyse. L'ensemble des entraînements est résumé dans le tableau ci-dessous :

SGD	Default	alpha=1e-3	a=1e-3, p=0,9	nester, p=0.9
AdaGrad	Default	alpha=1e-2	alpha=1e-5	
RMSProp	Default	rho=0.7	rho=0.99	alpha=1e-5
AdaDelta	Default	rho=0.7	rho=0.99	alpha=1e-5
Adam	Default	bêta1=0,7	bêta1=0,99	a=1e-5, b=0,99

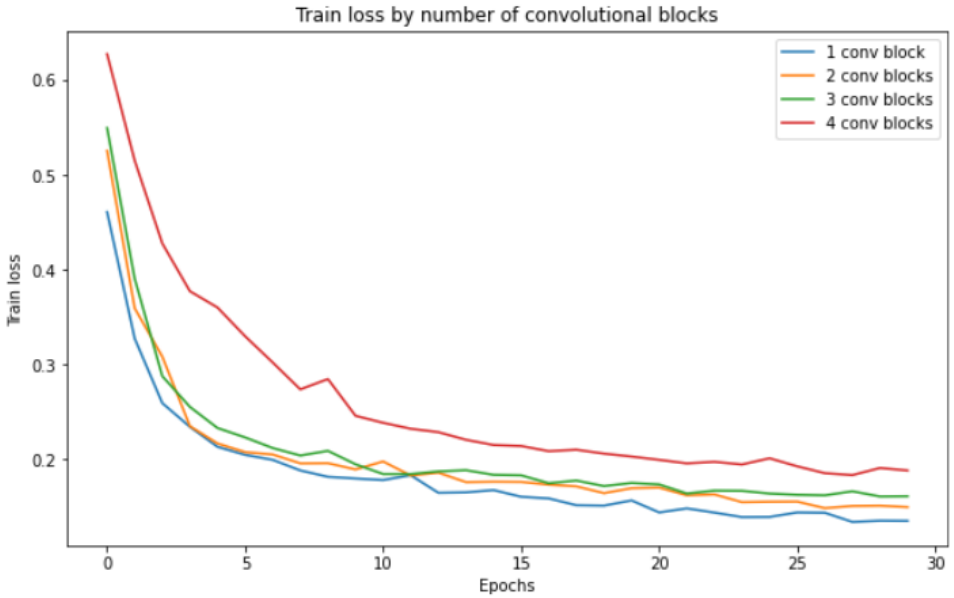
Nous avons décidé de nous entraîner sur 30 époques, car au-delà de la validation, la précision stagne, ce qui fait que le modèle commence à être légèrement overfité.

Maintenant que nous avons défini notre dispositif de formation, passons aux résultats.

