

HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU  
CANTON DE VAUD



GESTION ET VALORISATION DE PROJET DE MACHINE LEARNING

GML

---

**Crapauduc - 2022**

---

*Authors:*

Schaller JORIS  
D'Ancona OLIVIER  
Logan VICTORIA  
Akouumba ERICA LUDIVINE  
Wichoud NICOLAS

January 11, 2023

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Gestion du projet</b>	<b>3</b>
2.1	Organisation du projet . . . . .	3
2.2	Gestion du temps de travail . . . . .	3
2.3	Gestion des tâches et répartition . . . . .	4
<b>3</b>	<b>Outils</b>	<b>6</b>
3.1	Colabeler . . . . .	6
3.2	Conversion de format . . . . .	6
3.3	Comptage des labels . . . . .	6
3.4	Folder Shortener . . . . .	6
3.5	Fusion des dataframes . . . . .	6
<b>4</b>	<b>Data preparation</b>	<b>7</b>
4.1	Acquisition des données . . . . .	7
4.2	Stockage des Données . . . . .	7
4.3	Labellisation des données . . . . .	8
<b>5</b>	<b>Filtrage</b>	<b>10</b>
5.1	Analyse des données labellisées . . . . .	10
5.1.1	Analyse temporelle . . . . .	11
5.1.2	Analyse météorologique . . . . .	13
5.2	Détecteur de planches . . . . .	14
<b>6</b>	<b>Modèles</b>	<b>16</b>
6.1	Choix des modèles . . . . .	16
6.2	Les échecs . . . . .	16
6.3	Les réussites . . . . .	19
6.3.1	Detectron2 . . . . .	19
6.3.2	FASTER R-CNN . . . . .	19
6.3.3	RETINA net . . . . .	22
6.3.4	<i>Detection Transformer (DE-TR)</i> . . . . .	23
6.4	Modèles évalués . . . . .	24
<b>7</b>	<b>Analyses</b>	<b>25</b>
7.1	Contexte . . . . .	25
7.2	Lecture d'un benchmark COCO . . . . .	26
7.3	Tensorboard de faster RCNN . . . . .	27

7.4	Loss de RetinaNet . . . . .	28
7.5	Évaluations des deux modèles sélectionnés . . . . .	28
7.5.1	Modèle final choisi . . . . .	29
7.6	Où est le problème . . . . .	30
7.6.1	Analyse des images non détectés par Faster-RCNN . . . . .	30
7.6.2	Analyse des images non détectés par RetinaNet . . . . .	32
<b>8</b>	<b>Prochaines étapes</b>	<b>33</b>
8.1	Comptage . . . . .	33
<b>9</b>	<b>Conclusion</b>	<b>34</b>

# Chapter 1

## Introduction

Ce projet de machine learning nous a été proposé dans le cadre du cours de Gestion et valorisation de projet en Machine Learning (GML), donné au cours du cinquième semestre du cursus de Bachelor en informatique et système de communication orienté ingénierie des données à la HEIG-VD. Le but de ce travail est de nous faire découvrir la gestion et organisation impliquée par un travail de machine learning, autant au niveau de la recherche technologique qu'au niveau de l'organisation d'équipe, notamment au niveau de la distribution de tâches, gestion d'équipe et de délais.

Le projet décrit ici est un projet existant ayant déjà été réalisé plusieurs fois par le professeur et de précédents étudiant.e.s : l'étude de crapauducs. Un crapauduc est un “petit conduit sous une route, permettant le passage protégé des batraciens” - selon Le Robert. En 2017, dix-huit crapauducs ont été construits le long de la route d'Aubonne à Gimel - canton de Vaud - afin de permettre aux grenouilles, crapauds et tritons de traverser la route des bois à l'étang en toute sécurité. L'image 1.1 montre l'un de ceux qui ont été installés.

