Dans ce travail pratique on doit développer un réseau de neurones à convolution (CNN) capable de distinguer entre les images des dauphins et des requins. Pour cela on a un ensemble de données qui comprend un total de 16000 images séparées en part égal d'images de dauphins et de requins, ces images sont réparties entre 13000 images de données d'entrainement et 3000 images de données de test.

1) Montage de l'architecture et entrainement du modèle

Tout d'abord, on a divisé les données d'entrainement en 11000 images pour l'entrainement et 2000 images pour la validation, les proportions entre les images de dauphins et les images de requins sont dans le tableau ci-dessous.

Données	Dauphins	Requins	Total
Entrainement	5500	5500	11000
Validation	1000	1000	2000

On n'a pas effectué de prétraitement de données.

L'optimisateur qu'on a utilisé est Adam et le paramètre associé à cet optimisateur est learning_rate = 0.001 (*Optimizer = Adam(learning_rate=0.001))*

On a aussi ajouté dans le code "1_Modele.py" une fonction lr_schedule qui ajuste automatiquement le taux d'apprentissage en fonction de l'epoch spécifiée. Le but de cette fonction est d'entraîner le modèle avec un taux d'apprentissage plus élevé au début (pour une convergence rapide) et le réduire progressivement au fur et à mesure que l'entraînement progresse (pour affiner le modèle), ce qu'elle fait c'est que à chaque 10 epochs le taux d'apprentissage est réduit de 10% de sa valeur initiale.

Comme mentionner la taille du lot (*batch size*) d'entrainement qu'on a utilisé est de 11000 et la taille du lot (*batch size*) de validation est de 2000.

On a utilisé un nombre d'époques (*number of Epochs*) égal à 50, sans arrêt précoce et un fit_batch_size de 64.

La fonction "feature_extraction" qui fait le traitement de la couche d'entrée a été divisée en quatre étages avec des couches de convolution, de normalisation par lots (Batch Normalization), des activations ReLU et des opérations de Max Pooling successives pour réduire la taille spatiale de l'entrée tout en augmentant le nombre de filtres à chaque étape. Le nombre total de couches utilisées est de 16.

Le code effectue les opérations suivantes sur la couche d'entrée ; d'abord on applique une couche de convolution avec 64 filtres de taille (3, 3) et un padding de 'same' pour conserver la taille spatiale de l'entrée, le résultat est passé à une couche de Batch Normalization pour normaliser les activations de la couche de convolution, ensuite on applique une fonction d'activation ReLU pour faire un entraînement plus rapide et plus efficace, et finalement on fait une opération de Max Pooling pour simplifier la sortie et permettre de réduire le nombre de paramètres que le réseau doit apprendre.

On répète ces étapes trois fois de plus, mais à chaque itération le nombre de filtres dans la couche de convolution double, c'est à dire la première couche de convolution a 64 filtres, la deuxième 128 filtres, la troisième 256 filtres et la quatrième en a 512.

À la dernière étape, le résultat de la dernière couche de Max Pooling (après la couche de convolution avec 512 filtres) sera l'entrée de la fonction de la partie complètement connectée.

En résumé on a utilisé :

- 4 couches de convolutions (Conv2D) avec des filtres de taille 3X3 et un padding de 'same'. Ces couches de convolutions ont différents nombres de filtres.
- 4 couches de Batch Normalization,
- 4 couches d'activation avec la fonction ReLU, pour introduire la non-linéarité dans les activations de la couche
- 4 Couches d'échantillonnage MaxPooling2D avec une taille de la fenêtre 2x2 et le paramètre padding mis à "same", pour réduire la taille spatiale de l'activation de moitié

La partie complètement connectée est composée d'une couche d'entrée (Flatten) pour transformer les données d'entrée en un vecteur plat enfin de permettre la connexion des couches Dense, deux couches cachées (Dense), deux couches d'activation (ReLU et sigmoïde) et une couche de dropout avec un taux de 0.4.

