Telecom SudParis

Projet Informatique 1ère Année
PRO 3600

Thomas COUVAL, Victoire GORGE, Iwan IBNOULOUAFI, Maxence LASBORDES

Reconnaissance d'espèces marines à l'aide d'une intelligence artificielle

Enseignant responsable : Badran Raddaoui

Table des matières

1 Introduction

L'Australie est un pays remarquablement beau; sa barrière de corail est la plus impressionnante au monde. En effet, la plus grande, elle abrite quelque 1500 espèces de poissons, 400 espèces de coraux et tout juste 130 espèces de requins. Elle compte un écosystème maritime très vaste. Cependant elle est en danger de part le réchauffement climatique et d'autre part la présence de COTS une espèce de poissons mangeurs de corail. Mais des groupes de scientifiques, touristes et locaux se sont donnés la missions de contrôler dans un premier temps l'évolution de la population de mangeurs de corail pour prédire leur effet sur cet écosystème et plus particulièrement sur la barrière de corail. Pour, dans un deuxième plan, établir un plan durable pour la barrière et ses habitants. Pour cela, à savoir recenser les populations mangeurs de corail, ils utilisent une méthode appelée "Manta Tow". Elle consiste à faire des prélèvements sous forme photos des fonds marins de la barrière de corail tous les 200 mètres. Alors qu'elle permet de visualiser clairement l'évolution du corail, elle reste très limitée; Il est difficile d'établir un modèle global à toute la barrière, La quantité de données est telle que garder des archives et les analyser reste très fastidieux. Enfin les données restent rudimentaires quant à l'élaboration d'études poussées.

C'est pourquoi notre projet informatique consistera à créer une intelligence artificielle capable, à partir de cette base de données, de detecter, suivre et identifier les espèces nocives pour les coraux au cours du temps. L'interet étant de simpliciter la tâches aux plongeurs et aux spécialistes quand à la cartographie de ces especes dans le fond marin.

2 Cahier des charges

Notre objectif est d'écrire une application permettant, à partir d'une vidéo source, d'identifié et de suivre dans le temps les COTS présents.

2.1 Back-end

Le back-end constitue les fondations de notre outil: il est le programme permettant de trouver les COTS dans une image. Il nous faut donc, pour celui-ci, construire une structure qui combine algorithme de reconnaissance d'image et réseaux de neurones afin de pouvoir identifier, tracker et classer les COTS présents dans flux vidéos entrés.

Pour le back-end, les contraintes sont les suivantes :

- Le temps moyen d'identification et de classification des COTS en image doit être inférieur à **25 secondes** (afin que la reconnaissance reste dans le domaine du "temps réel") et sont taux de réussite d'au moins **55%**.
- Le temps d'entrainement des modèles de réseaux pour l'identification et la classfication ne doit pas dépasser les **10-15 heures** (afin que le data-set puisse être mis-à-jour la nuit après chaque expedition)
- Le language de développement choisi est le Python

2.2 Front-end

En terme de front-end, nous souhaitons dans un premier temps limiter l'interface à un terminal pour des raisons de practicité. Dans un second temps, nous envisageons de porter le logiciel sur une interface web afin de faciliter l'utilisation à tout type d'appareils (mobile, laptop, ...).

Pour ce front-end, les contraintes sont les suivantes :

- L'interface web doit-etre simple, ergonomique et pertinente: Drag-and-drop feature, affichage claire des resultats et performance des algorithmes, ...
- Il doit être possible d'utiliser un flux vidéo en continue *(webcam)* comme source d'entrée.
- Iteraction simple et robuste entre front-end et back-end.
- Les languages de développement choisis sont HTML, CSS et JS.