À l'intérieur de ces crapauducs ont été installée une caméra équipée d'un capteur, qui implique la prise d'une petite série de photos (*Figure 1.2*) lors de la détection de mouvement, ainsi qu'une planche afin de faciliter la distinction des objets du sol. Comptant moins de caméras que de crapauducs, les caméras n'étaient pas rattachée à un crapauduc et comptent donc des images prises depuis différents crapauducs.



Figure 1.1: Un des 18 crapauducs installés pour l'étude

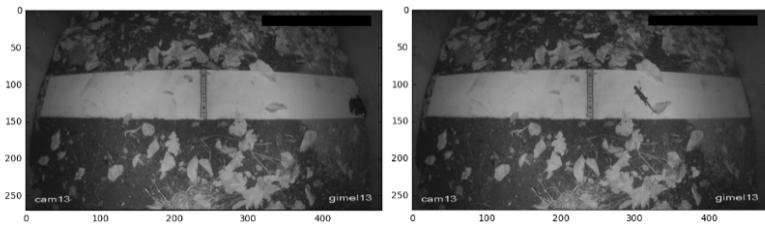


Figure 1.2: Exemples des images d'amphibiens prises par les cameras

Au terme de la première année d'utilisation, ces voies ont été empruntées par plus de 6'000 crapauds, grenouilles et tritons confondus. Ce comptage a été effectué par des chercheurs, qui ont du regarder les images prises par les caméras et compter les animaux "à la main". Le projet Crapauduc vise ainsi à utiliser l'apprentissage automatique (Machine Learning) pour automatiser le comptage des batraciens.

Notre objectif pour ce projet est donc de détecter la présence ou non de grenouille/crapaud et tritons (en considérant les grenouilles et crapauds comme une seule et même catégorie) en utilisant l'apprentissage automatique, ce afin de déterminer si la construction de ces crapauducs est efficace. Pour se faire, nous avons les prises des caméras du 23 février 2017 au 20 avril de la même année, totalisant près de 1 million d'images, dont on connaît pour chacune la caméra dont elle provient ainsi que le moment où la photo a été prise (date, heure, minute et seconde).

Enfin, le professeur Satizabal Mejia Hector Fabio nous a également mis à disposition sa labelisation pour certaines images, des bounding box pour certaines également, ainsi que les données météos enregistrées durant cette période (notamment la température, le vent, la précipitation et l'humidité).

# Chapter 2

## Gestion du projet

### 2.1 Organisation du projet

Dès le départ, nous avons décidé de travailler avec git, plus précisement en utilisant le site <https://github.com>. Nous avons donc créé une organisation afin de séparer les différents dépôts. Nous en avons définis 2, mais les membres de l'organisation étaient libres d'en ajouter d'autres.

- `crapauduc`. Ce dépôt est le dépôt principal où les notebooks des modèles sont déposés, nous y avons aussi placé les rapports des anciens étudiants afin d'y avoir un accès rapide. Nous y avons aussi déposé un subset d'image d'environ 0.5 Gib permettant le fine tuning.
- `utils`. Ce dépôt contient des scripts faisant des transformormations ou des analyses sur les données. Nous y avons par exemple un script qui permet de convertir les annotations de csv à COCO.

De plus, nous avons créer un compte google ayant le doux nom de `student GML` afin d'avoir un espace google drive de 15 GiB pour stocker les données ainsi qu'une intégration facilitée dans le service `colab.research.google.com` de Google. Nous croyions être prêts.

### 2.2 Gestion du temps de travail

Dès le départ, nous avons décidé de travailler à distance afin de dédier la totalité de la journée à ce projet sans perdre de temps dans les transports publics. En effet, le mardi où tombe le cours de GML, nous n'avons pas d'autre cours que ce dernier. Ainsi, un mardi typique se déroule comme suit:

- 8h00 - 13h15: Libre, mais souvent on prépare la séance de l'après-midi.
- 13h15 - 15h: Appel Teams, où nous expliquons notre avancement, normalement les différents problèmes rencontrés durant la semaine doivent être réglé avant la réunion. Planification des tâches pour la prochaine semaine, et répartition des tâches. Durant chaque réunion un membre du groupe prend des notes afin d'avoir un historique des discussions, ce procès verbale des réunions est stocké sur le google drive de `student GML`.