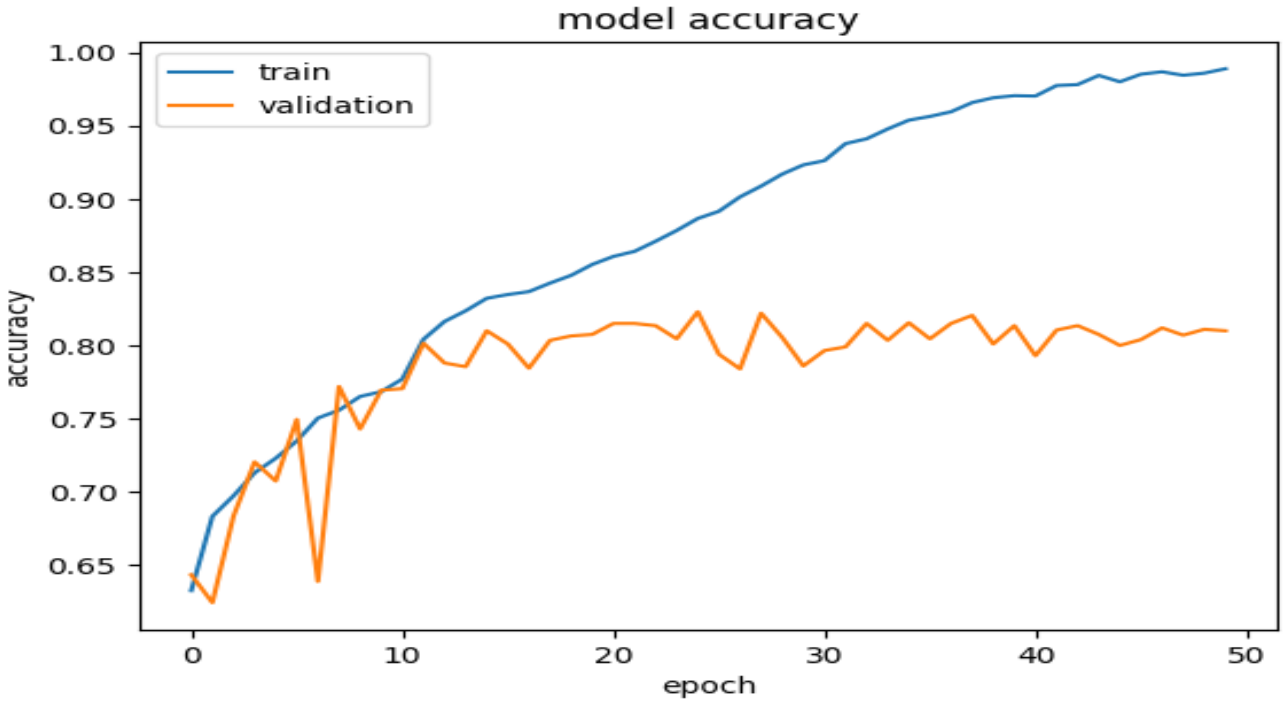
La première couche Dense avec 512 neurones prend en entrée le vecteur plat (résultant de la couche Flatten) et produit un vecteur de sortie avec 512 dimensions, la deuxième couche Dense avec 1 neurone produit une valeur de sortie unidimensionnelle.

Pour les couches d'activation, la première est appliquée sur la sortie de la première couche dense et utilise la fonction ReLU, la deuxième est pour produire une couche de sortie avec la fonction sigmoïde pour la classification binaire.

Lors de l'entrainement, le modèle a donné les résultats suivants (Voir aussi annexe) :

- Le temps total d'entraînement en minutes : 8m4s
- L'erreur minimale commise lors de l'entrainement (*Minimum Loss*) : 0.0328
- L'exactitude maximale de l'entraînement (*Maximum Accuracy*) : 0.988

La courbe d’exactitude (*accuracy*) par époque



Cette courbe affiche l'exactitude (accuracy) du modèle sur l'ensemble d'entraînement et l'ensemble de validation au cours des différentes époques d'entraînement.

