Philip Voinea

VOIP85020100

Prof. Mohammed Bouguessa

INF7370

TP2: Construction d'un modèle de classification d'images d'animaux

Montage de l'architecture et entrainement du modèle

Ensemble de données

L'ensemble de données d'entrainement fournies constitue 24000 images d'animaux (4000 pour chacune des six classes d'animaux : baleine, dauphin, morse, phoque, requin et requin baleine). J'ai réservé 20% de cet ensemble de côté pour la validation, c'est-à-dire 4800 images (800 par classe).

L'ensemble de données de test constitue 6000 images d'animaux, soit 1000 pour chacune des six classes.

Traitement des données

Les dimensions des images varient (la longueur d'un côté varie d'un minimum de 35 pixels à un maximum de 256 pixels); mais vu que l'entrée de l'architecture CNN a une taille fixe, un ajustement est donc nécessaire pour uniformiser les dimensions des images. À cette fin, j'ai appliqué un padding sur les images en collant chaque image au centre d'un arrière-plan noir de 256x256 pixels, ce qui correspond aux dimensions maximales de l'ensemble des données. Je n'ai pas appliqué de redimensionnement ou de zoom afin de garder les images aussi fidèles que possibles aux images sources.

J’ai commencé par le data augmentation qui existait dans le code de base à implémenter du data augmentation quand j’ai remarqué que mes modèles surapprenaient les données d’entrainement sans amélioration sur la validation de la manière suivante :

shear_range=0.1 (cisaillement de l’image de -0.1 à 0.1 degrés)

zoom_range=0.1 (zoom de -10% à 10%)

rotation_range=30 (rotation de -30 à 30 degrés)

brightness_range=(0.7,1.3) (luminosité de 70% à 130% de l’original)

channel_shift_range=30 (décalage des canaux RVB de -30 à +30)

height_shift_range=0.1 (décalage vertical de l’image de -10% à 10%)

width_shift_range=0.1 (décalage horizontal de l’image de -10% à 10%)

horizontal_flip=True (reflection horizontale aléatoire de l’image)

Paramètres et hyperparamètres

L’optimisateur Adam avec les valeurs attribuées par défaut dans Keras ($\alpha = 0.001$, $\beta1 = 0.9$, $\beta2 = 0.999$, $\varepsilon = 10^{-7}$) m’a suffi. J’utilise l’entropie croisée comme fonction de perte.

Pour l’entrainement, j’ai utilisé des lots de 64, soit la taille maximale de lot que les GPU T4 de Google Colab peuvent supporter.

J’ai entrainé le modèle pendant 100 époques sans arrêt précoce *automatique*.

La raison pour laquelle je n’ai pas implémenté d’arrêt précoce automatique est que le risque me paraissait trop élevé; en effet, mes modèles passaient parfois nombreuses époques à stagner avant d’améliorer l’exactitude sur les données de validation, souvent parce qu’une époque passée avait trop bien performé sur les données de validation, peut-être par chance. Étant donné ces oscillations chaotiques d’époque en époque dans la

performance sur les données de validation, j'ai jugé que ce serait mieux d'arrêter manuellement l'entrainement lorsque les exactitudes tombent ou stagnent *en moyenne*. J'aurais pu implémenter ou développer un algorithme d'arrêt précoce basé sur une moyenne mobile comme critère d'arrêt, mais je me suis contenté de le faire manuellement et à l'œil étant donné le petit nombre de modèles que j'ai dû tester.

Architecture

Les entrées passent par trois blocs de convolution avant de passer par trois couches complètement connectées et la couche de sortie de 6 neurones. J'utilise du drop-out avec un taux d'extinction de 0.2 et toutes les fonctions d'activation sont ReLU. J'utilise également de la régularisation L1L2 sur les couches denses avec l1 = 10^-5 et l2 = 10^-4.

Premier bloc de convolution:

- Deux couches de convolution 2D avec 32 filtres chacune de taille 3x3, avec un remplissage zero padding (padding = 'same') pour conserver la taille de l'image et une activation ReLU.
- Une normalisation par lot (BatchNormalization).
- Une couche de sous-échantillonnage (MaxPooling2D) avec une fenêtre de 2×2 et un remplissage zero padding (padding = 'same').
- Un dropout avec un taux d'extinction de 0.2.