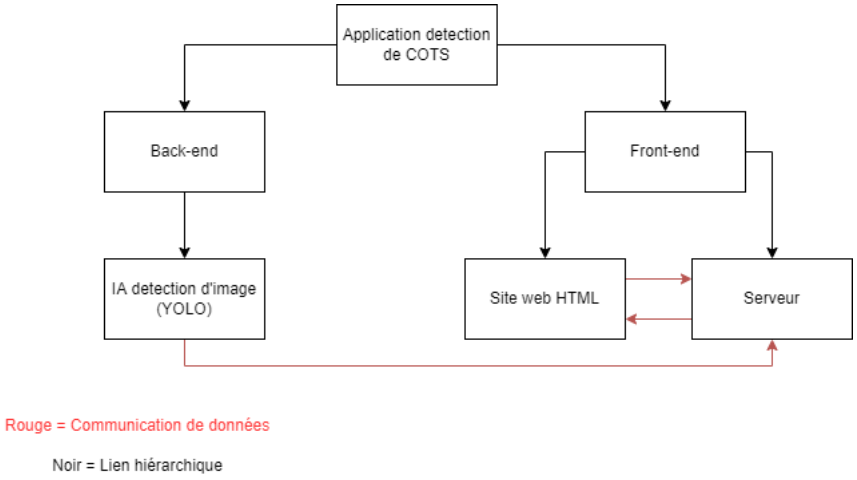

Figure 1: Structure de notre logiciel

3 Développement

Comme énoncé précedemment, l'objectif de notre back-end est de détecter, classifier et tracker les COTS présents dans les vidéos sous-marines qui constituent notre data-set. Pour cela nous avons envisagé deux structures différentes (YOLO et Faster-RCNN) que nous mettons en compétition afin de determiner le back-end le plus pertinent à notre étude. Toutes deux utilisent des réseaux de neurones.

Avant de présenter notre implementation de ces 2 structures, le contenu brut qui constitue notre data-set doit être préalablement traité afin de rendre les opérations futurs plus simple et robuste.

3.1 Préparation des données

Les datas que nous utilisons pour entrainer et evaluer nos modeles sont fournis par la Great Barrier Reef Foundation (GBRF). Ces datas se présentent sous la forme de deux type de fichier:

- Des suites de photos sous-marines qui constituent des vidéos.
- Des fichiers .csv (comma seperated value) qui donne pour chaque image la position et la taille des COTS présents dans celle-ci.

Combinées, ces données peuvent être visualisées comme ceci:

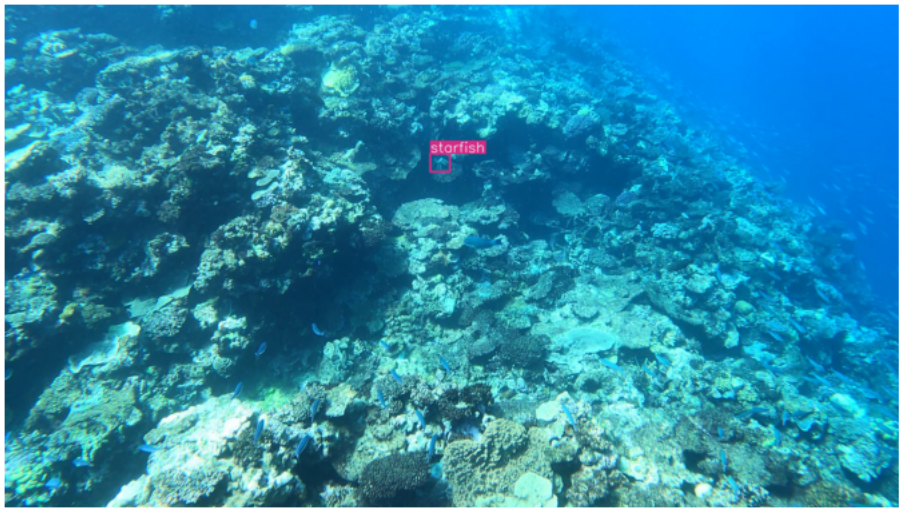

Figure 2: Exemple des sorties attendues de notre programme

Etant donné que chaque photo ne possède pas forcement de COTS, il nous faut, dans un premier temps, extraire celles en possédant afin qu'elle puisse être utilisé pour l'entrainement des réseaux de neurones.

```
import pandas as pd

train = pd.read_csv('train.csv')
train = train.loc[train['annotations'] != '[]']

def create_new_dataset():
    sub_train = train.loc[:,['video_id','video_frame']]
    video_pos = []
    for i in sub_train.values:
        video_pos.append('import/train_images/video_' + str(i[0])
        + '/' + str(i[1]) + '.jpg')
    for i in tqdm(range(len(video_pos))):
        shutil.copy(video_pos[i],'data_set/' + str(i) + '.jpg')

create_new_dataset()
```