La séance du mardi se résume donc essentiellement à un partage d'information entre les différents groupes de travail composé de 1 à 3 étudiant.e.s. Le travail proprement dit est pour la plupart effectué en dehors des réunions, soit le mardi après la réunion soit à un autre moment choisis par les membres du groupe.

## 2.3 Gestion des tâches et répartition

Nous avons poussé notre utilisation de github, en gérant nos tâches à l'aide de l'outil de gestion de projet kanban directement intégré dans github. Ainsi, nous pouvons savoir à n'importe quel moment quel membre de l'équipe travail sur quelle partie du projet. De plus, nous pouvons voir les tâches en cours, les tâches terminées, les tâches en attentes, etc.

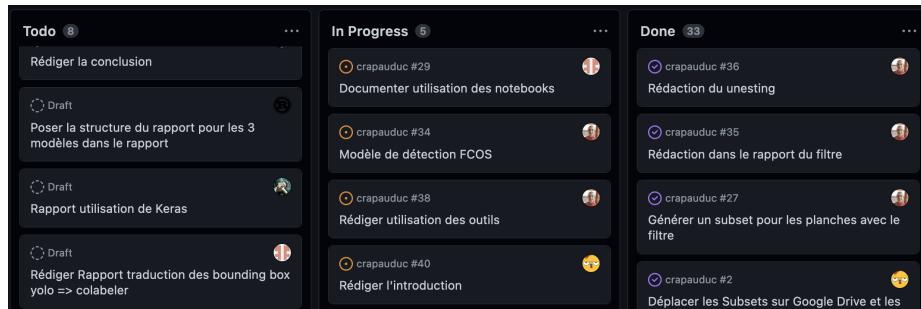


Figure 2.1: Nos tâches du Kanban réparties en 3 catégories : à faire, en cours et terminées.

**Problèmes d'organisation:** Nous avons décidé de ne pas élire de responsable au seins des étudiants. Cette décision a bien fonctionné pour certains aspects du travail, comme par exemple: la prise des notes durant les réunions. Cependant les appels Teams étaient généralement coordonés par Olivier et Joris sans vraiment que cela ait été prévu. Ce n'était pas voulu puisque nous souhaitions une organisation horizontale, mais cette manière de fonctionner s'est imposé naturellement et nous avons gardé cette organisation pour la suite. Par coordination nous entendons de manière générale animer la discussion et amorcer les points suivants. L'organisation de travail à elle aussi subit des changements au cours du projet. Au début, nous avons travaillé de manière très individuelle sur les petites tâches initiales. Nous souhaitions pouvoir travailler de manière très parallèle et ce choix semblait être bon. Néanmoins, cela nous a fait par la suite rencontrer deux nouveaux problèmes:

1. Friction lors de la communication.
2. Tâches trop complexes pour une seule personne.

Notre organisation initiale fonctionnait bien au début du projet puisque nous avions beaucoup de petites tâches et nous avons bien avancé. Cependant, les tâches devenaient de plus en plus grosses, les réunions ont pris de plus en plus de temps. En effet, nous avons rencontré beaucoup de problèmes qui étaient difficiles à résoudre seul, nous en discutions donc durant les réunions, et celles-ci commençait à prendre trop de temps. Après quelques séances peu efficaces, nous avons réalisé qu'il serait plus judicieux de travail en petits groupes afin qu'une partie de la communication se fasse déjà entre les membres du sous-groupe et ainsi que l'on réduise les informations à partager lors des réunions. De plus travailler à plusieurs permettait de surmonter les problèmes rencontrés plus facilement.