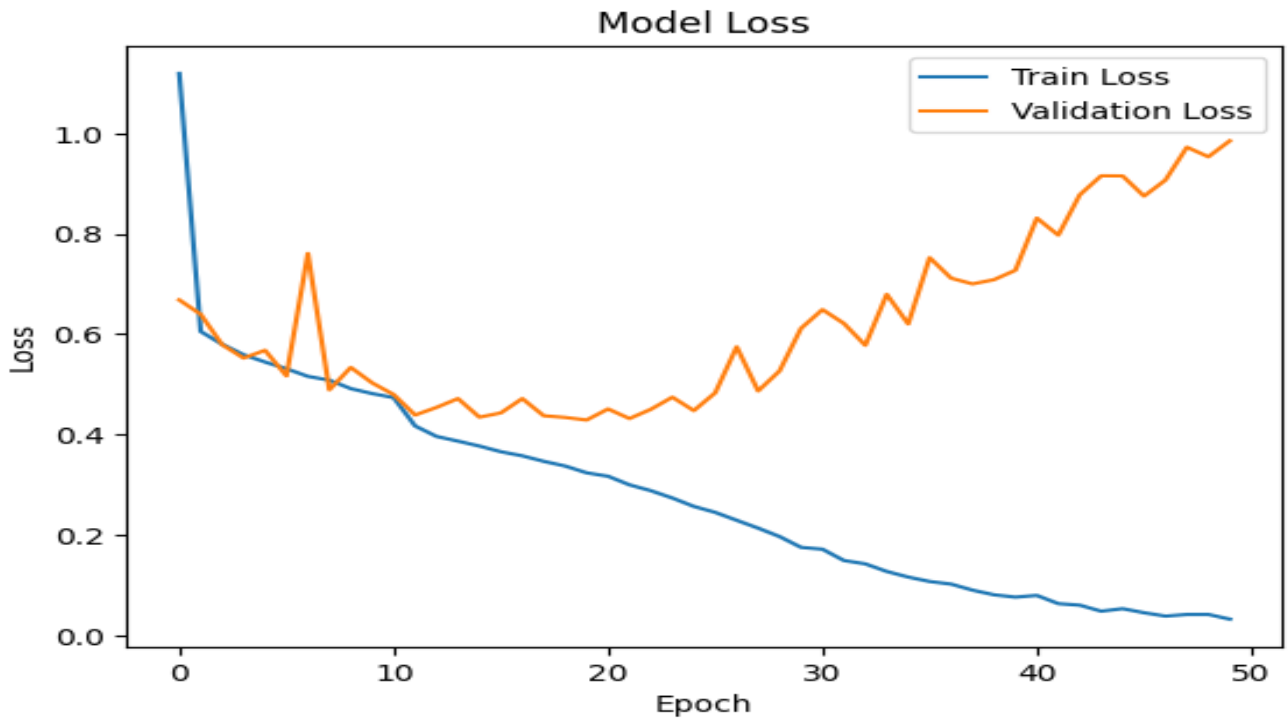
La courbe 'train' représente l’exactitude sur l'ensemble d'entraînement, tandis que la courbe 'validation' représente l’exactitude sur l'ensemble de validation.

Elle permet de visualiser comment la performance du modèle évolue au fur et à mesure de son apprentissage.

On voit que les deux courbes augmentent ensemble pour les époques inférieures à 12 ou 15, et après la courbe d’entraînement continue d’augmenter au même rythme, tandis que celle de validation diminue le rythme ou stagne.

Cela montre que le modèle apprend et ajuste les hyperparamètres dans les quinze premières époques au même rythme que l’entrainement, et après l’apprentissage des hyperparamètres se fait lentement.

La courbe d’erreur par époque (Training vs Validation)



La courbe d’erreur par époque permet de comprendre comment le modèle se comporte à mesure qu'il apprend à partir des données fournies.