Maintenant, il nous faut changer le format .csv des annotations car il ne convient pas à la plateforme que nous utilisons pour entrainer les modèles. On préferera utiliser ici le format YOLOv5 Pytorch txt. Voici donc comment la convertion est réalisée.

```
def parse_annotation(input_annotation):
    input_annotation = input_annotation.split(',')
    output = []
    for i in input_annotation:
        current_value = ''.join(c for c in i if c.isdigit())
        if(len(list(current_value)) > 0):
            output.append(int(current_value))
    return output

def create_annotation_files():
    for i in tqdm(range(train.shape[0])):
        current_row = train.iloc[i].values[-1][2:-1].split('{')
        current_file = open('annotations/' + str(i) + '.txt','w')

        for j in current_row:
            current_row_annotation = parse_annotation(j)

            new_x = np.interp(current_row_annotation[0],[0,1280],[0,1])
            new_y = np.interp(current_row_annotation[1],[0,720],[0,1])
            new_width = np.interp(current_row_annotation[2],[0,1280],[0,1])
            new_height = np.interp(current_row_annotation[3],[0,720],[0,1])

            new_x += new_width/2
            new_y += new_height/2

            current_file.write('starfish ' + str(new_x) + ' ' + str(new_y) +
            ' ' + str(new_width) + ' ' + str(new_height) + '\n')

        current_file.close()

create_annotation_files()
```

Maintenant les fichiers bien annotés, nous uploadons les images et leurs annotations respectives sur la plateforme Robotflow. Cette plateforme nous permet d'abord de diviser notre data-set en 3:

- 70% des images sont utilisées pour entrainer les modèles
- 20% des images sont utilisées pour évaluer les modèles
- 10% des images sont utilisées pour tester l'inférences des modèles

Dans un second temps, Robotflow nous permet de modifier les images afin d'accélerer nos processus. En effet, une manière simple de réduire le temps d'entrainement et d'execution de nos IA et de réduire la taille de leurs entrées. Ainsi en diminuant la résolution de images, on optimise la performance temporelle de notre programme. Cependant il faut faire attention: en effet trop réduire la dimension des entrées peut

biaiser nos resultats. Ici, les COTS sont generalement de petite taille donc la compression choisi sera faible.

Enfin, Robotflow permet une utilisation simple des modèles d'IA que nous étudions notamment grâce à son interaction avec la plateforme Google Collab sur laquelle nous entrainons nos modèles.

Maintenant les données traitées, nous allons présentés nos resultats sur les deux approches choisies.

3.2 1ère approche : Faster R-CNN

3.2.1 Fonctionnement théorique

La première méthode envisagée est d'utiliser Faster R-CNN développé en 2015 notamment par Ross Girshick. Celle-ci se décompose en plusieurs parties. Dans un premier temps, notre entré qui est une photo RGB (représenté par un tenseur de dimension $n \times m \times 3$) est passé dans un CNN (ou réseau de neurone convolutif) afin d'obtenir en sortie une feature map. Comme l'indique son nom, une feature map est une "image" qui représente de manière abstraite les caractéristique d'une image tel que la présence de coin, de courbure, de lignes, etc... Ces caractéristiques sont transcrite sous la forme d'un champ scalaire de la taille de l'image.

