Université Nice Sophia Antipolis

POLYTECH NICE-SOPHIA

Projet de Fin d'Etude

Classification d'images de plancton par Deep Learning ou apprentissage hybride Deep Features/Random Forest

Encadrants : Eric Debreuve, Frédéric Precioso

Etudiants : Cédric Bailly, Thomas Mahiout, Thomas Jalabert

Remerciements

Nous tenons à remercier vivement ***Mr. Debreuve Eric*** et ***Mr. Precioso Frédéric***, nos deux superviseurs, pour le temps qu'ils nous ont consacré et les différents conseils/idées qu'ils nous ont donné afin d'améliorer notre projet.

Nous remercions également ***Mr. Irisson Jean-Olivier***, océanographe au Laboratoire Océanographique de Villefranche, de nous avoir fourni le dataset et pour ses réponses quant à nos interrogations. Il a su nous partager son enthousiasme et son expertise sur le sujet.

Table des matières

Introduction

Contexte

Les planctons jouent un rôle crucial dans la santé de notre écosystème. En effet, ceux-ci comptent pour plus de 50% de la matière organique produite sur Terre, les zooplanctons (animaux) sont à la base du réseau alimentaire aquatique et les phytoplanctons (plantes) absorbent le CO2 provenant de l'atmosphère et produisent de l'oxygène. Il est ainsi intéressant d'analyser l'évolution et la répartition des populations de planctons dans les océans afin de mesurer leur santé.

L'objectif de ce projet a donc été de classifier des images de planctons fournis par le LOV (Laboratoire Océanographique de Villefranche) de manière automatique grâce à des méthodes de Deep-Learning et notamment grâce aux réseaux de neurones convolutionnels (CNN/ConvNets).
En effet, le LOV utilise déjà diverses méthodes de classification basée sur des features extraites grâce à leur expertise et du Random Forest comme algorithme de décision. L'intérêt des réseaux de neurones convolutionnels est qu'ils permettent d'automatiser l'extraction des features, cependant le modèle hybride du LOV combinant CNN (pour extraire les features) et RF (pour la partie décisionnelle) n'a pas donné le score de classification attendu.

C'est dans ce contexte que se place notre PFE, l'objectif étant d'améliorer le score de classification en modifiant le dataset ainsi que les paramètres et l'architecture du CNN.
Nous allons donc voir dans une première partie les différentes techniques que nous avons mises en place afin d'améliorer la qualité du dataset (Pre-processing / Data Augmentation) ainsi que la définition de notre première architecture pour notre CNN. Puis, nous verrons les différentes architectures que nous avons proposés (notamment basé sur la taxonomie du vivant) afin d'améliorer le score de classification.

Langage et outils utilisés

Notre projet a intégralement été réalisé en Python qui possède de nombreuses librairies afin d'effectuer du machine learning (scikit-learn), du deep learning (Keras avec Tensorflow en backend) et du traitement d'image numérique (scikit-image, Opencv). La librairie Keras fournie notamment de nombreuses fonctionnalitées permettant de construire des réseaux profonds rapidement et simplement.

De plus, la phase d'entraînement d'un réseau profond étant très gourmande en terme de temps de calculs et le nombre d'images étant élevée, nous avons utilisé la librairie

cuDNN afin de faire tourner les calculs sur GPU qui sont bien plus adapté pour le processing d'images.

Nous avons également eu accès au cluster de calcul de l'INRIA utilisant le système OAR. L'utilisation de ce cluster fait partie d'une des difficultées que nous avons rencontré, au delà de la prise en main de celui-ci (définition des environnement python, soumission de job etc…). En effet, les GPU's du cluster étant achetés par les différentes équipes de l'INRIA, ceux-ci sont prioritaires lors de l'exécution de leur jobs et donc notre entraînement peut être *kill* a tout moment. Pour remédier à ce problème, il nous a fallu définir des checkpoints afin de mémoriser les poids présent sur nos liaisons entre neurones ainsi que l'epoch durant laquelle le programme s'est fait kill.

I Entraînement de classifieurs par Deep Learning

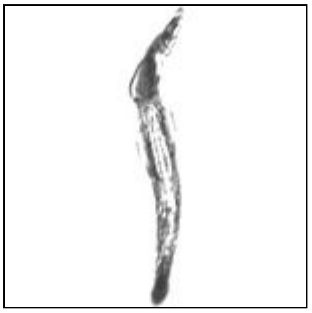
1) Qualité du dataset

Le dataset sur lequel nous avons effectué nos travaux a été proposé par le Laboratoire Océanique de Villefranche-Sur-Mer (LOV) par l'intermédiaire de J-O Irisson biologiste au LOV. Il existe un fort déséquilibre dans la représentation des classes dans le dataset. En effet, certaines classes sont sur représenté (trois classes concentres plus de 50% du dataset) alors qu'une dizaine d'autres classes contiennent moins de 10 images. De plus, la résolution des images est très variable. Il est nécessaire de redimensionner les images en évitant au maximum la distorsion. Enfin, les planctons sont des objets définis en trois dimensions, les images sont nécessairement en 2D et présente une grande variabilité de forme et de taille pour une même espèce. Nos modèles doivent être suffisamment robustes pour s'adapter à cette limitation.