Deuxième bloc de convolution:

- Deux couches de convolution 2D avec 64 filtres chacune de taille 3x3, avec un remplissage zero padding (padding = 'same') pour conserver la taille de l'image et une activation ReLU.
- Une normalisation par lot (BatchNormalization).
- Une couche de sous-échantillonnage (MaxPooling2D) avec une fenêtre de 2×2 et un remplissage zero padding (padding = 'same').

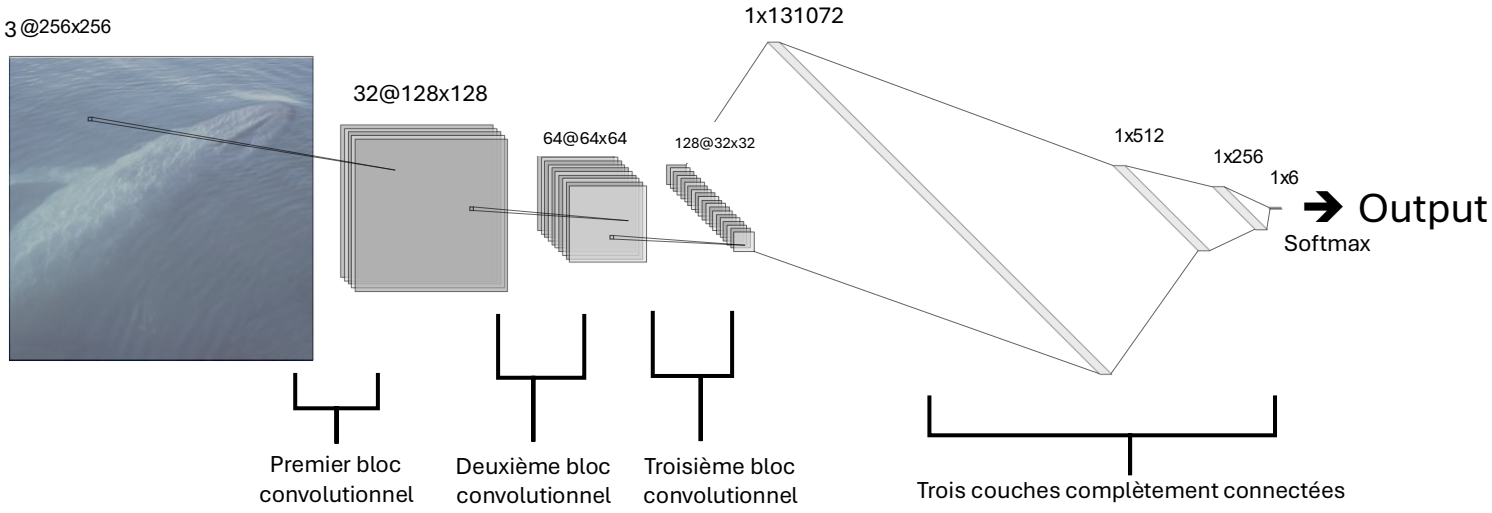
- Un dropout avec un taux d'extinction de 0.2.

Troisième bloc de convolution:

- Deux couches de convolution 2D avec 128 filtres chacune de taille 3x3, avec un remplissage zero padding (padding = 'same') pour conserver la taille de l'image et une activation ReLU.
- Une normalisation par lot (BatchNormalization).
- Une couche de sous-échantillonnage (MaxPooling2D) avec une fenêtre de 2×2 et un remplissage zero padding (padding = 'same').
- Un dropout avec un taux d'extinction de 0.2.

Partie entièrement connectée (fonction fully_connected) :

- La couche « Flatten » convertit les matrices obtenues en un vecteur de longueur 131072.
- Une couche dense de 512 neurones avec fonction d'activation ReLU et régularisation L1L2 (l1=1e-5 et l2=1e-4).
- Un dropout avec un taux d'extinction de 0.2.
- Une couche dense de 256 neurones avec fonction d'activation ReLU et régularisation L1L2 (l1=1e-5 et l2=1e-4).
- Un dropout avec un taux d'extinction de 0.2.
- Une couche dense de sortie à 6 neurones avec fonction d'activation softmax.