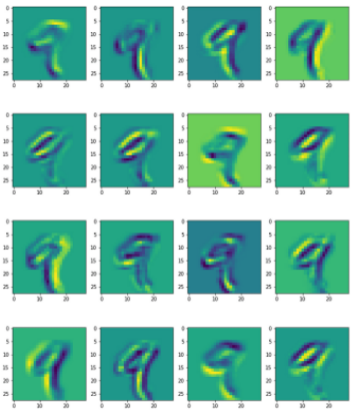

Figure 3: Exemple d'une feature map extraite grace à un CNN

Pour arriver à un tel resultat, le CNN procède de la manière suivante : il passe l'entrée à travers la structure suivante

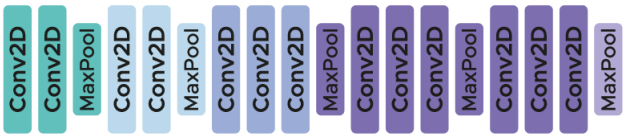

Figure 4: Structure du CNN VGG16 et de ses différentes couches

Dans un premier temps, le CNN passe l'entré dans une série de couche de convolution en cascade permettant d'extraire certaines caractéristiques de l'image. Chacune de ces couches applique plusieurs noyaux de convolution (ou filtres) sur leur entrée, noyaux dont les paramètres sont initialement aléatoires mais par la suite ajustés lors de la phase d'entrainement du réseau.

Par la suite, la sortie des premières couches de convolution est passée par une couche de max-pooling. En théorie, cette couche permet de reduire la taille des features map en sortie des couches de convolution afin d'optimiser les performances d'une part, et de ne garder que les informations pertinentes d'autre part. En pratique, on segmente les features maps selon un grillage uniforme puis, dans chaque case, on ne garde que la valeur de la feature map ayant la plus grande valeur.

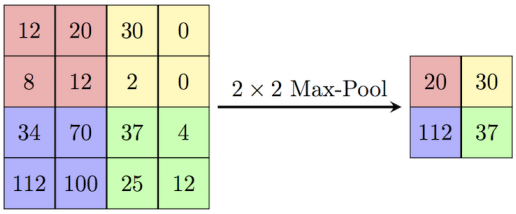

Figure 5: Entrée et sortie d'une couche de max-pooling

Ces différentes couches de convolution et de max-pooling sont mise en cascade pour finalement former notre CNN. L'interet d'une telle structure est de pouvoir, tout le long du parcours, obtenir une decomposition de plus en plus abstraites de notre image d'entrée. Cela permet d'obtenir des données informatiquement plus pertinentes et donc plus traitables pour determiner la classe et la position de nos objets (ici les COTS).

On utilise ici le CNN VGG16 qui est la pointe de la technologie des CNN en terme de reconnaissance d'image. Ainsi le nombre de couche et leurs paramètres nous sont dictés par l'architecture de VGG16.

Maintenant que l'on possède de quoi classifier les potentiels COTS présents dans l'image (grâce à la feature map), il nous faut aussi une méthode pour déterminer la position des COTS dans l'image au préalable. Pour cela, Fast-RCNN propose l'utilisation d'un second réseau de neurones intitulé RPN (pour réseau de proposition de région). Ce dernier prend en entré la même feature map issue du CNN explicité plus haut puis donne en sortie une liste de boites encadrants les possibles objets présents en images. Chaque boite possède 4 coordonnées : (x_1, x_2, w, h). Le fonctionnement du RPN étant particulièrement compliqué, nous passerons son explication.

Pour finir, pour chaque boite-objet proposée par RPN, nous prenons la partie de la feature map correspondante (ROI pooling) que nous entrons dans deux fully-connected layer network en parallèle (réseau de neuronne classique) afin d'obtenir d'une part la distribution de probabilité des classes (ici COTS ou background) à l'aide d'un soft-max, et d'autre part les coordonées finales de la boite encadrant l'objet.