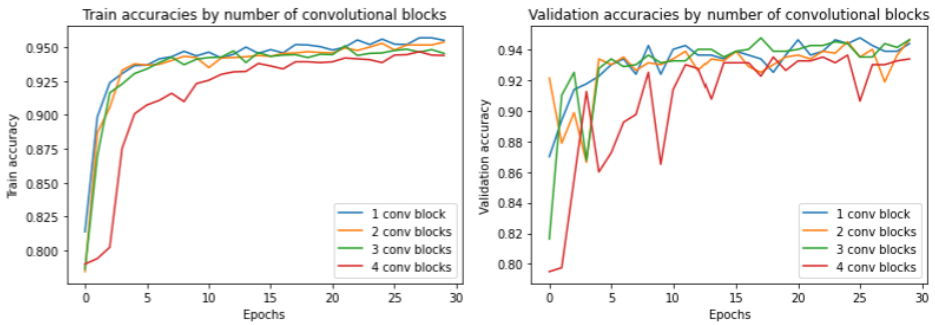
Résultats

- Blocs convolutifs

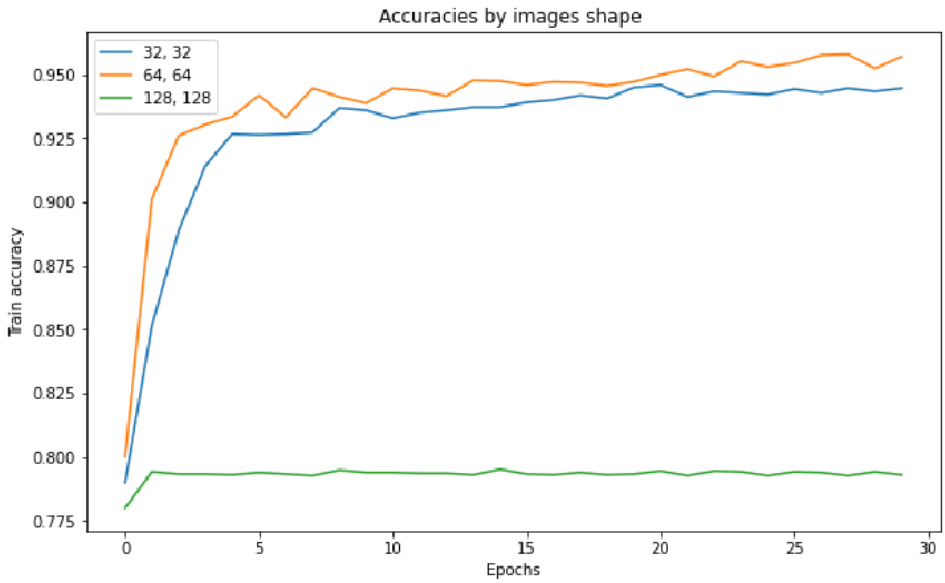
Après l'entraînement avec des modèles contenant 1 à 4 blocs convolutifs, nous obtenons les pertes d'entraînement suivantes. Il semble clair que nos images sont trop petites pour un modèle contenant 4 blocs convolutifs ou plus, alors qu'à partir de 3, l'aspect est similaire. Puisque plus il y a de blocs, moins il y a de paramètres à entraîner. Nous pourrions opter pour un modèle avec trois blocs convolutifs pour des raisons de performance. Néanmoins, nous avons choisi le modèle le plus simple, avec un bloc convolutif. Cela nécessite d'entraîner deux millions de paramètres.



De plus, ceci est confirmé par les accuracy de training et de validation.

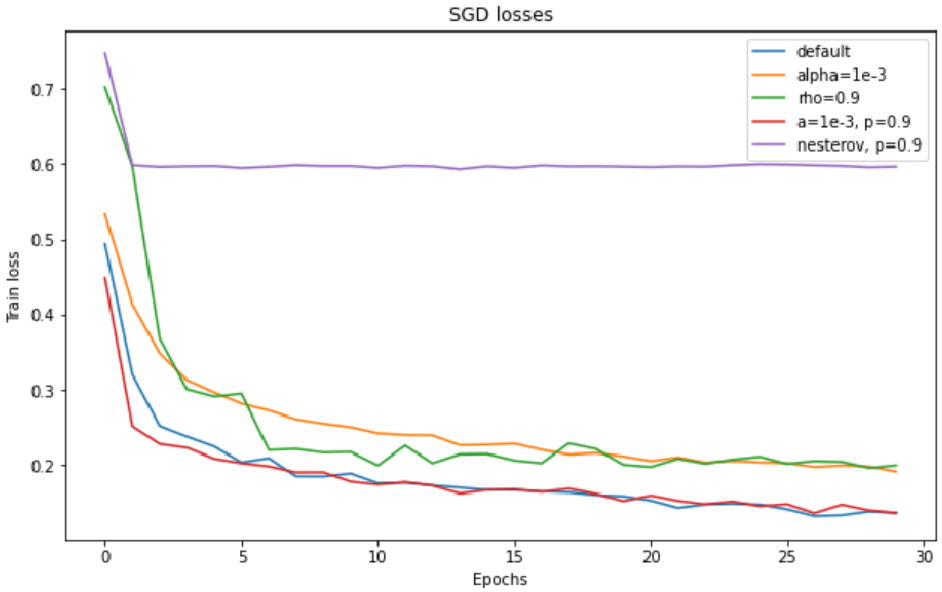


- Image Shape



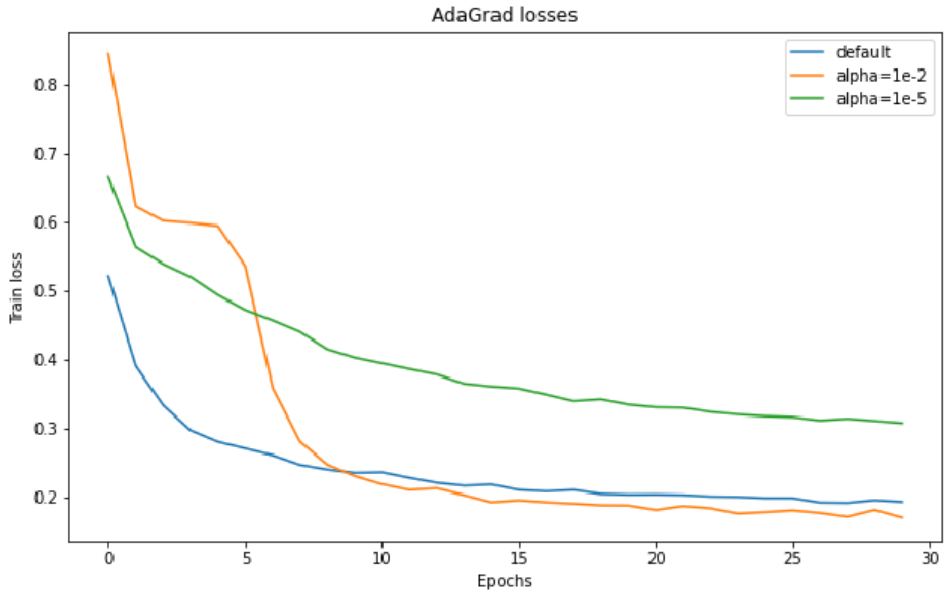
En entraînant notre modèle avec des images de différentes tailles, nous constatons que la précision est toujours meilleure avec des images de 64 plutôt que de 32. Cependant, avec une taille d'image de 128, nous sommes étonnamment bloqués à 79% de précision. Cela peut être dû au fait que le modèle prédit toujours le même résultat, comme nous l'avons vu précédemment, l'ensemble de données est composé à 79% d'images "avec masque".