La courbe d'entraînement représente l'évolution de l'erreur sur l'ensemble de données d'entraînement à mesure que le modèle apprend. Elle montre comment l'erreur diminue au fil des époques. La baisse régulière de l'erreur indique que le modèle apprend correctement à partir des données d'entraînement.

La courbe de validation représente l'évolution de l'erreur sur un ensemble de données de validation indépendant pendant l'entraînement. Cet ensemble de données est utilisé pour apprendre les hyperparamètres du modèle en utilisant des données qu'il n'a pas encore vues.

La courbe de validation montre une baisse de l'erreur au début, puis effectue une augmentation, cela signifie que le modèle devient sensible à l’apprentissage des hyperparamètres après certaines époques, cela explique l’observation fait avec la courbe d’exactitude par époques, ou l’apprentissage des hyperparamètres commence à se faire lentement après quelques époques.

Justification des choix et les facteurs ayant contribués à l'amélioration de l'entraînement

Les valeurs de training_batch_size et validation_batch_size déterminent le nombre d'échantillons traités dans chaque lot de formation et de validation, une taille de lot plus importante peut améliorer la vitesse d'apprentissage, et conduire à des gradients plus stables pendant la formation et entraîner une convergence plus rapide, par contre des valeurs aussi élevées peuvent entraîner des problèmes de mémoire, en particulier si le GPU ne dispose pas d'une mémoire suffisante pour les gérer, mais dans ce travail on n'a pas beaucoup rencontré un tel problème, puisqu'on a utilisé le paramètre fit_batch_size qui permet une utilisation efficace du GPU sans occuper trop de mémoire.

La variation de ces valeurs n'a pas beaucoup contribué à l'amélioration de l'entrainement. Au début on a utilisé les valeurs d'entraînement de 10000 pour training_batch_size et 3000 pour validation_batch_size, ensuite on est passé aux valeurs 11000 et 2000, l'entraînement c'est un peu améliorer mais l'exactitude sur les données de test est restée en bas de 70%.

La variable image_scale définit la taille à laquelle les images d'entrée sont redimensionnées, dans ce travail les images sont redimensionnées à 80x80 pixels, on a fait varier cette taille entre 100x100 pixels et 75x75 pixels, et le meilleur résultat a été obtenu avec une taille de 80x80 pixels. Mais, l'exactitude sur les données de test est restée toujours en bas de 78%.

Le paramètre fit_batch_size détermine la taille du lot pendant l'apprentissage du modèle, c'est le nombre d'échantillons utilisés pour calculer les gradients à chaque itération pendant l'apprentissage.

Les lots de petite taille tendent à fournir des gradients plus précis, mais peuvent ralentir l'apprentissage en raison des mises à jour fréquentes. Les lots de plus grande taille sont efficaces en termes de calcul, mais peuvent produire des gradients moins précis. On a commencé avec une valeur de 32, puis 64, et 128. Le fit_batch_size de 64 a donné un meilleur résultat, mais là encore l'exactitude est toujours restée en bas de 78%.

Le paramètre fit_epochs est le nombre de fois que l'ensemble des données sera passé par le modèle pendant l'apprentissage, elle doit être suffisamment élevée pour que le modèle converge, mais pas trop pour éviter le surapprentissage.

Dans ce travail, fit_epochs est l'un des paramètres qu'on a beaucoup fait varier, On a utilisé les valeurs entre 10 et jusqu'à 60. Les valeurs plus petites n'ont pas donné d'amélioration de l'entraînement, mais en l'augmentant on a pu voir une nette amélioration, jusqu'à avoir dés fois une exactitude de 78%.

On a utilisé une valeur finale de 50 epochs, et on peut dire que ce facteur a contribué à l'amélioration de l'entrainement.

La fonction feature_extraction est conçue pour extraire les caractéristiques des images d'entrée à l'aide de plusieurs couches convolutives comportant un nombre variable de filtres.