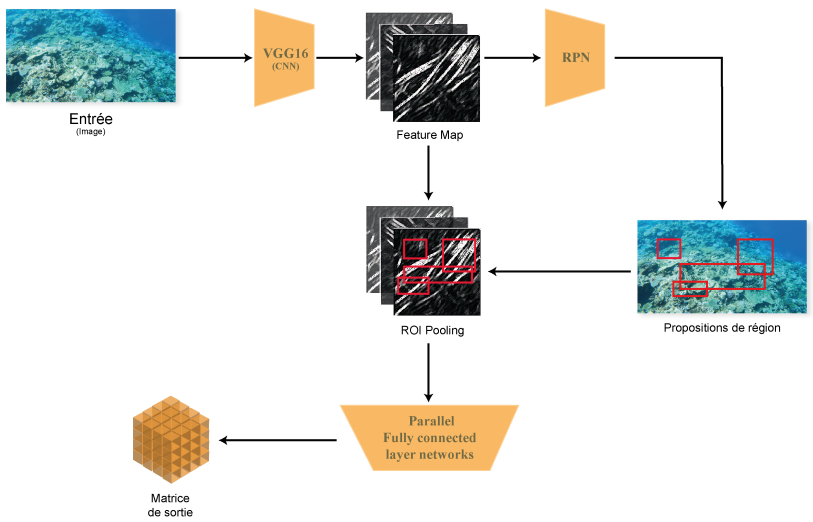

Figure 6: Structure complète de Faster-RCNN et de ses différentes étapes

3.2.2 Application pratique

Pour son implementation nous utilisons comme cité précedemment la plateforme Google Colab. En effet celle-ci confère deux grands avantages:

- Elle offre des ressources TPU et GPU conséquentes offrant une puissance de calcul importante lors de l'entrainement du modèle.
- Elle offre des notebooks déjà préparer pour entrainer Faster-RCNN sur notre dataset.

Pour l'entrainement du modèle, nous avons choisi les paramètres suivants:

- batch-size *(nombre de photo passée avant de mettre à jour les poids du réseaux)* $= 4$
- epoch *(nombre de fois ou on passe l'entièreté du dataset dans le network)* $= 120$
- learning-rate *(Pas utilisé dans le calcul du gradient de la fonction de coût lors de la backpropagation)* $= 0.001$

En réalité, le learning-rate varie au cours de l'entrainement du modèle afin de ne pas converger vers un minimum local de la fonction de coût. Le learning-rate affiché ici est celui de départ.

Après deux heures d'entrainement du modèle et une inférence sur la partie validation du data-set, on obtient les metrics suivant:

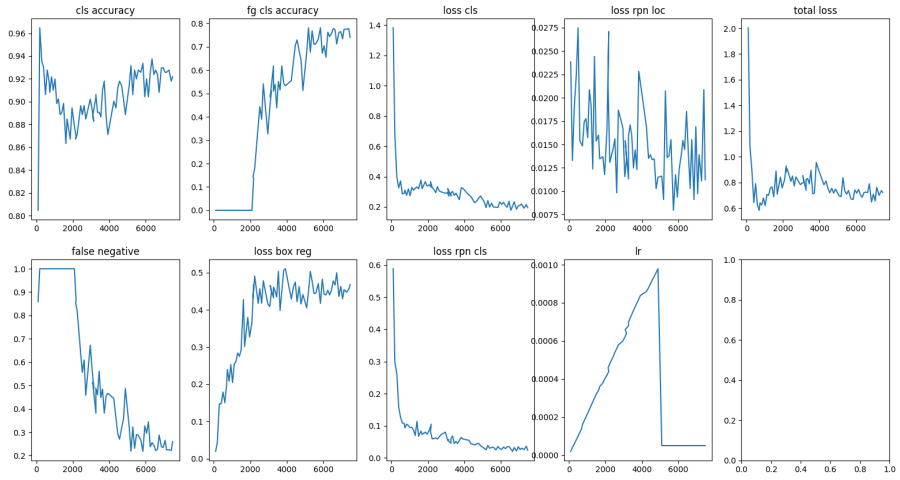

Figure 7: Metrics de notre modèle Faster-RCNN

On observe dans un premier temps que *loss rpn cls* signifiant Loss Region Proposal Network classification (on parle donc ici de la fonction permettant d'évaluer la qualité du RPN à trouver dans l'image la présense de starfish. Loss RPN CLS n'est pas à confondre à IoU (Intersection over union) qui lui permet de mesurer la précision des bounding box du RPN) décroit drastiquement aux premières epochs avant de converger après le coude passé. On comprend la que le réseaux apprend rapidement et converge des 1500 epochs vers un minimum locaux de la fonction de cout. Il en est de même pour Loss CLS qui donne la fonction de coût de la classification de objets (à savoir si ils sont des COTS ou non).