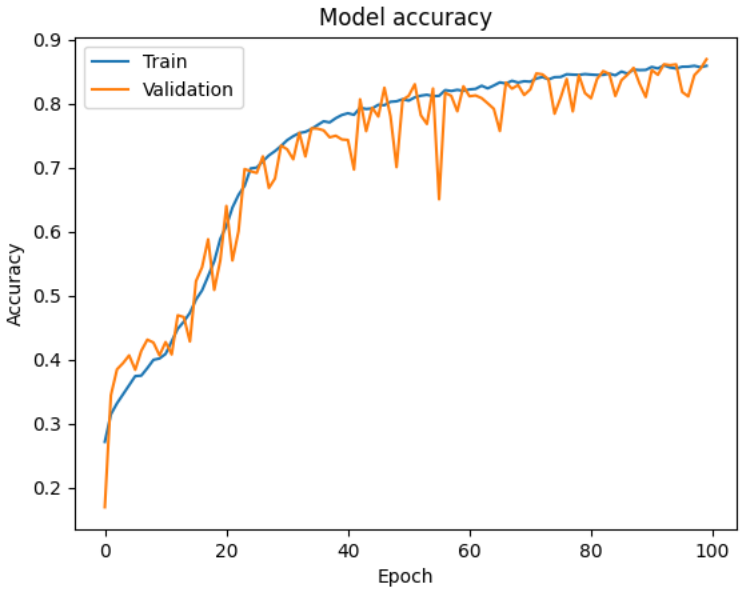


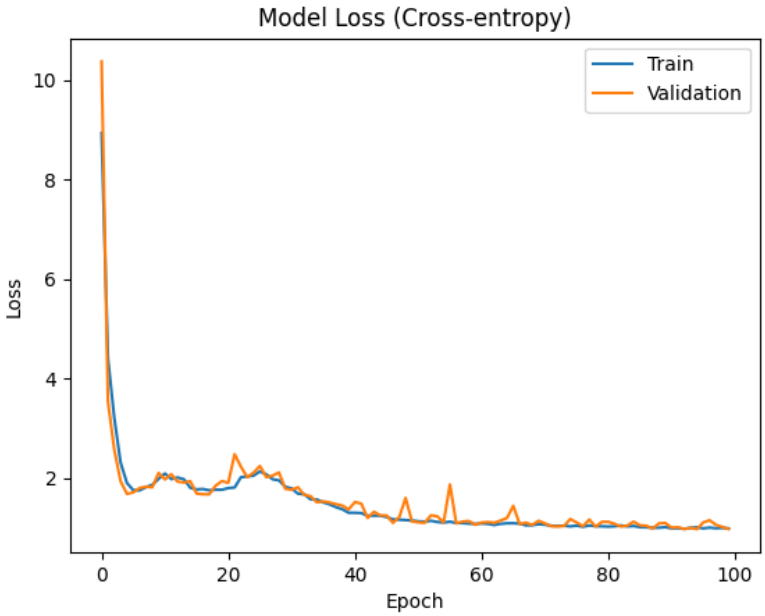
Résultats d'entrainement

L'entrainement a duré environ 1 heure et 53 minutes.

L'erreur minimale commise lors de l'entrainement fut de 0.9746 sur les données d'entrainement et 0.9675 sur les données de validation.

L'exactitude maximale atteinte lors de l'entrainement fut 0.8603 sur les données d'entrainement (augmentées) et 0.8698 sur les données de validation.





Justification du choix de l'architecture

1. **Le squelette**

Je me suis inspiré de l'architecture fournie par l'exemple MNIST. D'abord, j'ai seulement adapté la couche de sortie pour un problème à 6 classes et il était clair que le problème était bien trop petit.

J'ai donc essayé deux approches différentes pour complexifier le modèle : élargir et approfondir les couches complètement connectées et ajouter des couches de convolution.

J'ai ajouté une troisième couche de convolution avec 128 filtres, soit le double du nombre de filtres dans la couche de convolution précédente parce que j'ai remarqué que la deuxième couche de convolution dans l'exemple contenait le double du nombre de filtres de la première couche de convolution. J'ai recherché cette architecture et j'ai lu qu'il est commun d'augmenter le nombre de filtres dans les couches de convolution subséquentes parce que tandis que les premières couches tendent à capturer les caractéristiques de bas-niveau (des lignes, des coins, des bordures, etc.), les caractéristiques de haut-niveau captées par les dernières convolutions sont plus complexes et diverses et bénéficient donc de plus de filtres, ce qui correspond avec mes connaissances préalables en CNN.