On a fait un choix de 4 couches convolutives avec des profondeurs croissantes (64, 128, 256, 512) et cela semble convenir à la tâche actuelle. On a choisi ce nombre de couches, en l'occurrence quatre couches convolutives, pour créer un processus d'extraction de caractéristiques en profondeur. L'avantage des réseaux plus profonds avec plus de filtres c'est qu'ils peuvent capturer des motifs et caractéristiques plus complexes et aider le modèle à apprendre des représentations hiérarchiques à partir des images.

On a ajouté des couches MaxPooling2D pour réduire les dimensions spatiales, la BatchNormalization pour stabiliser l'apprentissage, et une fonction d'activation ReLU pour introduire la non-linéarité et permettre au réseau de neurones d'apprendre des relations complexes dans les données.

D'abord on a commencé avec un modèle de deux couches et on les a augmentées progressivement et à la quatrième couche on a vu une nette amélioration qui est resté stable après plusieurs exécutions du modèle.

L'augmentation du nombre de couches et de filtres a contribué à l'amélioration de l'entraînement et a permis de produire un modèle avec une exactitude plus élevée.

La fonction fully_connected définit les couches entièrement connectées (denses) après l'étape d'extraction des caractéristiques. On n'a pas beaucoup fait varier les variables et paramètres de cette fonction.

La fonction comporte une couche dense de 512 neurones suivie d'une activation ReLU et d'un dropout avec un taux de 0,4, et une couche dense finale avec un seul neurone et une activation sigmoïde.

Le choix de 512 neurones et d'un taux d'abandon de 0,4 a permis de capturer des modèles plus complexes, d'éviter le surapprentissage et de contrôler la capacité des couches entièrement connectées

On a utilisé une fonction (lr_schedule) pour programmer le taux d'apprentissage pendant la formation, elle réduit progressivement le taux d'apprentissage.

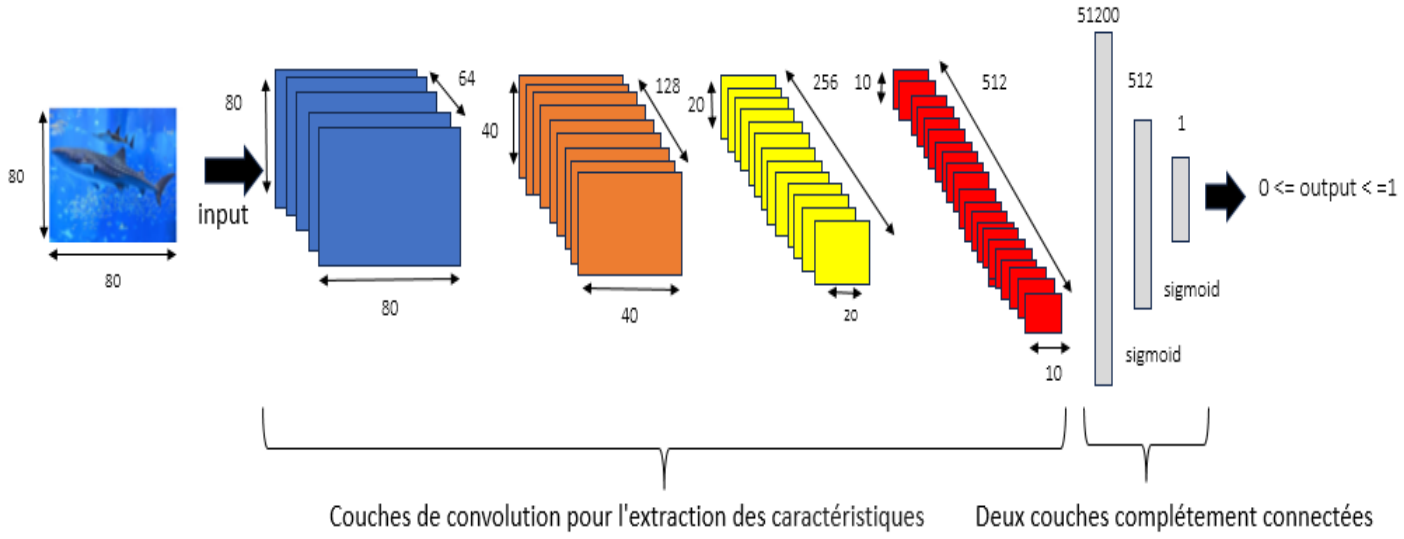
La diminution progressive du taux d'apprentissage comporte plusieurs avantages comme par exemple : Permettre au modèle de converger plus efficacement et éviter les dépassements ou les oscillations, améliorer la stabilité pendant l'apprentissage, éviter les dépassements ou les oscillations.