2) Pré-Processing & Data augmentation

Pour améliorer la qualité de notre jeu de donné, nous avons au préalable pré-processé nos images, notamment en les redimensionnant. En effet, la taille des images de notre dataset étant très hétéroclyte : 20x20px pour les plus petites et jusqu'à 300x300px pour les plus grandes d'entre elles. Cette étape permet à la fois d'obtenir des images de meilleur qualité en plus d'être nécessaire lors de l'entrainement de notre CNN : en effet, nos inputs doivent avoir la même taille car la couche d'entrée du réseau de neurone convolutionnel est composé d'autant de perceptron que de pixels en entré. Qui plus est, les librairies Keras et Tensorflow mettent à disposition des outils permettant d'adapter automatiquement la résolution des images, seulement si une image à des proportions très différentes de celles en entrée du réseau convolutionnel, elles seront fortement déformés.

En revanche, puisque les images de planctons ont un fond blanc on peut parvenir à conserver la forme originale de l'objet. Pour cela on agrandit ou rétrécit l'image de façon uniforme pour que la plus grande de ses deux dimensions corresponde à la résolution requise et on complète l'image avec des pixels blancs.



De plus, afin d'agrandir la taille et l'équilibre d'images entre les différentes classes nous avons procédé à une phase de Data Augmentation. Nous avons utilisé différentes transformations d'images :

- Rotation (suivant une loi uniforme de 0 à 90 degrés)
- Retournement (Flipping) aléatoire
- Translation
- Cisaillement léger
- Ajout de bruit Gaussien

Ces transformations permettent également de rendre notre apprentissage plus robuste. En effet, en créant de manière artificielle de nombreuses images de planctons dans des sens/positions aléatoires et plus ou moins floues, on permet à notre CNN d'être plus robustes lors de la classification d'images inédites ou inhabituelles.

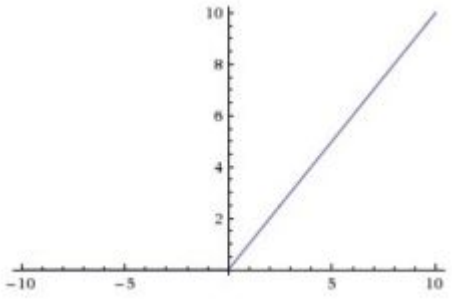
3) Choix d'une architecture

L'architecture de base de notre CNN est principalement inspirée par les travaux de l'équipe Deep Sea, vainqueurs du challenge Kaggle sur la classification de planctons, et le papier de Karen Simonyan & Andrew Zisserman de l'université d'oxford. Cette architecture consiste en un réseau très profond (~16 couches) et de multiple convolutions 3x3.

Toutes les images d'entrées de notre CNN ont une dimension de 95x95 et sont en niveau de gris (1 seul channel). Après chaque convolutions, on applique une opération supplémentaire appelé fonction d'activation, qui va permettre d'ajouter de la non linéarité à notre modèle. En effet, la majorité des données provenant du monde réel étant non-linéaire, il est souhaitable que notre CNN apprennent ces non-linéarités (les étapes de convolutions sont des opérations linéaires).

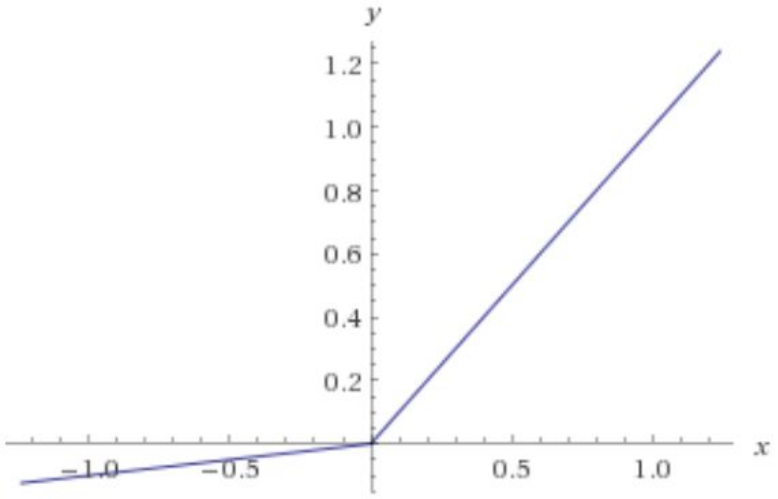
La fonction d'activation ReLU est une des plus utilisée lors de la définition de réseau profond car elle a été prouvée plus rapide et efficace (que les fonctions tanh ou sigmoid par exemple) afin d'entraîner des réseaux profonds sur un large datasets. Elle est définit de la manière suivante : $f(x) = max(0, x)$.

Output = Max(zero, Input)