- Descente de Gradient Stochastique



Pour les expériences avec SGD, nous pouvons clairement voir l'effet du momentum (courbe rouge), qui aide le modèle à converger beaucoup mieux que sans (courbe orange). Nous avons délibérément effectué cette comparaison avec un faible taux d'apprentissage afin de voir la contribution du momentum.
D'autre part, le SGD avec le taux d'apprentissage par défaut et sans momentum converge assez bien, donc théoriquement il devrait converger encore mieux avec le momentum... mais en fait il ne le fait pas. Nous pouvons voir qu'avec le momentum (courbe verte), la perte est bruyante et converge difficilement aussi bien que les autres. Enfin, l'expérience avec l'accélération n'a pas fonctionné, elle est restée bloquée à 79% de précision. Nous le commenterons plus tard.

- AdaGrad

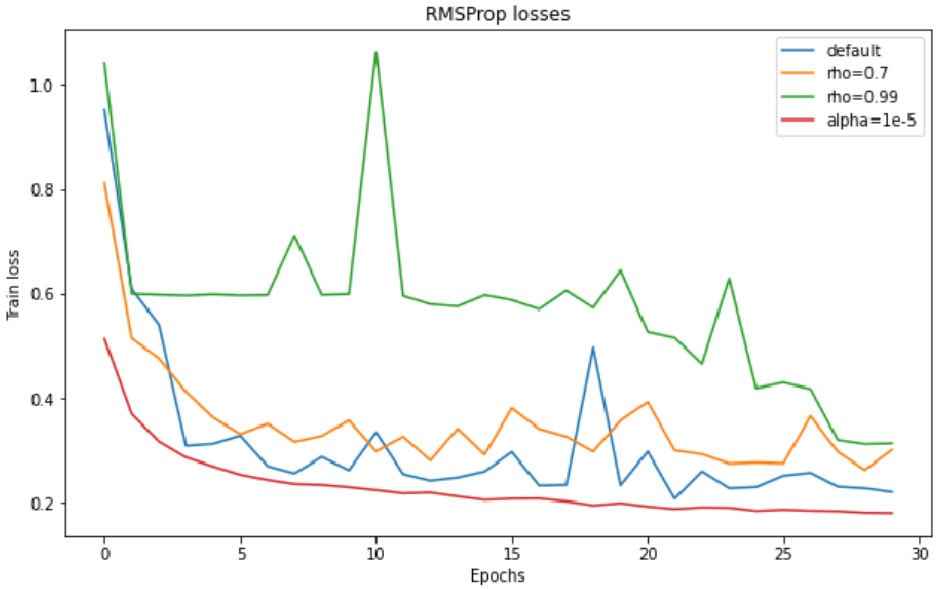


Cette comparaison est intéressante. Nous savons qu'AdaGrad est construit pour avoir un taux d'apprentissage adaptatif pour chaque paramètre. Donc en théorie, il n'y a pas besoin de régler le taux d'apprentissage. Ici, nous voyons que cela fonctionne effectivement lorsque le taux d'apprentissage est plus élevé que la valeur par défaut (0.001), le modèle converge même mieux que celui par défaut. Cependant, lorsque le taux d'apprentissage est trop faible, la convergence est beaucoup plus difficile à atteindre.

En effet, on sait que le taux d'apprentissage de chaque paramètre est divisé à chaque fois par la somme des gradients précédents, d'où la difficulté de convergence si le taux de base est vraiment trop petit.

Un autre point intéressant est que nous pouvons voir que la convergence n'est pas très rapide. Ceci est particulièrement visible sur la courbe verte où elle semble ralentir de plus en plus sans vraiment converger. C'est ce que nous avons décrit théoriquement, si tout va bien avec les optimizers suivants la convergence sera plus rapide.