J'ai ensuite doublé les couches de convolution afin d'augmenter le champ réceptif des convolutions, premièrement parce que je doutais bien que les caractéristiques importantes dans des images d'animaux de dimensions 256x256 ne requièrent pas un plus grand champ réceptif que les caractéristiques de chiffres dans des boîtes 28x28. Mais plutôt que de garder le même nombre de convolutions et d'agrandir leur champ réceptif à 5x5, j'ai décidé de doubler les couches de convolution 3x3; j'ai vu cette pratique dans des architectures CNN et j'ai lu que deux convolutions 3x3 requièrent moins d'hyperparamètres qu'une convolution 5x5 et que la non-linéarité additionnelle peut être avantageuse (Simonyan & Zisserman, 2015). J'ai donc essayé cette méthode et cela a beaucoup amélioré mes résultats.

Finalement, j'ai choisi la fonction d'activation ReLU pour toutes mes couches afin d'éviter des problèmes de gradient (vanishing and exploding gradient) et de simplifier les calculs. J'ai considéré la fonction LeakyReLU, qui me semblait auparavant nettement supérieure étant donné que les neurones ne peuvent pas mourir, mais le fait que la fonction ReLU demeure toujours plus populaire, d'après le cours, m'a intrigué. Cela m'a poussé à rechercher les avantages de ReLU sur LeakyReLU et j'ai appris que la mort de neurones peut constituer un avantage car cela encourage les représentations parsimonieuses et peut donc réduire le overfitting. J'ai donc gardé ReLU comme fonction d'activation à travers l'architecture.