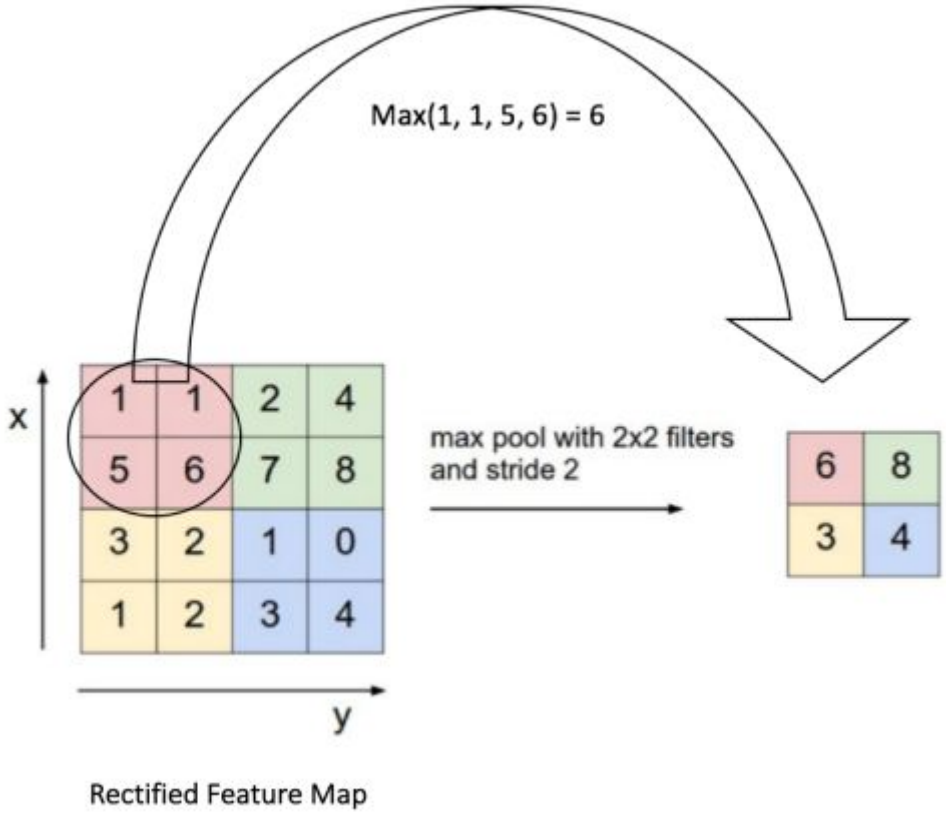


Cette fonction remplace ainsi (dans la matrice de feature obtenu après convolution) tout les pixels négatifs par 0. Cependant cette fonction peut entraîner un certain problème : si l'on fixe un pas d'apprentissage un peu trop élevé, il est possible qu'une partie du réseau devienne inactif pour la plupart des inputs. Pour éviter ce problème, nous avons décidé d'utiliser une variant de cette fonction : LeakyReLU (Rectified Linear Unit) :

- $f(x) = x \; si \; x > 0$
- $f(x) = \alpha x \; si \; x \leq 0$



De plus, après plusieurs étapes de convolutions, nous appliquons une étape de Pooling (“regroupement”) sur les features map rectifiés. L’étape de pooling permet de réduire la dimension des features extraites mais surtout d’extraire les informations spatiales importantes permettant de décrire l’image. Il existe différents types de pooling (Sum,Average, Max…) cependant le max pooling est le plus utilisé en deep learning puisqu’il permet d’extraire dans une sous région de la feature map la valeur maximale, c’est à dire l’information la plus importante.



Ainsi, cette étape de pooling est très importante afin de :

- Rendre la dimension des images de départs (Width x Height) de taille raisonnable.
- Réduire le nombre de paramètres à optimiser et contrôler l'overfitting
- Rendre le réseau robuste aux petites déformations (translations, distorsions…)

Mis ensemble, ces couches successives (Convolution + LeakyReLU & Pooling) permettent d'extraires les features importantes qui seront passé en entré d'un algorithme de décision (RandomForest, Neural Networks,...).

Voici l'architecture de base que nous avons utilisé et avec laquelle nous avons obtenus les meilleurs résultats :