Un point que nous remarquons après ce travail et le suivant : nous avons souvent débloquer des problèmes en les abandonnant puis en y revenant plus tard, ceci nous a permis d'aborder le problème une seconde fois avec de nouvelles connaissances qui nous ont fait avoir une deuxième approche différente de la première.

Avec notre organisation actuelle, c'est à dire une réunion par semaine, nous n'arrivions pas forcément bien à laisser quelques jours le problème pour y revenir à tête reposée. Nous pensons que faire des réunions moins régulièrement, toutes les deux semaines par exemple, permettrait d'avoir plus de temps et ainsi de retravailler plusieurs fois le même problème entre deux réunions. Cependant cette solution demande des membres du groupe une plus grande autonomie, néanmoins en alliant cette proposition avec les petits groupes de travail présentés précédemment nous pensons que ça peut donner de bons résultats.

Un autre problème qui nous a embêté sur tout le projet et spécialement la fin du projet est le manque de ressources. Nous avons commencé par travailler en local, sur nos propres machines. Cependant, nous sommes très vite passés à Atlas afin de pouvoir héberger l'entièreté des données (500GiB) proche des notebooks. Atlas est une machine fournie par les professeurs sur laquelle un serveur Jupyter tournait. Cependant, nous n'avions pas accès à git depuis cet ordinateur ce qui posait des problèmes pour garder le projet synchronisé entre nos ordinateurs, github et Atlas. De plus, une fois les différentes analyses préliminaires terminées, il n'était absolument pas suffisant pour faire un entraînement d'un modèle comme DETR (60+ millions de paramètres). Nous avons donc migré sur l'offre gratuite de Google Colab. Les entraînements nécessitants toujours plusieurs dizaines de minutes, nous avons décidé de payer Colab Pro afin d'avoir accès à des GPUs premiums et de pouvoir tester et fine tuner rapidement des modèles. Nous avons donc entraîné plusieurs modèles sur Colab Pro, et les avons évalués avec le benchmark COCO, ceci est expliqué plus en détail dans le chapitre 7. Le soucis est le suivant, en ayant déplacé le projet sur plusieurs infrastructures, nous avons eu un projet éparsillé où il était dur de retrouver la dernière version. Un autre point notable est le manque de crédit à la fin du projet ce qui nous a empêcher de faire certaines analyses. En effet, DETR est tellement gros qu'il ne peut pas être chargé en mémoire sur un GPU non premium. Ainsi certaines expériences que nous aurions voulu réaliser n'ont pas été faites par manque de moyens. Nous voulions par exemple réaliser une centaine de prédictions sur un nouveau set et faire une fonction qui permet de filtrer certaines prédictions.

En conclusion, nous sommes satisfait de l'organisation et du déroulement de ce projet, tous les membres du groupe ont travaillé sur des parties diverses et variées du projet. Tout le monde a ainsi pu expérimenter avec au moins un modèle de machine learning. De plus, la gestion s'est faite de manière naturelle et a permis de garder une bonne entente entre les différents étudiants même durant les moments où nous rencontrions des problèmes. Nous sommes particulièrement fière d'avoir su adapté notre organisation au cours du projet, afin de le mener à bien et ce malgré notre manque de connaissance évident dans le domaine.

# Chapter 3

## Outils

**But** Nous avons constitué un repository github contenant des scripts permettant de transformer les données brutes en données utilisables pour l'analyse. Ces scripts sont disponibles dans le repository utils sur github.

### 3.1 Colabeler

Afin de réaliser les bounding box et les labels, nous avons utilisé le logiciel Colabeler permettant d'annoter les images pour l'object detection. Ainsi nous pouvons ajouter des bounding box facilement et rapidement. Il a été utilisé dans le cadre de la création du filtre et dans la constitution du dataset de test.

### 3.2 Conversion de format

Nous avons écrit un petit script python permettant de convertir les labels en différents formats. Il existe plusieurs manières de définir les bounding box. Elles peuvent être définies comme un point d'ancrage et une taille plus une hauteur ou simplement être 2 points. De plus, il existe différentes nomenclatures pour stocker ces images telles que le format coco stocké dans un fichier .json qui est associé au dataset coco. Il se peut que les données soient encore stockées sous forme d'un csv ou d'un fichier manifest qui peut être utile pour des services comme amazon sagemaker.