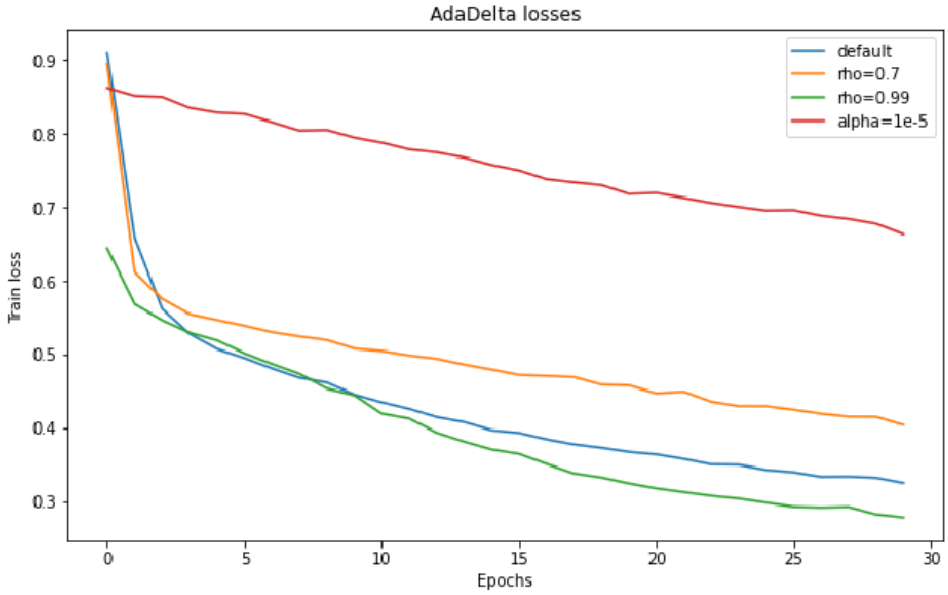
- RMSProp



Comme nous pouvons le voir sur le graphique, RMSProp semble avoir de meilleures performances avec un taux d'apprentissage très faible. Ce comportement pourrait potentiellement venir du fait que, dans ce cas particulier, un faible taux d'apprentissage permet à l'optimizer d'atteindre le minimum plus précisément, sans le dépasser, même si ce taux est adaptatif. En ce qui concerne le taux de decacy, le modèle se comporte assez bien lorsque le taux de decacy est faible, mais comme prévu, les résultats convergent moins rapidement qu'avec la valeur par défaut de 0,9. En effet, c'est l'effet d'AdaGrad qui n'est pas suffisamment atténué. D'autre part, avec un taux de decacy très élevé, ici de 0,99, les résultats sont très mauvais. Encore une fois, il semble que le modèle soit resté bloqué à 79% de précision jusqu'à la vingtième epoch.

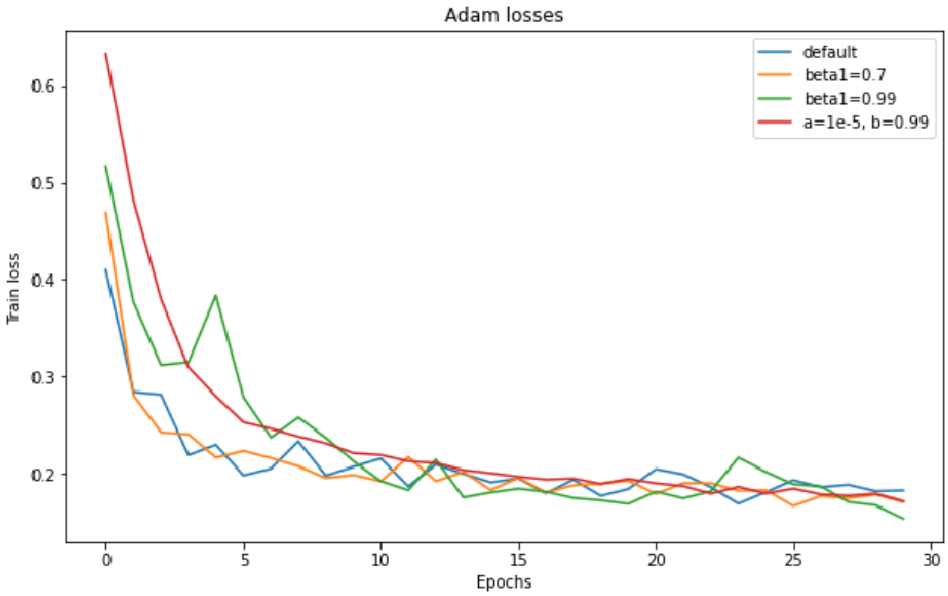
Revenons un instant sur ce dernier problème. Notre modèle est resté bloqué à 79% de précision à de nombreuses reprises pour différents optimizers. Nous pensons que cela est dû au fait que le modèle atteint trop rapidement un minimum, qui serait alors un minimum local, empêchant le modèle d'en sortir. Car bien sûr, le problème réel n'est pas du tout convexe. Néanmoins, et cela devient très intéressant, d'après ce que nous avons compris de ce document de recherche, il a été démontré que les optimiseurs adaptatifs tels que RMSProp finissent par converger vers le minimum global pour les problèmes non convexes. Cependant, cela n'est pas encore clair pour le moment. Pour revenir à notre cas, après la vingtième epoch, l'optimizer représenté par la courbe verte finit toujours par converger vers ce qui semblerait être le minimum global, mais ne termine pas avant la trentième. Cela correspondrait donc à ce qui est montré dans l'article de recherche.