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_11 (Conv2D)           (None, 95, 95, 32)        320
_________________________________________________________________
leaky_re_lu_12 (LeakyReLU)   (None, 95, 95, 32)        0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 93, 93, 16)        4624
_________________________________________________________________
leaky_re_lu_13 (LeakyReLU)   (None, 93, 93, 16)        0
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 46, 46, 16)        0
_________________________________________________________________
dropout_6 (Dropout)          (None, 46, 46, 16)        0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 46, 46, 64)        9280
_________________________________________________________________
leaky_re_lu_14 (LeakyReLU)   (None, 46, 46, 64)        0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 44, 44, 32)        18464
_________________________________________________________________
leaky_re_lu_15 (LeakyReLU)   (None, 44, 44, 32)        0
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 21, 21, 32)        0
_________________________________________________________________
dropout_7 (Dropout)          (None, 21, 21, 32)        0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 21, 21, 128)       36992
_________________________________________________________________
leaky_re_lu_16 (LeakyReLU)   (None, 21, 21, 128)       0
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 19, 19, 128)       147584
_________________________________________________________________
leaky_re_lu_17 (LeakyReLU)   (None, 19, 19, 128)       0
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 17, 17, 64)        73792
_________________________________________________________________
leaky_re_lu_18 (LeakyReLU)   (None, 17, 17, 64)        0
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
dropout_8 (Dropout)          (None, 8, 8, 64)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 4096)              0
_________________________________________________________________
dense_3 (Dense)              (None, 512)               2097664
_________________________________________________________________
leaky_re_lu_19 (LeakyReLU)   (None, 512)               0
_________________________________________________________________
dropout_9 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_4 (Dense)              (None, 68)                62073
=================================================================
Total params: 2,450,793
Trainable params: 2,450,793
Non-trainable params: 0
_________________________________________________________________
```

4) Optimisation de l'entraînement

Nous avons également procédé à de nombreux tests afin d'optimiser notre entraînement. Comme nous avons pu le voir précédemment, notre modèle contient plus de 2 millions de paramètres à entraîner, il est donc crucial de chercher à optimiser au maximum notre modèle. Lorsque l'on essaye d'améliorer le score de classification, on essaye de minimiser une fonction de perte ; la *categorical cross-entropy* défini tel que : $J = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} log(p_{ij})$

- *M : Nombre total de classes*
- *N : Nombre images*
- *y : label de l'image considéré*
- $p_{ij} \varepsilon (0, 1)$ *selon si l'image a bien été classifié ou non.*

Cette fonction de perte est très adaptée lorsque l'on veut faire de la classification multi-classe puisqu'elle est un indicateur direct de la force de prédiction de notre modèle (maximise la vraisemblance de nos données). Nous avons donc essayé de nombreux optimiseurs : Variationnelles (Descente de gradient), Stochastiques (gradient stochastique).

Les optimiseurs stochastiques sont moins sensibles à l'initialisation que le gradient variationnel et permettent d'explorer bien plus de scénarios, ce qui permet en théorie de trouver le minimum global d'une fonction. Cependant ils sont bien plus long à converger mais ceci n'est pas un problème dans notre cas, notre but étant d'améliorer le score de classification.

Après de nombreux tests, nous avons utilisé l'optimiseur SGD en utilisant la méthode des moments. En effet, fixer le pas d'apprentissage est un point important lorsque l'on fait de l'apprentissage automatique : un pas trop grand peut rendre l'algorithme divergeant, un pas trop petit le rendre trop lent à converger. L'optimiseur SGD que nous avons utilisé permet de définir le taux d'apprentissage comme une fonction décroissante avec le temps : ainsi les premières epochs produisent un plus fort changement dans les paramètres que les epochs suivantes.

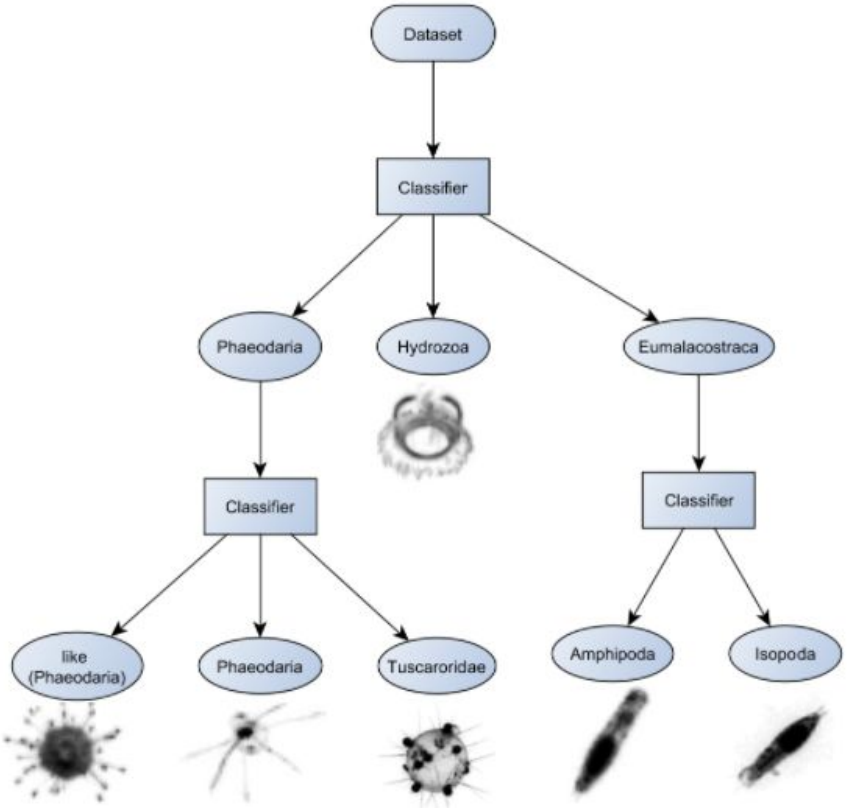
La méthode du moment directement à l'apprentissage par back-propagation. L'idée est de conserver en mémoire les différentes mises à jours effectués afin de calculer la suivante en fonction du gradient actuel et de la modification précédente. Cela permet notamment à la méthode stochastique de trop osciller (c'est à dire perte d'énergie inutile) lors de la recherche du minimum global.

II Architectures hiérarchiques

Notre ensemble de données comprend des images d'une soixantaine d'espèces de plancton ainsi qu'une taxonomie des espèces de planctons présentes. Nous souhaitons exploiter cette connaissance pour affiner nos modèles et possiblement améliorer les résultats de classifications.