### 3.3 Comptage des labels

Un petit script a été mis sur pied afin de compter les labels déjà effectués. Ce qui permet d'avoir une liste des images déjà traitées et de constituer un subset rapidement pour entraîner des algorithmes.

### 3.4 Folder Shortener

Ce script bash permet de simplifier le chemin d'accès aux images pour une question de clarté et d'entretien du projet.

### 3.5 Fusion des dataframes

# Chapter 4

## Data preparation

### 4.1 Acquisition des données

**Problème** Lors de ce projet, les données doivent être accessible à tous les membres et doivent être stockées de manière uniformisée pour faciliter le travail de groupe. Nous avons alors opté pour une structure regroupant les images par caméra et le nom de fichier correspondant est la date ISO standardisée de la date de la prise du fichier.

**Source** Nous avons récupéré un disque dur comprenant les 500GB dans le bureau de nos professeurs. La structure de fichier était partitionnée par caméra, année, jour, heure, minute. Cette structure était pratique pour naviguer dans les dossiers mais posait un problème pour extraire les informations car les métadonnées étaient stockées dans le path du fichier et non dans un fichier .csv externe. La nouvelle structure partitionnée par camera permet d'avoir toutes les images regroupées et ainsi d'avoir les métadonnées au même endroit. Nous avons ainsi écrit des scripts de transformations que l'on peut trouver dans le repository utils sur github.

**Format** Les images sont au format JPEG, toutes les images sont de la même taille, 1920x1080 pixels.

**Numéro de séquence** Une information qui n'était pas présente originellement était le numéro de séquence des images. Lorsque la caméra détectait un mouvement continu, la même action pouvait résulter sur plusieurs images différentes. Nous avons donc considéré une séquence valide si sur la même caméra, les images sont prise à la suite dans un interval de temps inférieur à 2 secondes. Ce numéro est ainsi ajouté aux métadonnées et permet de réaliser des analyses plus approfondies.

### 4.2 Stockage des Données

Afin de stocker les données, nous utilisons deux espaces de stockage différents. Premièrement, nous utilisons le serveur atlas mis à disposition pour stocker les images brutes. Deuxièmement, nous utilisons Google Drive pour stocker les subsets d'images traitées. De cette manière, nous avons une source de donnée fiable et pouvons ainsi tous travailler en parallèle avec les mêmes données uniformisées.

**Datalake** Les données désarborisées ainsi que les données originales sont stockées sur le serveur Atlas dans le dossier `/home/crapauduc/data/`. Ce dossier est accessible à tous les membres du groupe. Les images sont stockées dans des dossiers par caméra et le nom de fichier est la date ISO standardisée de la date de la prise du fichier.

**Subsets** Les subsets sont stockés dans le Google Drive et peuvent être utilisés pour tester et entraîner différents algorithmes

### 4.3 Labellisation des données

**Problème** Comme dans tout projet de machine learning, nous avons besoin de données labellisées manuellement au préalable que l'on peut fournir comme données d'entraînement à nos réseaux de neurones. Dans le cadre de ce projet, on peut distinguer deux grands types de données labellisées. Ces deux types de labellisation ont été effectués avec le même outil de labellisation polyvalent, à savoir Colabeller, et sont décrites plus précisément dans les deux prochains paragraphes.