- AdaDelta



Pour AdaDelta, nous avons réalisé exactement les mêmes expériences que pour RMSProp car ils ont un fonctionnement similaire (sachant que les valuers par défaut sont les mêmes pour les deux). Mais précisément, nous voyons que les résultats sont très différents. De plus, les pertes sont presque identiques pour le training et la validation.
Tout d'abord, nous voyons que l'absence de configuration a permis la convergence pour AdaDelta, contrairement aux autres optimiseurs. Nous le verrons plus tard, mais les accuracies sont en réalité assez mauvaises.
Comme prévu, un taux de decacy plus élevé permet au modèle de mieux converger. Ce que nous voyons également, c'est qu'un taux d'apprentissage trop faible fonctionne mal, tout comme avec AdaGrad. RMSProp fonctionne un peu différemment et semble fonctionner beaucoup mieux avec des taux d'apprentissage faibles.

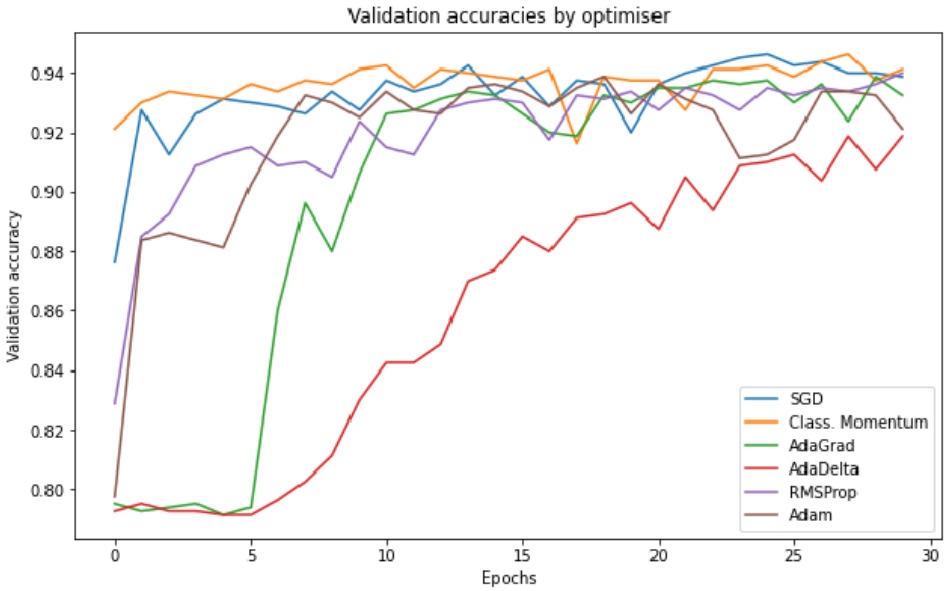
- Adam



Nous arrivons enfin à Adam, qui est censé être le plus efficace pour ce type de problème. Nous pouvons constater que les résultats sont effectivement très bons. On voit l'aspect adaptatif de l'algorithme, car même avec un taux d'apprentissage très faible, la convergence est très bonne. D'autre part, pour toutes les expériences ici, nous voyons que la convergence se fait très rapidement, ce qui s'explique par la contribution du momentum dans l'algorithme. Certains articles font référence à une certaine sensibilité d'Adam vis-à-vis de son hyperparamètre $\beta 1$. Cela est visible sur le graphique, mais nous le verrons juste après, les précisions sont effectivement meilleures de quelques points lorsque $\beta 1$ est élevé (0,99), et ce même avec un faible taux d'apprentissage.