1) Classifieur en arbre

Le classifieur en arbre (Alshdaifat, Coenen, and Dures 2013) reprend la structure de la taxonomie. Une taxonomie est simplement un arbre qui classe les espèces selon des caractéristiques génétiques. On considère que les espèces proches génétiquement se ressemblent. Un classifieur en arbre est constitué de plusieurs classifieurs organisés en arbre. Chaque classifieur est entraîné pour classifier une partie de l'arbre. Ils sont constitués d'un bloc CNN pour l'extraction de features et d'un bloc MLP pour la décision. Chaque classifieur développe des features spécialement pour distinguer la partie de l'arbre qui lui correspond. Le classifieur du premier niveau développera des caractéristiques grossières qui permettent de distinguer des ensembles de classes. Et les classifieurs de dernier niveau développeront des caractéristiques fines pour distinguer des classes similaires.

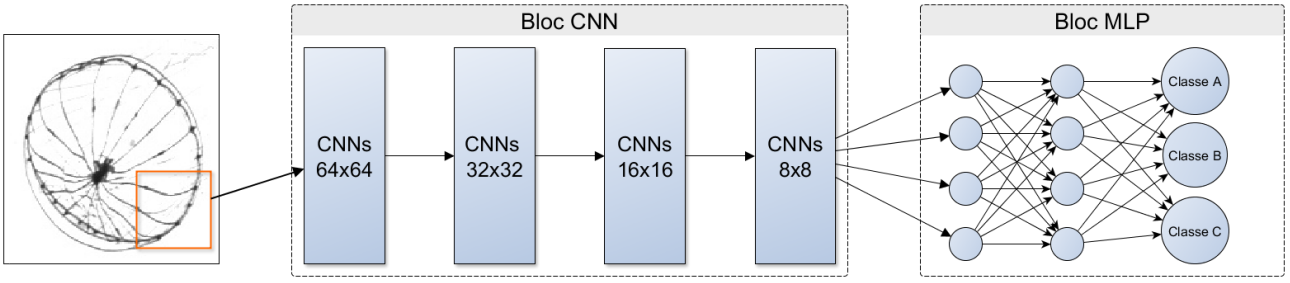


Pour la phase d'apprentissage le classifieur en arbre est bien plus long à entraîner que sa version classique. En effet, il est nécessaire de reprendre la totalité du dataset d'entrainement pour chaque niveau de l'arbre. Le classifieur est donc exactement k fois plus long pour l'apprentissage que l'équivalent classique avec k = le nombre de niveaux de l'arbre.

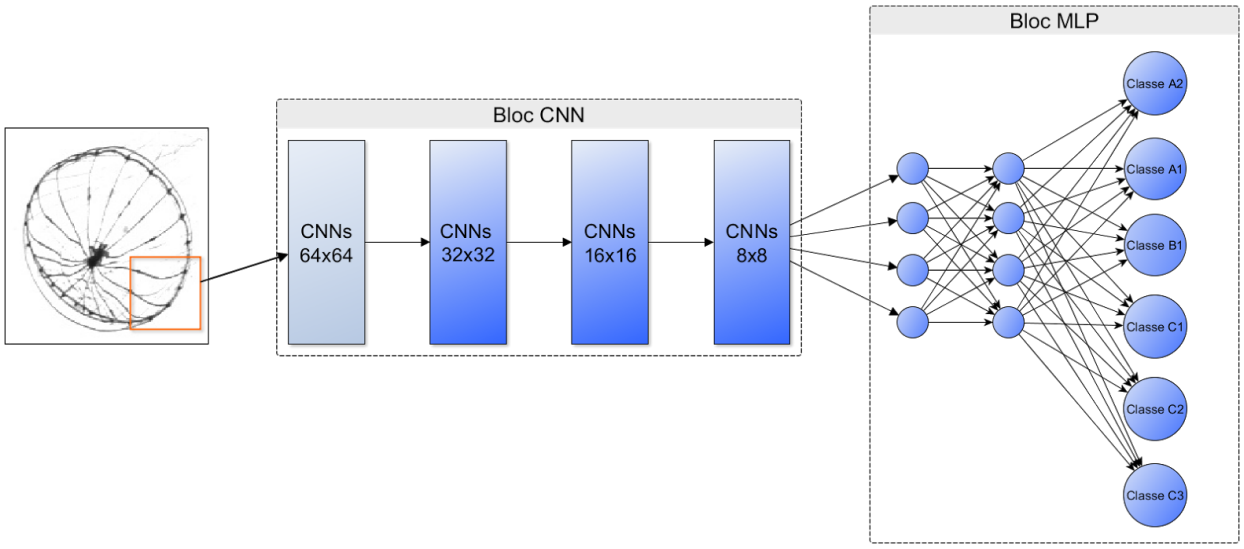
Une fois l'apprentissage terminé, il faut passer l'image dans un classifieur à chaque niveau de l'arbre pour connaitre la classe d'une image. Les résultats sont équivalent ou moins précis qu'un modèle à plat après entrainement. Cependant, il est possible d'améliorer plus facilement les résultats, car il est assez simple d'identifier les sous-arbres faisant baisser les résultats généraux et de fine-tuner ces modèles spécifiquement.

2) Classifieur à niveaux

Le classifieur "à niveaux" est un classifieur modifié à chaque niveau de la taxonomie. L'objectif est de reprendre la structure de la taxonomie pour entraîner les filtres CNNs. Les filtres du niveau k seront entrainés pour reconnaître les features du niveau k de la taxonomie. Sa construction s'effectue en plusieur étapes. Tout d'abord, on construit un réseau de neurones naïfs constitué d'un bloc de CNNs et d'un bloc MLP. On entraine le réseau à classifier le niveau le plus haut de la taxonomie. Après avoir atteint un certain score de classification on conserve les poids de la première couche de CNNs et on reprend des neurones naïfs pour le reste de l'architecture. On relance l'entrainement sur le niveau suivant de la taxonomie et ainsi de suite jusqu'à avoir assimilé l'ensemble de la taxonomie.



Architecture naïve entraînée sur les superclasses



On conserve la première couche de l'architecture et on relance l'apprentissage sur les sous-classes.

A la suite de l'apprentissage, le bloc CNN a appris les features les plus pertinentes à chaque niveau hiérarchique et est censé être capable de donner une réponse fiable. Par manque de temps nous n'avons pas pu tester convenablement cette architecture.

3) Hiérarchie basé sur les erreurs de classifications

Après avoir proposé plusieurs hiérarchies basées sur la taxonomie de nos espèces de plancton, nous voulions déterminer si une structure qui prendrait compte des résultats de notre classifieur à plat, pourrait donner de meilleurs résultats. L'objectif est de regrouper plusieurs classes partagent de fortes erreurs de classifications entre elles. Si par exemple deux classes sont considérées comme difficiles à distinguer l'une de l'autre par le classifieur, elles seront regroupées afin d'être traitées par un classifieur uniquement dédié à leur distinction.

Afin d'interpréter les résultats de nos classifieurs, on utilise des matrices de confusion où les coefficients C[i,j] donnent la fréquence de classification d'une image en classe j, sachant qu'elle appartient réellement à la classe i. L'objectif de notre nouvelle architecture, serait de créer des groupes en fonction de la matrice de confusion. Intuitivement des classes i et j tel que C[i,j] et C[j,i] ont un fort indice nécessiteraient qu'on les entraîne à nouveau puisque le classifieur n'a pas réussi à bien les distinguer. On espère qu'un classifieur entraîné à différencier ces deux espèces uniquement , aurait de meilleurs résultats sur ces deux espèces que le classifieur assigné à différencier toutes les espèces à la fois.

Le principe est donc de créer des clusters d'espèces ayant de fort indices de confusions entre elles, on utilise ainsi la matrice de confusion comme un graph de similarité. Pour ce

faire on peut se baser sur la théorie des graphs et notamment des méthodes de graph cut, ou bien d'autres méthodes de clustering telle que Hierarchical clustering, K-Mean ou Spectral Clustering. Beaucoup de ces méthodes nécessitent que les relations de similarité soient réciproques, soit C[i,j] = C[j,i], ce qui n'est pas forcément le cas. On va donc rendre la matrice symétrique en l'additionnant à sa transposée. Cependant pour privilégier des relations réciproques (C[i,j] ≈ C[j,i]) par rapport à des relations unilatérales (C[i,j] >> C[j,i]) on multiplie tous les coefficients de la matrice de confusion et de sa transposé par un facteur **α** < 1 avant de les additionner.