**Classification** Même si l'objectif final du projet n'est pas de classifier les photos par animal mais plutôt de localiser les animaux sur les photos, nous avons décidé d'utiliser la classification pour une étape intermédiaire, à savoir le détecteur de planche qui permet de déterminer si une photo a une grande probabilité de contenir un animal. Un certain nombre de photos labellisées étaient fournies au début du projet, mais cette labellisation concerne uniquement les animaux et ne donne aucune information sur la présence ou non de la planche sur les images. Nous avons donc dû partir de zéro pour ce travail de labellisation. Heureusement, la labellisation pour une tâche de classification est plutôt rapide puisqu'il suffit d'indiquer pour chaque image si elle contient une planche ou non, ce qui revient en gros à appuyer sur un bouton à chaque fois que l'on voit une planche. Nous avons donc choisi d'analyser un échantillon relativement grand de 5554 images aléatoires issues du tunnel numéro 2. Malgré la rapidité de la labellisation, nous avons rencontré un problème qui réside dans le déséquilibre entre les deux classes planche et non-planche. En effet, l'immense majorité des images contiennent une planche visible et on ne peut donc pas fournir ces données telles quelles au réseau de neurones. Nous avons donc choisi de nous restreindre à un sous-ensemble de 600 images dont environ la moitié contiennent une planche, et il se trouve que cela fut largement suffisant comme on peut le constater au vu des bons résultats obtenus par le détecteur de planche présentés plus loin dans le rapport.

**Localisation** A l'inverse de la classification, la localisation des animaux sur les photos est l'objectif principal de ce projet. Malheureusement, ce type de labellisation prend beaucoup plus de temps que la labellisation pour une tâche de classification, en particulier une tâche de classification binaire comme pour le détecteur de planche. En effet, il est désormais nécessaire pour chaque image contenant un animal de dessiner une bounding box autour de l'animal en question et de spécifier à chaque fois de quel animal il s'agit. Par chance nous avions déjà à disposition pour cette tâche de localisation un certain nombre de données labellisées disponibles dans le fichier `path_and_bounding_box.csv`. Nous avons choisi de tout de même essayer de labelliser quelques centaines d'images supplémentaires afin d'être certains de ne pas manquer de données d'entraînement. Cependant, cette tâche s'est avérée extrêmement longue et fastidieuse sans apporter de réelle plus-value au projet et nous avons

donc finalement décidé d'abandonner et de nous limiter aux 2000 labels mis à disposition, ce qui est amplement suffisant pour entraîner un réseau de neurones standard.

# Chapter 5

## Filtrage

**Idée générale** Le but de ce chapitre est de décrire les différentes méthodes de filtrage investiguées, dans le but d'améliorer la qualité des données.

**Problème** Le dataset original est composé de 18 caméras regroupant environ 1 million d'images. Une bonne partie de ces images sont des faux positifs. Il est donc nécessaire de filtrer les images afin de ne garder que les images qui nous intéressent. Une première observation nous fait remarquer que les images uniquement constituées de feuilles n'ont jamais d'animaux. Ensuite, une deuxième lecture nous fait remarquer que les animaux se déplacent plus facilement par temps humide. Et finalement, nous constatons que les animaux sont nombreux certains jours. À partir de ces observations, nous avons élaboré 3 méthodes pour filtrer les images et ainsi augmenter notre probabilité de trouver des animaux pour constituer de nouveaux labels ou constituer un dataset de validation. Ces méthodes sont décrites dans les sections suivantes.

### 5.1 Analyse des données labellisées

Comme dit en introduction, notre professeur monsieur Satizabal Mejia Hector Fabio nous a fourni les bounding box pour certaines images. C'est sur le fichier "path\_and\_bounding\_box.csv" - que nous avons préalablement créé à partir de ces données - que nous avons effectué l'analyse exploratoire des données. Nous nous basons ainsi 2020 images dont :

- 224 observations de tritons ;
- 201 observations de grenouilles-crapauds.

Ces données s'étendent sur la période du 9 mars au 15 avril 2017. Il est intéressant de noter que les données contenant des tritons et/ou des grenouilles-crapauds s'étendent du 9 mars au 1er avril, c'est-à-dire que l'on n'a pas observé entre le 1er avril et le 15 avril. Nous avons donc observé la présence de tritons et/ou grenouille-crapaud au travers de ces données via des variables temporelles - heure et jour - et via des variables météorologiques - telles que humidité, température ou précipitation.