Par ailleurs, on observe que CLS ACCURACY (Classification Accuracy) et FG CLS ACCURACY (Foreground Classfication Accuracy) converge lentement mais surement vers une valeur stable réfletant la capicité du modèle à bien classifié les COTS détecter par le RPN.

Enfin, il nous reste à correler l'ensemble de ces courbes avec celle False Negative. En effet, malgrés que notre network arrive à génerer de nombreuses bounding-box et à bien classer ces dernières, on ne sait pas si les COTS détéctées sont dans les data-set mais surtout si elles ont toutes étaient détectéess. False negative permet de répondre à cette question en calculant le nombre de faux positifs produit par le réseau. On voit ici que la courbe décroit avant de converger vers la valeur de 0.2. Ainsi, notre IA detecte correctement les COTS présents en image mais réalise des erreurs malgrés tout (ici de l'ordre de 20%) .

Figure 8: Exemple de sortie donnée par Faster-RCNN

Enfin, lors de l'inférence sur la portion test de notre data-set, on note un temps moyen de detection de 0.2 s par image. En conclusion, Faster-RCNN est un modèle pertinent de détection d'image permettant de répondre à notre cahier des charges. Dans la partie 3.4, nous allons le comparer à YOLO.

3.3 2ème approche: YOLO

3.3.1 Fonctionnement théorique

La seconde approche envisagée est d'utilisé la structure YOLO développé courant 2015. Celle-ci se décompose en deux parties: dans un premier temps, comme Faster-RCNN, l'image est passé dans un CNN afin d'en extraire une feature map. Dans un second temps, au lieu d'extraire des regions d'interets pour savoir ou sont les objets qui nous interessent en image, YOLO divise l'image par une grille de dimension $S \times S$.

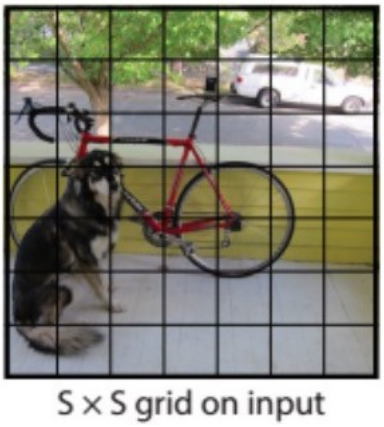

Figure 9: Exemple d'une subdivision de l'image en un grille

A chacune des cellules de la grille est associé une seule classe et B boites-objet. Ainsi la seconde partie de YOLO trouve pour chaque cellule la classe d'objet associé et la boite-objet la plus probable qui encadre l'objet présent en cellule. Informatiquement, on réalise cette étape à l'aide d'un fully connected layer en prenant pour input la section de la feature map présent dans la cellule qu'on étudie.

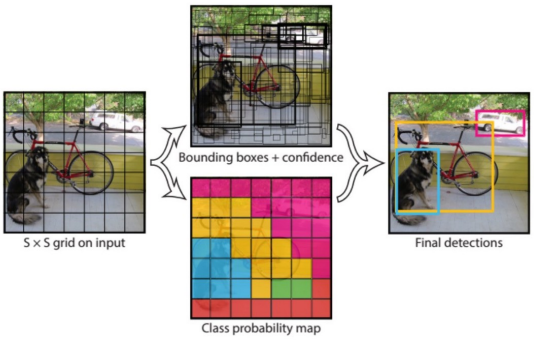

Figure 10: Différentes etapes de YOLO et la sortie obtenue

3.3.2 Application pratique

Comme pour Faster-RCNN, nous développons et entrainons YOLO sur un notebook Google Colab. Nous utilisons les hyperparamètres suivants:

- batch-size *(nombre de photo passée avant de mettre à jour les poids du réseaux)* = 4
- epoch *(nombre de fois ou on passe l'entièreté du dataset dans le network)* = 120
- learning-rate *(Pas utilisé dans le calcul du gradient de la fonction de coût lors de la backpropagation)* = 0.001

Apres une heure d'entrainement du network avec un GPU Tesla T4, on obtient les metriques suivantes:

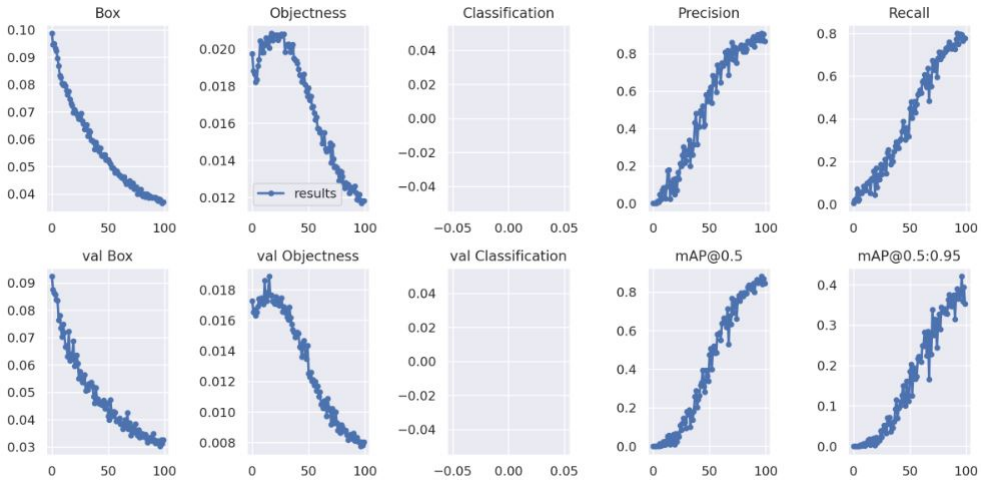

Figure 11: Metrics de notre modèle YOLO

D'abord, on observe que les metrics fournits par YOLO divergent quelque peu de ceux fournit par Faster-RCNN mais cela n'empechera par leur comparaison. Dans un

premier temps on voit qu'à la fois pour la période de validation et celle d'entrainement, la fonction de coût box (qui est un IoU) et Objectness (qui mesure l'erreur dans à quelle point la network se trompe dans la classification) décroit constamment montrant que le modèle apprend bien et que les fonctions de coût converge vers un minimum. Globalement, le modèle arrive à classifier.

Néanmoins pour savoir si il classifie bien comme on le souhaite, nous devons nous réferer à Recall, Precision et MAP. Par définition:

$$\text{precision} = \frac{\text{nombre de COTS correctement détectés}}{\text{nombre de COTS détectés}} \tag{1}$$

$$\text{recall} = \frac{\text{nombre de COTS correctement détectés}}{\text{nombre de COTS présents}} \tag{2}$$

On voit ici que les courbes recall et precision sont (en moyenne local) strictement croissante et converge reciproquement vers 0.8 et 0.9. On comprend donc que, d'une part, la majorité des COTS detectés sont des bien des COTS, et d'autre part, que quasi-totalité des COTS présents en image sont détectés. Néanmoins, le fait que precision > recall montre que le modèle à tendance à sur-classifier le background.

Enfin, le MAP (Mean Average Precision) permet d'évaluer à quel point le modèle réussit la tâche donnée (classifier des images) en se basant sur les valeurs de precision et recall. Ici sa croissance et sa valeur finale à 0.85 montre la pertinence de notre modèle et sa capacité à réaliser la tâche.