```
[[ 0.7   0.1   0.1   0.025 0.025 0.025 0.025]
 [ 0.09  0.7   0.11  0.025 0.025 0.025 0.025]
 [ 0.15  0.05  0.7   0.025 0.025 0.025 0.025]
 [ 0.05  0.05  0.05  0.7   0.05  0.05  0.05 ]
 [ 0.05  0.05  0.05  0.05  0.4   0.35  0.05 ]
 [ 0.05  0.05  0.05  0.05  0.4   0.35  0.05 ]
 [ 0.05  0.05  0.05  0.05  0.05  0.05  0.7  ]]
```

Exemple de matrice de confusion

```
[[ 0.836  0.308  0.351  0.190  0.190  0.190  0.190]
 [ 0.308  0.836  0.277  0.190  0.190  0.190  0.190]
 [ 0.351  0.277  0.836  0.190  0.190  0.190  0.190]
 [ 0.190  0.190  0.190  0.836  0.223  0.223  0.223]
 [ 0.190  0.190  0.190  0.223  0.632  0.612  0.223]
 [ 0.190  0.190  0.190  0.223  0.612  0.591  0.223]
 [ 0.190  0.190  0.190  0.223  0.223  0.223  0.836]]
```

Matrice de confusion symétrique obtenue

L'approche retenue pour obtenir nos groupes d'espèces se base sur l'algorithme Hierarchical Clustering. À partir de la matrice symétrique de confusion définie ci-dessus, on fixe tous les coefficients inférieurs à un certain seuil de confusion à 0 (le seuil de confusion est choisie par l'utilisateur et représente l'erreur de prédiction limite pour laquelle on considère qu'un nouvel entraînement est nécessaire).

```
                                  m
[[ 0.836  0.308  0.351  0.     0.     0.     0.   ]
 [ 0.308  0.836  0.165  0.     0.     0.     0.   ]
 [ 0.351  0.165  0.836  0.     0.     0.     0.   ]
 [ 0.     0.     0.     0.836  0.     0.     0.   ]
n [ 0.     0.     0.     0.     0.632  0.612  0.   ]
 [ 0.     0.     0.     0.     0.612  0.591  0.   ]
 [ 0.     0.     0.     0.     0.     0.     0.836]]
```

[0, 1, 2, 3, 4, 5, 6]

Une fois cette opération effectuée on va chercher le coefficient le plus élevé de la matrice qui ne soit pas sur la diagonale C[n,m], n ≠ m (on ne cherche que dans la partie triangulaire supérieure pour éviter des répétitions de calculs). On vas alors regrouper les espèces n et m entre elles et modifier l'espace vectorielle afin que la composante n devienne une combinaison linéaire des composantes n et m (on supprime également la composante m de la matrice) :

```
[[ 0.836  0.308  0.351  0.     0.     0.   ]
 [ 0.308  0.836  0.277  0.     0.     0.   ]
 [ 0.351  0.277  0.836  0.     0.     0.   ]
 [ 0.     0.     0.     0.836  0.     0.   ]
 [ 0.     0.     0.     0.     0.612  0.   ]
 [ 0.     0.     0.     0.     0.     0.836]]
[0, 1, 2, 3, [4, 5], 6]
```

On réitère ensuite cette opération jusqu'à ce qu'il n'y ai plus aucun indice strictement positif dans la partie triangulaire supérieur de la matrice et on aura obtenue nos clusters, sur lesquelles s'appuiera notre nouvelle hiérarchie.