L'ajout de cette fonction a grandement contribué à l'amélioration de l'entraînement et on a pu atteindre un taux d'exactitude dans les 80%.

La fonction de perte binary_crossentropy (entropie croisée binaire) est mieux adaptée pour les problèmes de classification binaire où la sortie souhaitée est une valeur binaire (0 ou 1). Puisqu'on a une seule sortie binaire (une seule unité de neurone dans la couche de sortie), on a utilisé la fonction de perte binary_crossentropy. En plus, lorsqu'on utilise la fonction mse (mean squared error, erreur quadratique moyenne) le taux d'exactitude est de 50% pour le même modèle qui donne les taux d'exactitude aux alentours de 80% avec la fonction entropie croisée binaire.

Le graphique suivant relate l'architecture du modèle qu'on a élaboré

<u>Architecture du modèle élaboré.</u>



Architecture globale du CNN utilisée pour la classification des images 'Dauphins' et 'Requins'

2) Évaluation du modèle

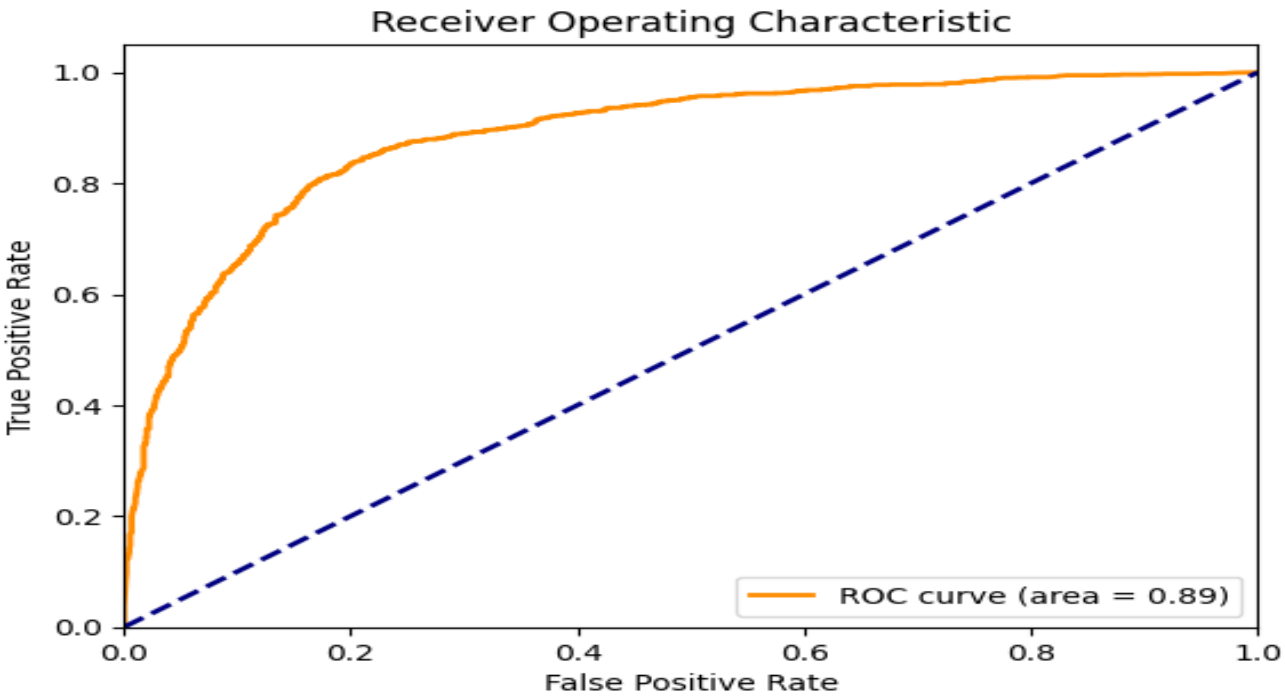
Lors de l'évaluation, le modèle a donné les résultats suivants (Voir aussi annexe) :