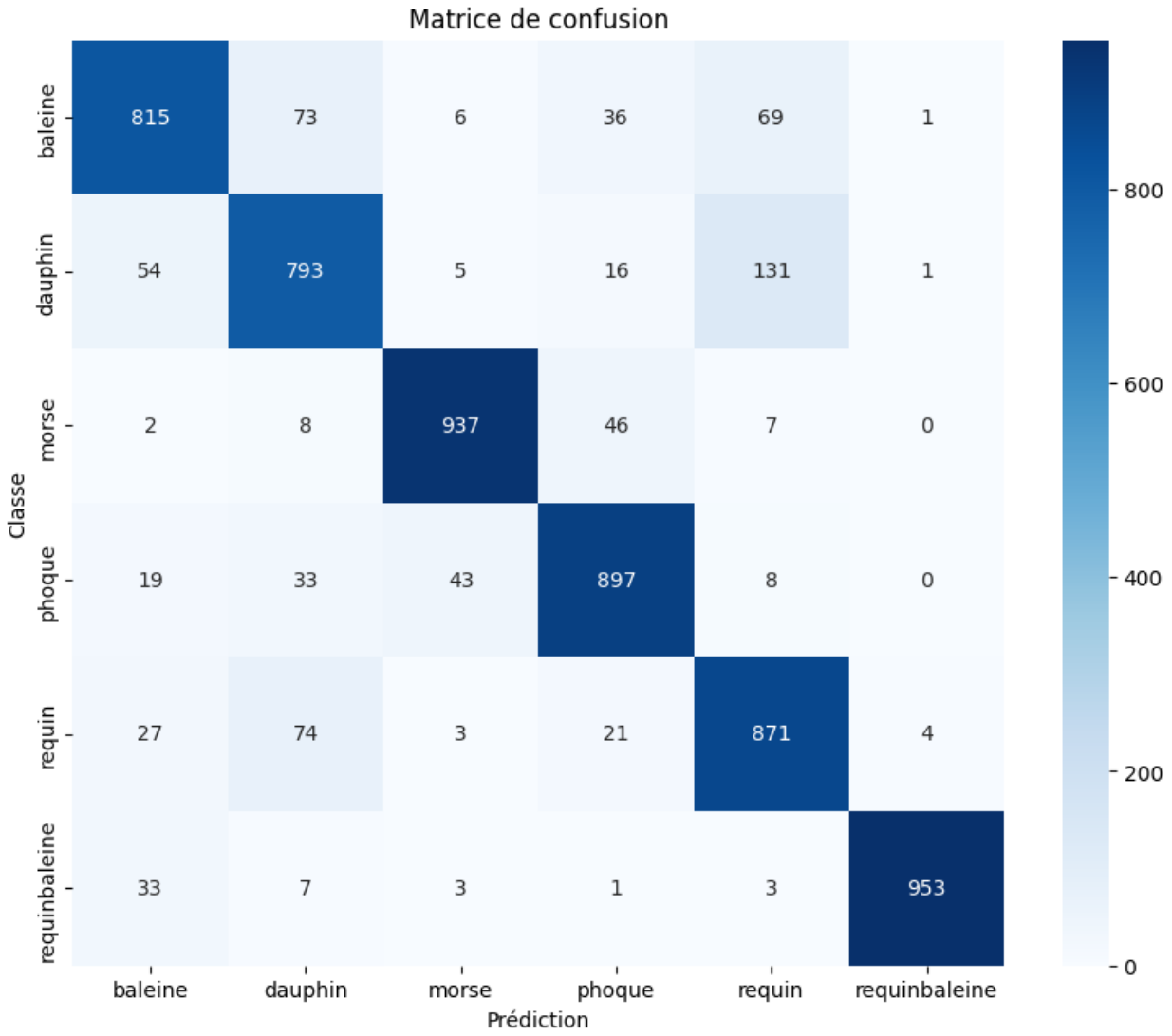
2. La régularization

J'ai implémenté un dropout avec un taux d'extinction minimal de 0.2 sur les couches denses parce que j'ai remarqué que la performance de mon modèle sur les données de validation stagnait même quand l'exactitude sur les données d'entrainement augmentait. Ce dropout a réduit le surapprentissage dont mon modèle souffrait, mais pas assez. J'ai donc testé trois mesures additionnelles en parallel : augmenter le taux d'extinction (à 0.3 et ensuite à 0.5), implémenter la régularization L1L2 sur les couches denses (avec les valeurs de défaut de Keras, l1=l2=0.01) et appliquer le dropout aux blocs de convolution aussi. Les deux premières mesures ont trop obstrué l'apprentissage même des données d'entrainement, tandis que la dernière mesure a bien marché.

J'ai ensuite appliqué une très légère régularization L1L2 (l1=10^-5, l2=10^-4) sur les couches denses parce que le modèle avait du mal à dépasser 82% en exactitude sur les données de test à cause du overfitting. J'ai gardé l1 < l2 parce que l'ajout des valeurs absolues des poids à la perte tire les poids vers 0 plus fortement que l'ajout des carrés des poids quand les poids sont proches de 0. C'est ainsi, j'ai réussi à atteindre une exactitude de 84.46%.

Évaluation du modèle

Exactitude : 84.46%



V: baleine
P: baleine

V: baleine
P: dauphin

V: baleine
P: morse

V: baleine
P: phoque

V: baleine
P: requin

V: baleine
P: requinbaleine

V: dauphin
P: baleine

V: dauphin
P: dauphin

V: dauphin
P: morse

V: dauphin
P: phoque

V: dauphin
P: requin

V: dauphin
P: requinbaleine

V: morse
P: baleine

V: morse
P: dauphin

V: morse
P: morse

V: morse
P: phoque

V: morse
P: requin

V: morse
P: requinbaleine

V: phoque
P: baleine

V: phoque
P: dauphin

V: phoque
P: morse

V: phoque
P: phoque

V: phoque
P: requin

V: phoque
P: requinbaleine

V: requin
P: baleine

V: requin
P: dauphin

V: requin
P: morse

V: requin
P: phoque

V: requin
P: requin

V: requin
P: requinbaleine

V: requinbaleine
P: baleine

V: requinbaleine
P: dauphin

V: requinbaleine
P: morse

V: requinbaleine
P: phoque

V: requinbaleine
P: requin

V: requinbaleine
P: requinbaleine

Conclusion

Les difficultés quant au choix de l'architecture sont discutées dans la section « Justification du choix de l'architecture » de la première section.

Je pourrais davantage améliorer le modèle en choisissant plus judicieusement les valeurs l1 et l2 et en expérimentant avec différentes architectures pour les couches de convolution.

Références

Simonyan, K., & Zisserman, A. (2015). *Very deep convolutional networks for large-scale image recognition* (arXiv:1409.1556v6). arXiv. https://arxiv.org/abs/1409.1556v6.