```
[[ 0.594  0.292  0.     0.     0.   ]
 [ 0.292  0.836  0.     0.     0.   ]
 [ 0.     0.     0.836  0.     0.   ]
 [ 0.     0.     0.     0.612  0.   ]
 [ 0.     0.     0.     0.     0.836]]
[[0, 2], 1, 3, [4, 5], 6]
```

```
[[ 0.487  0.     0.     0.   ]
 [ 0.     0.836  0.     0.   ]
 [ 0.     0.     0.612  0.   ]
 [ 0.     0.     0.     0.836]]
[[0, 2, 1], 3, [4, 5], 6]
```

Qui plus est, même si la hiérarchie obtenue n'est que sur deux étages, on peut réitérer ce processus sur les nouvelles matrices de confusions obtenue après l'entrainement du classifieur hiérarchique. On obtiendra alors une hiérarchie à trois étages et on pourra toujours continuer à étendre notre arbre si nécessaire.

Discussions sur les modèles hiérarchiques

Plusieurs architectures hiérarchiques ont déjà été étudiée pendant le projet. Cependant, d'autres pistes peuvent encore être explorés:

- D'autres architecture hierarchiques peuvent être construites : deux réseaux parallèles (un réseau dédié aux détails et l'autre aux éléments communs) couplé à un ensemble de réseaux de neurones. (Yan et al. 2015)
- Toutes nos solutions hiérarchiques sont composées de CNNs pour l'identification des features et d'un MLP pour la décision. L'utilisation de l'algorithme random forest en décision pourrait donner de meilleurs résultats.
- Pour le classifieur à niveau il serait possible d'optimiser l'entrainement en définissant une fonction de loss différente pour chaque couche. Cela permettrait d'entrainer chaque couche indépendamment des autres puisque chaque couche a un objectif différent.
- Pour le classifieur à niveau il est possible de conserver plus d'une couche dans la hiérarchie à chaque étape.
- Enfin pour la hiérarchie basée sur des matrices de confusions, d'autres algorithmes de génération de clusters tel que le K-Mean pourrait être testé. En effet, plutôt que de créer les classes avec un critères de seuil, on pourrait voir si spécifier un nombre de groupe pourrait amener à des résultats comparables.

III Segmentation de l'image

1) Segmentation par détection des contours

Une autre problématique soulevée au cour de ce projet fut la présence dans les données d'entrainements d'images contenant de multiples objets, pouvant eux même appartenir à des espèces différentes. Un CNN classique ne pouvant attribuer qu'un seul label par image, l'entrainement du classifieur nécessite tout comme la classification, qu'il n'y ai qu'une seule espèce présente sur l'image pour ne pas fausser les résultats du classifieur ou laisser des ambiguïtés sur le label lors de l'entrainement.
Qui plus est, dans jeux de données mis à notre disposition, les objets ont la propriété d'être noir sur fond blanc, ce qui les rend facile à séparer avec des librairies tel que scikit-image ou OpenCV qui permettent de trouver et comparer les contours (généralement pour la data augmentation on utilise uniquement l'objet ayant le plus grand contour).

Image d'origine *Seuillage* *Sélection de l'objet*

2) Besoin d'une segmentation par Deep Learning

Néanmoins la présence d'objet superposés pose problème. Pour l'entrainement on peut se permettre de manuellement enlever les rares images comportant des objets superposés en amont de la Data augmentation; cependant pour la classification, il faut une segmentation intelligente. Il existe de nombreux algorithmes permettant une segmentation par Deep Learning : R-CNN, Convolutional Sliding Windows YOLO, SegNet, WildCat…(Durand et al. 2017; Redmon et al. 2016)

Nous avions choisis en début de projet l'algorithme YOLO qui nécessite pour son entrainement des labels facile à générer (cadre rectangulaire contenant l'objet en plus du nom de l'espèce sélectionnée) et disposait d'une implémentation Open source sur Python nommé DarkFlow.

Puisque nous disposions déjà d'une méthode relativement efficace et peu coûteuse en temps de calculs pour repérer des objets ponctuels dans l'image, nous nous sommes concentré sur les cas difficiles d'objets superposés. Pour ce faire nous avons utiliser la Data Augmentation pour générer des images contenant plusieurs individus disposés de façon aléatoire; et donc éventuellement superposés.

Néanmoins, malgré plusieurs essaie avec des paramètres d'entrainements différents, les résultats de l'algorithme YOLO furent bien moins bon que ceux d'un CNN classique. Sur le Set Kaggle (121 classes) le meilleur score ne dépasse pas les 11 % de précision, sachant qu'en plus l'algorithme ne repère pas forcément les objets dans l'image (pour la meilleur précision, le taux de détection toutes classes confondus était d'environ 28 %). Le premier problème constaté fut que le classifieur prédisait la majorité des images par une dizaines de labels prépondérants. Ces labels correspondaient à certaines des espèces les plus massives présentes dans le Data Set. Sachant que l'erreur de classification dans la fonction coût est proportionnelle à la taille de l'objet dans l'image, nous avons pensé que lors de l'entrainement le réseau avait convergé en donnant plus de poids aux espèces délimiter par plus de pixels.