- L'exactitude (*accuracy*) du modèle sur les données de test : 0.8177 ou 81.77 %
- Le nombre total des images bien classées : 2452
- Le nombre total des images mal classées : 547

La matrice de confusion :

1197	303
244	1256

La courbe ROC



La matrice de confusion permet de comparer les prédictions du modèle avec les vraies étiquettes (ou classes) des données.

- 1197 échantillons ont été correctement classés comme dauphins (TN).
- 1256 échantillons ont été correctement classés comme requins (TP).
- 303 échantillons de la classe dauphin ont été mal classés comme classe requin (FP).
- 244 échantillons de la classe requin ont été mal classés comme classe dauphin (FN).

En résumé, le modèle a correctement classé 1197+1256=2453 échantillons., et mal classé 303+244=547 échantillons.

La courbe ROC affiche la relation entre le taux de vrais positifs et le taux de faux positifs pour différentes valeurs seuils de classification du modèle.

La ligne diagonale (ligne en pointillés) représente le modèle aléatoire, où le taux de vrais positifs est égal au taux de faux positifs. Notre modèle a une courbe ROC au-dessus de cette ligne, cela montre qu'il a une performance meilleure que le hasard.

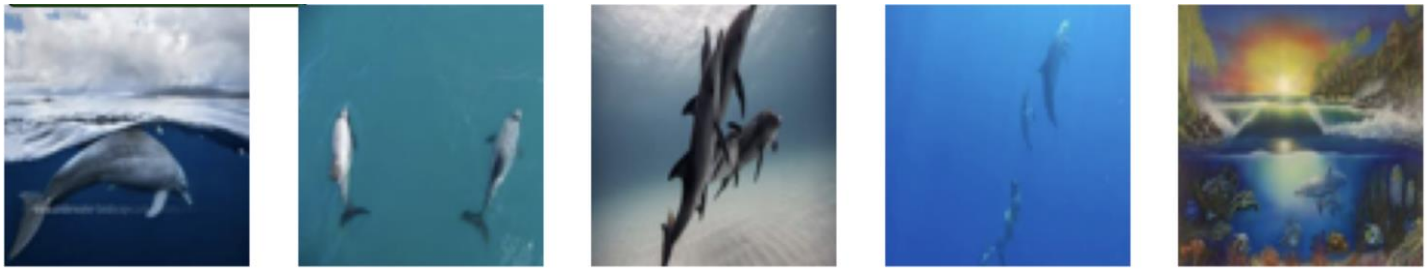
La courbe ROC qui se rapproche davantage du coin supérieur gauche et le score AUC-ROC plus élevée (0.89) indiquent une meilleure performance de classification pour notre modèle.

Images de dauphins et requins bien classés, et images de dauphins et requins mal classés.

Dauphins bien classés

Requins bien classés

Dauphins classes comme Requins



Requins classés comme dauphins

3) Conclusion

En conclusion, on va discuter d'une façon générale les problèmes rencontrés ainsi que les démarches possibles qui peuvent considérer pour améliorer notre modèle.