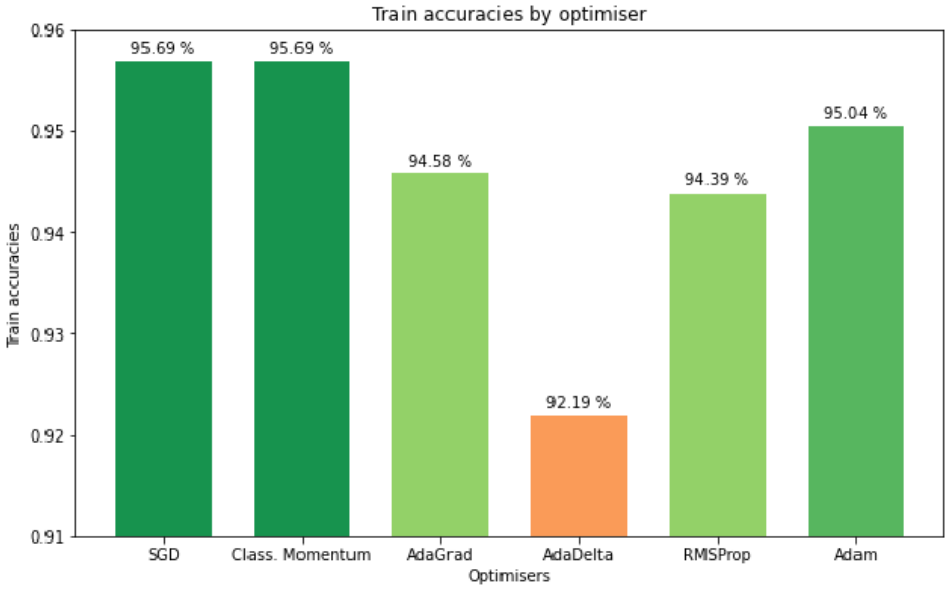
- Le meilleur optimizer

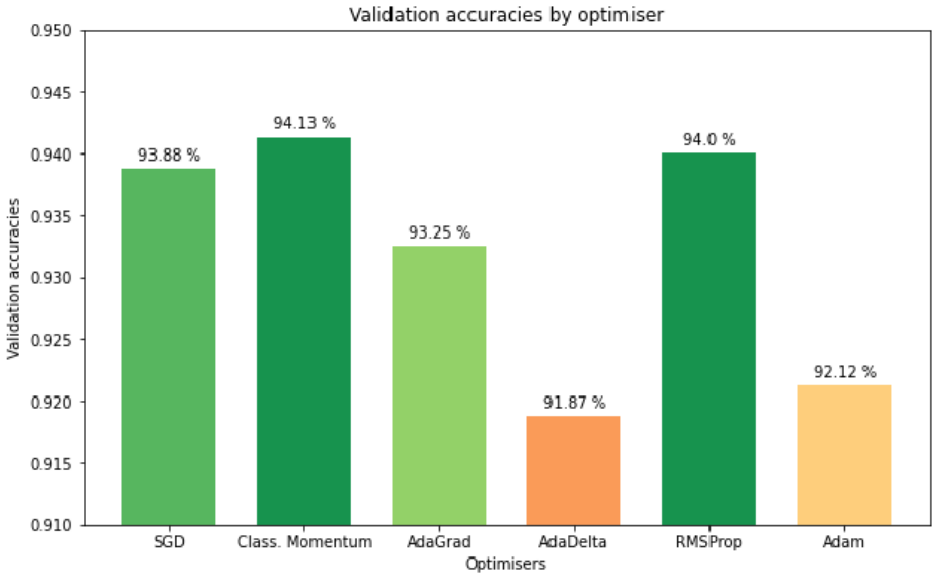
Nous sommes enfin arrivés à la fin de nos expériences, et il est temps de comparer les différents résultats. Si nous récapitulons, nous avons choisi un modèle avec un bloc convolutif (qui peut aller jusqu'à trois sans trop de différence), des images de taille 64, et 30 epochs. Regardons l'accuracy en fonction des epochs.



Nous observons plus ou moins les mêmes comportements pour le training et la validation. Comme nous l'avons vu, AdaDelta n'a pas le temps de converger assez rapidement avec notre modèle, nous l'éliminons donc tout de suite.
Ensuite, AdaGrad a tendance à rester coincé dans une vallée au début, mais finit par converger assez bien.
Pour les autres, le résultat final est assez proche, ce qui est intéressant c'est de voir la vitesse de convergence. Et aussi surprenant que cela puisse paraître, il semble que notre configuration permette à SGD de converger très rapidement, et au final d'obtenir la meilleure accuracy. D'une certaine

manière, nous pouvons dire que nous n'avons pas "besoin" de plus que SGD. Notons que, d'après nos expériences, et nous le voyons sur le graphique, SGD est meilleur avec le momentum et un faible taux d'apprentissage. En effet, comme nous le savons, le momentum permet d'accélérer la convergence dans la bonne direction (selon des dimensions spécifiques). Ici, le SGD converge déjà bien. En ajoutant du momentum et en diminuant le taux d'apprentissage, c'est le meilleur compromis pour notre configuration, c'est-à-dire notre jeu de données et notre modèle. Enfin, avant de comparer ces résultats avec l'état de l'art, affichons les différentes précisions dans un diagramme à barres.