Pour résoudre ce problème nous avons tenté deux solution, la première étant de redimensionner tous les objets pour qu'il aient une taille similaire en pixels, ce qui est équivalent à modifier la fonction de coût dans l'implémentation de l'algorithme. En revanche avec cette option on perd la possibilité de reconnaître les objets par leur taille et bien que le taux de détection du réseau augmente pour des seuils de détections d'objets équivalents, on n'améliore pas vraiment la précision du classifieur.

La deuxième idée était d'entraîner le classifieur pour plusieurs groupes d'espèces basé sur leur taille moyenne en pixels. Pour cette effet nous avons utilisé un algorithme K-Mean et nous avons pris comme donnés la racine carré de la surface moyenne de chaque espèces. Les résultats était un peu meilleurs mais c'était surement dû au fait que les classifieurs avaient moins d'espèces à classifier (avec 6 clusters le groupe avec la plus grande aire ne comporte que 8 espèces, on est loins des 121 espèces dont nous disposions au débuts). Qui plus est pour savoir quel classifieur utiliser pour un objets il est nécessaire de mesurer sa taille, et on retombe encore une fois sur le même problème que sont les objets superposés puisqu'ils faussent la mesure de l'aire).

Conclusion

Ce projet nous a permis de nous initier aux méthodes de Deep-Learning et aux problématiques qui en découlent. Nous avons notamment été confronté aux temps de calculs nécessaires à l'entraînement de nos architectures (parfois plusieurs jours) ce qui nous a poussé à adapter notre méthode de travail afin d'avancer pendant des phases où nous attendions les résultats. Nous pouvions entre autre rechercher dans des articles scientifiques de nouvelles architectures, méthodes d'optimisations ou stratégie pour améliorer notre dataset, toujours dans le but d'améliorer les résultats de la classification.

Cela nous a également permis d'utiliser pour la première fois un cluster de calcul, celui de l'INRIA et de comprendre les problématiques qui entourent la soumission de tâches. Par ailleurs certaines librairies ont nécessité que nous configurons nous-même notre environnement de travail (CUDA entre autre) pour effectuer des tests avant d'entrainer notre architecture sur le cluster de l'INRIA.

Au cours de ce projet nous avons également été directement en contact avec un océanographe (Jean-Olivier Irisson) et nous avons ainsi découvert les problématiques liées aux attentes/demandes client lorsque celui-ci vient d'horizon différent. Il a parfois fallu prendre la décision d'abandonner certaines classes telles que certains déchets qui ont affecté très négativement la précision de nos classifieurs ou encore des classes très insuffisamment représentés.

Nous avons aussi dû plusieurs fois remettre en question notre méthodologie pour savoir comment interpréter nos résultats, ou encore savoir si le calcul de la précision de notre classifieur était fiable (indépendance entre jeux d'entrainement et jeux de test,). De plus, quand les résultats de certains entraînement n'étaient pas à la hauteur de nos espérances, il a fallu chercher d'où pouvait provenir le problème et trouver un moyen d'y remédier. Ce fut d'autant plus difficile que pour tester une nouvelle solution il fallait souvent attendre parfois plusieurs jours.

Glossaire

CNN : Convolutional Neural Networks

Epoch : Une epoch correspond à un parcours complet de notre jeu de donnée.

Feature ou Caractéristique : Forme caractéristique propre à une classe ou un ensemble de classe. Présente dans une image et correctement identifiée, elle participe à la décision du classifieur.

Modèle hiérarchique : Modèle utilisant une hiérarchie entre les images en plus de la classification

Modèle à plat : A l'inverse du modèle hiérarchique, le modèle à plat ou modèle classique ne prend pas en compte de hiérarchie entre les images. C'est à dire que toutes les images sont au même niveau de profondeur.

MLP : Multi Layer Perceptron

LOV : Laboratoire Océanographique de VilleFranche-Sur-Mer

RCNN : Region based CNN.

YOLO : You Only Look Once, algorithme permettant de détecter des objets en temps réel.

Références

Durand, Thibaut, Taylor Mordan, Nicolas Thome, and Matthieu Cord. 2017. “WILDCAT: Weakly Supervised Learning of Deep ConvNets for Image Classification, Pointwise Localization and Segmentation.” In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. https://doi.org/10.1109/cvpr.2017.631.

Redmon, Joseph, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. “You Only Look Once: Unified, Real-Time Object Detection.” In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. https://doi.org/10.1109/cvpr.2016.91.

Yan, Zhicheng, Hao Zhang, Robinson Piramuthu, Vignesh Jagadeesh, Dennis DeCoste, Wei Di, and Yizhou Yu. 2015. “HD-CNN: Hierarchical Deep Convolutional Neural Networks for Large Scale Visual Recognition.” In *2015 IEEE International Conference on Computer Vision (ICCV)*. https://doi.org/10.1109/iccv.2015.314.

Karen Simonyan & Andrew Zisserman, ICLR 2015, “Very deep convolutional neural networks for large scale image recognition”

Ilya Sutskever, James Martens, George Dahl & Geoffrey Hinton, “On the importance of initialization and momentum in deep learning”

Ruslan Salakhutdinov, Joshua B. Tenenbaum, and Antonio Torralba, Member, *IEEE, TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, VOL. 35, NO. X, XXXXXXX 2013, “Learning with Hierarchical-Deep Models”*