Voici la plupart des problèmes qu'on a rencontrés:

- Au début le modèle été trop simple pour capturer la complexité des données, et on avait des performances médiocres sur les données d'entraînement et de validation.
- Le modèle ne convergeait pas correctement et des fois trop lentement, mais une fois qu'on a ajusté le taux d'apprentissage, le problème a semblé être résolu.
- Trouver les hyperparamètres appropriés tels que le taux d'apprentissage, le taux de dropout, la taille du batch, etc., pour avoir de bonnes performances du modèle.
- Pouvoir interpréter le modèle profond, lorsqu'il commençait à être plus complexe, comme comprendre les raisons pour lesquelles le modèle prenait certaines décisions.
- Quelques erreurs dans le programme.
- La limite de nombre d'accès au GPU de google colab.

Pour améliorer ce modèle, on peut considérer plusieurs démarches et techniques, comme par exemple:

- Augmenter le jeu de données en appliquant des transformations aléatoires aux images existantes, telles que la rotation, le redimensionnement, le décalage horizontal/vertical, la translations, etc. Cela permettra au modèle de voir davantage de variations dans les images et de mieux généraliser.
- Réévaluer les valeurs des hyperparamètres tels que le batch size, le nombre d'époques, les taux d'apprentissage, le nombre de filtres dans les couches convolutionnelles, le nombre de neurones dans les couches entièrement connectées, etc. En utilisant des techniques telles que la recherche par grille ou la recherche aléatoire, On peut trouver les combinaisons optimales d'hyperparamètres pour améliorer les performances du modèle.
- Analyser les erreurs commises par le modèle sur le jeu de validation ou de test et examiner les images mal classées. Cela peut aider à comprendre les types d'erreurs commises, ou à identifier les faiblesses du modèle et pouvoir améliorer ces performances. Ensuite, on peut chercher des moyens de résoudre ces problèmes spécifiques, comme collecter davantage de données pour les cas sous-représentés, ajouter des exemples mal classés dans le jeu de données, ou apporter des modifications aux techniques d'augmentation des données.

- Tester différents optimisateurs d'apprentissage tels que RMSprop, Nadam, l'optimisation par momentum (SGD avec momentum) ou Adagrad pour voir comment ils affectent la convergence du modèle.
- Analyser les gradients du modèle peut aider à identifier les problèmes de vanishing gradients ou d'explosion des gradients, ce qui peut affecter la stabilité de l'apprentissage.
- En plus du dropout, on peut essayer d'autres techniques de régularisation comme la pénalisation L1 ou L2 sur les poids du réseau.
- Utiliser plusieurs modèles différents et les combiner en utilisant des méthodes d'ensemble, telles que le vote majoritaire ou le bagging. Cela peut aider à améliorer les performances en combinant les points forts de différents modèles.
- Nettoyer les Données en supprimant ou remplaçant les images présentant des problèmes de qualité graves, cela peut empêcher le modèle d'apprendre à partir de données bruyantes ou trompeuses.

Annexe:

Capture d'écran pour L'erreur minimale commise lors de l'entrainement et L'exactitude maximale de l'entraînement.

```
Epoch 50/50
172/172 [==============================] - ETA: 0s - loss: 0.0328 - accuracy: 0.9888
Epoch 50: val_accuracy did not improve from 0.82300
172/172 [==============================] - 9s 50ms/step - loss: 0.0328 - accuracy: 0.9888
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy', 'lr'])
```

Capture d'écran pour L'exactitude (*accuracy*) du modèle sur les données de test, le nombre total des images bien classées et le nombre total des images mal classées.

```
Found 3000 images belonging to 2 classes.
3000/3000 [==============================] - 16s 5ms/step - loss: 0.4899 - accuracy: 0.8177
>Test loss (Erreur): 0.4898982048034668
>Test précision: 0.8176666498184204
3000/3000 [==============================] - 8s 3ms/step
> 2452  étiquettes bien classées
> 547 étiquettes mal classées
Confusion Matrix:
[[1197  303]
 [ 244 1256]]
```