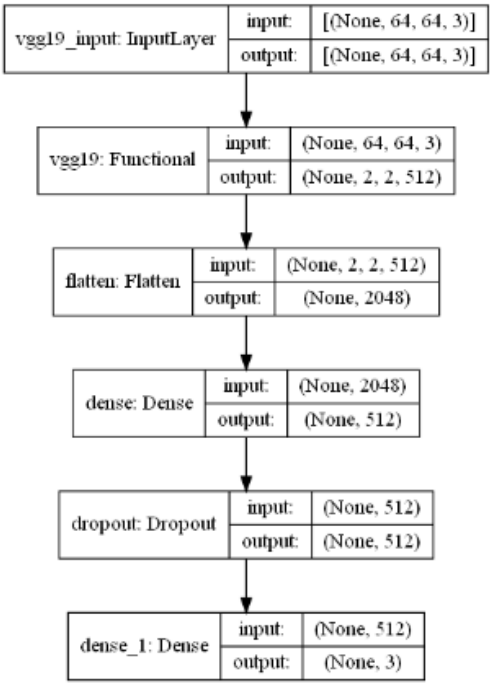


Notez donc que, bien que SGD soit la meilleure solution, RMSProp semble bien fonctionner aussi, mieux que Adam (même si ce dernier fonctionne très bien aussi). Pour conclure, nous confirmons ce que nous avons dit précédemment, à savoir qu'un optimizer SGD bien réglé fonctionne finalement mieux qu'Adam, qui se “règle” en quelque sorte tout seul.

État de l'art

Nous avons également comparé les résultats de notre modèle et le choix des optimizers avec un modèle de pointe pour la classification d'images, le VGG19. Il existe de nombreux autres modèles, mais pour réduire le temps d'apprentissage, nous avons décidé de nous en tenir à un seul.

Nous avons ajouté au modèle VGG19 deux couches denses avec, entre les deux, une couche de Dropout avec 0,2 comme taux de dropout. Ce modèle est initialement utilisé avec des images de plus haute résolution (234, 234, 3), ce qui le rend moins adapté à nos images de basse résolution, nous avons donc adapté l'input pour qu'il fit avec notre shape.

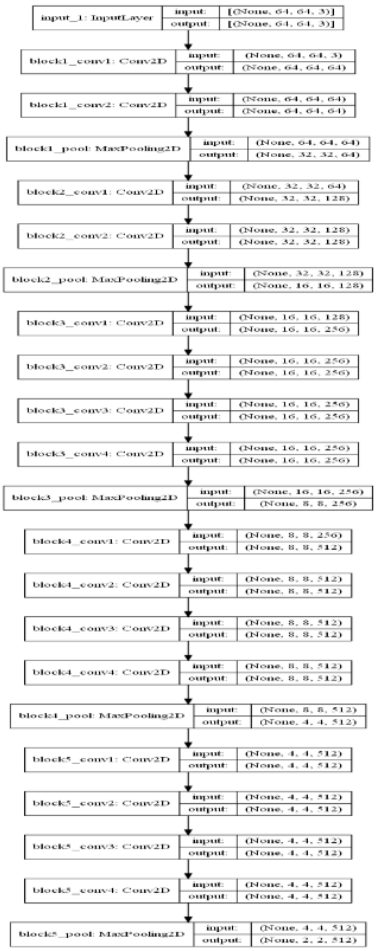


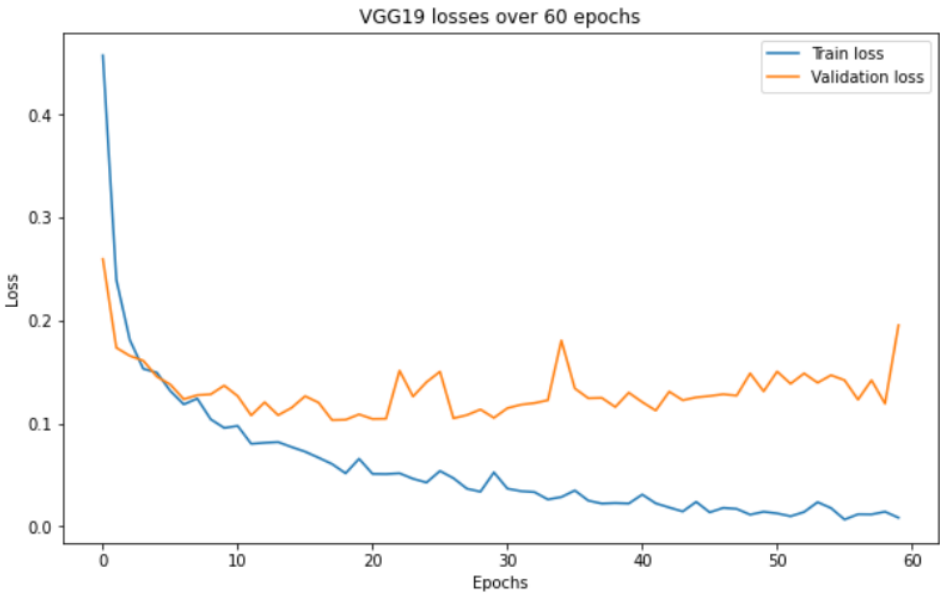
Il se compose de 16 blocs convolutifs ainsi que de 5 couches de max pooling. Il y a deux blocs de 2 blocs Conv2D et trois blocs de 4 blocs Conv2D qui sont suivis par ces couches de Max Pooling. Il s'agit donc d'un modèle assez vaste qui n'a pas besoin de s'entraîner avant d'ajouter le FCNN correspondant à nos catégories.
Comme il s'agit d'un grand modèle, nous avons décidé de l'entraîner avec les meilleurs paramètres d'optimisation.