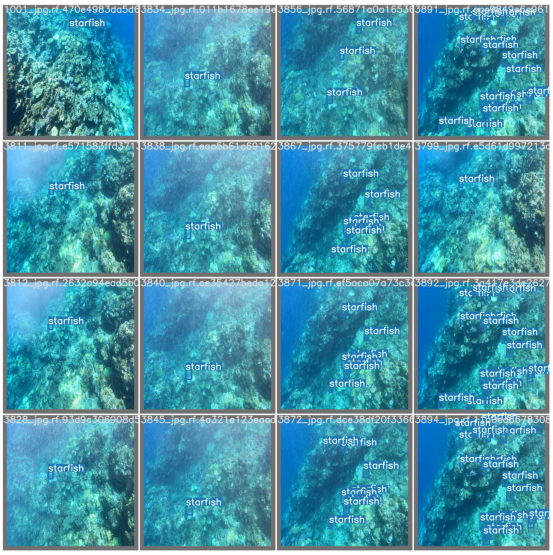

Figure 12: Exemple de sortie donnée par YOLO

Lors de l'inference sur la portion test de notre data-set, on note un temps moyen de detection de 0.012 s par image.

3.4 Comparaison des modèles

Maintenant les deux modèles traités, entrainés et évalués, nous venons à nous demander lequel est le plus pertinent pour notre cahier des charges.

Dans un premier temps, YOLO réalise la détection en 0.01s, s'entraine en 1h et possède un MAP finale de 0.85. Faster-RCNN lui réalise la détection en 0.2s, s'entraine en 2h et possède un MAP finale de 0.74 (le calcul se fait à partir des metrics de FRCNN à disposition). Malgrés que les deux modèles répondent au cahier des charges, YOLO sort du lot part sa rapidité et sa meilleur performance. Il est donc le modèle choisit pour notre logiciel.

3.5 Développement du Front-end

Pour ce livrable 2, nous ne nous sommes pas penchés sur le front-end (la page web) et l'implementation de l'IA avec celui-ci puisque cela correspond à l'objectif du livrable 3.

4 Organisation

4.1 Plan de charge prévisionnel

Description de l'activité	Charge en %	Charge en H	Maxence	Victoire	Thomas	Iwan
Total	100.00 %	200	50	50	50	50
Gestion de projets	25.00 %	50	10	10	12	18
Réunion de lancement	5.00 %	11	2	3	2	4
Planning prévisionnel et suivi d'activités	4.00 %	9	2	2	3	2
Réunions de suivi	5.00 %	10	3	3	2	2
Rédaction	5.00 %	9	1	1	2	5
Outils collaboratifs	6.00 %	11	2	1	3	5
Spécification	5.00 %	10	3	3	3	1
Définition des fonctionnalités	5.00 %	10	3	3	3	1
Conception préliminaire	15.00 %	30	9	5	6	10
Définition d'un modèle de données	3.00 %	7	2	1	1	3
Choix de la base de données	3.00 %	6	2	1	1	2
Définition des fonctionnalités	4.00 %	8	3	1	2	2
Définition des modules	5.00 %	9	2	2	2	3
Conception détaillée	25.00 %	50	14	15	13	8
Définition des classes Définition des méthodes	6.00 %	13	5	2	4	2
Définition des tests unitaires	6.00 %	12	2	5	2	3
Auto-formation	9.00 %	18	6	6	5	1
Maquettage des interfaces	4.00 %	7	1	2	2	2
Codage	20.00 %	40	8	11	12	9
Codage des classes	7.00 %	14	5	5	2	2
Codage des méthodes	6.00 %	13	1	5	5	2
Codage des tests unitaires	7.00 %	13	2	1	5	5
Intégration	5.00 %	10	2	4	2	2
Intégration des modules	2.00 %	5	1	2	1	1
Tests d'intégration	3.00 %	5	1	2	1	1
Soutenance	5.00 %	10	4	2	2	2
Préparation de la soutenance	2.00 %	5	2	1	1	1
Soutenance	3.00 %	5	2	1	1	1