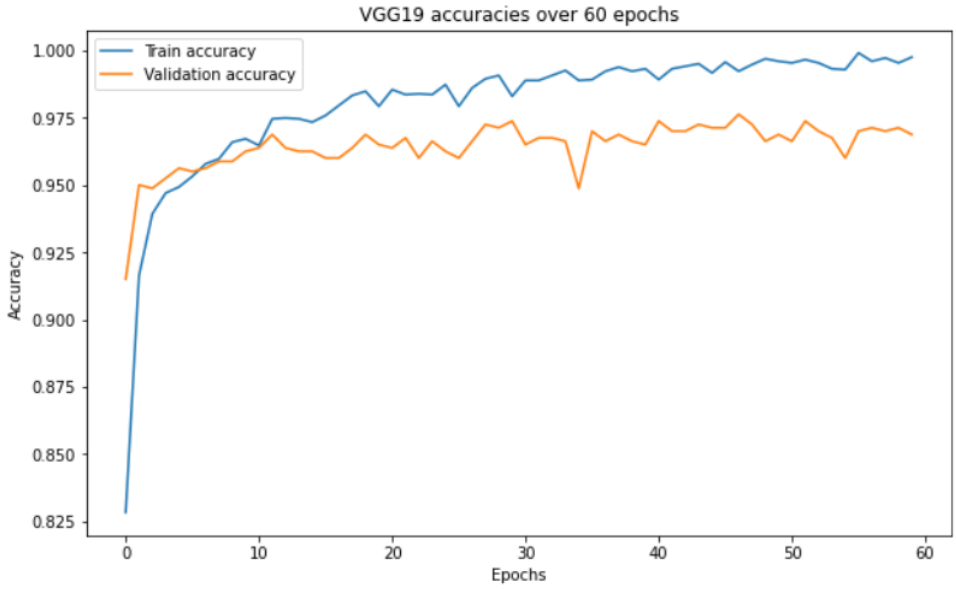
D'après nos résultats précédents, le meilleur optimizer pour notre étude de cas était SGD avec un taux d'apprentissage de 0,0001 et un momentum de 0,9.

Comme nous voulions obtenir les meilleurs résultats, nous avons augmenté le nombre d'epochs à 60 et nous avons finalement obtenu une précision finale sur l'ensemble de validation de 97,5 %, ce qui est très bien car nous avions un très petit ensemble de données.
Cela représente également une augmentation de 2 % par rapport au modèle très basique que nous avons utilisé. Par conséquent, il est encore possible de s'améliorer si nous trouvons un meilleur dataset avec des images de meilleure qualité.

Nous pouvons également voir sur le graphique une légère augmentation de l'overfitting de l'epoch 30 jusqu'à 60. Elle représente environ 2%.






VGG19 accuracies over 60 epochs
Train accuracy
Validation accuracy
Accuracy
Epochs
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0
10
20
30
40
50
60

Conclusion

Nos expériences ont clairement montré que Adam et RMSProp restent les solutions les plus simples et les plus efficaces à première vue. Cependant, nous avons constaté qu'un optimiseur SGD bien réglé peut être plus performant qu'un optimiseur adaptatif tel qu'Adam, qui essaie en quelque sorte de se “tuner” lui-même.
Cependant, nous avons été confrontés à quelques problèmes, notamment le manque de temps. Nous aurions pu être encore plus précis dans nos résultats. Nous aurions aimé utiliser la "Data Augmentation", ce qui nous aurait permis d'avoir le même nombre d'images dans les 3 catégories différentes et donc d'équilibrer le jeu de données.
Nous n'avons pas pu augmenter le nombre d'epochs en raison du temps de calcul associé au fait que nous devions tester des centaines de combinaisons. Heureusement que les GPU existent !