Aussi, il est important de noter ici que plusieurs images peuvent faire partie du passage du même objet ; l'ensemble de données compte en effet 425 images dont environ prises. C'est pourquoi nous resterons très généraux pour cette première analyse des données. Voici donc ce que l'on a observé sur les données contenant des tritons et/ou des grenouilles-crapauds :

### 5.1.1 Analyse temporelle

#### Date

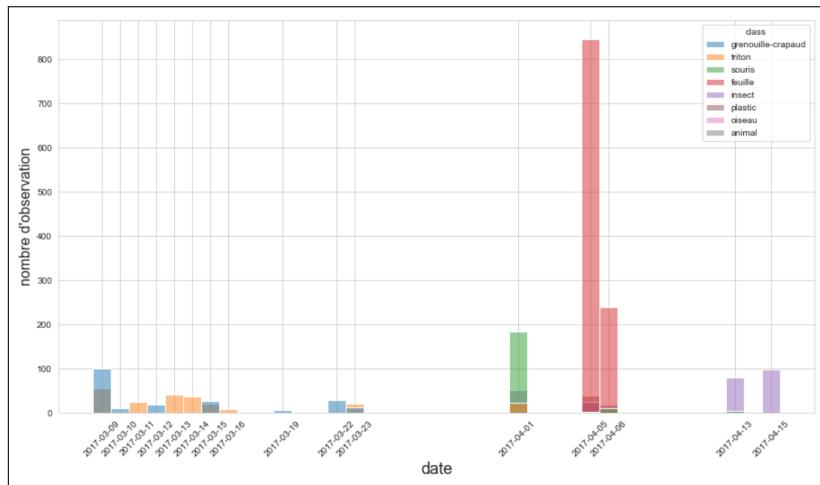


Figure 5.1: Fréquentation des objets en fonction de la date

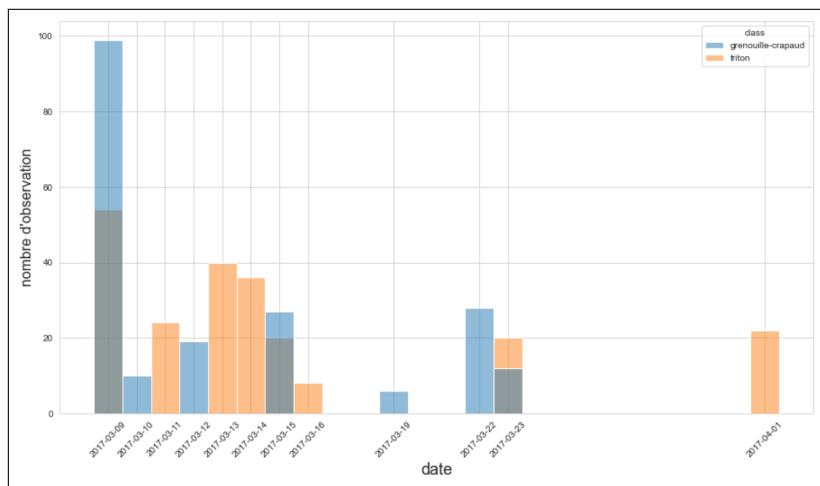


Figure 5.2: Fréquentation des batraciens observés en fonction de la date

On observe donc d'après la figure 5.2 que les batraciens d'intérêt utilisent particulièrement les crapauds en mois de mars. Le reste des observations durant cette période, nous indique cependant que l'on observe peu de données en avril.

Cependant, d'après cette ressource <http://www.karch.ch/karch/home/amphibien/osservazione-di-a.html> sur internet, les batraciens se reproduisent en fin février-début mars. On peut donc considérer que la déduction de fréquentation plus élevée des crapauds par les batraciens en mars peut être considérée pour un premier filtrage pertinent des images.

## Heure

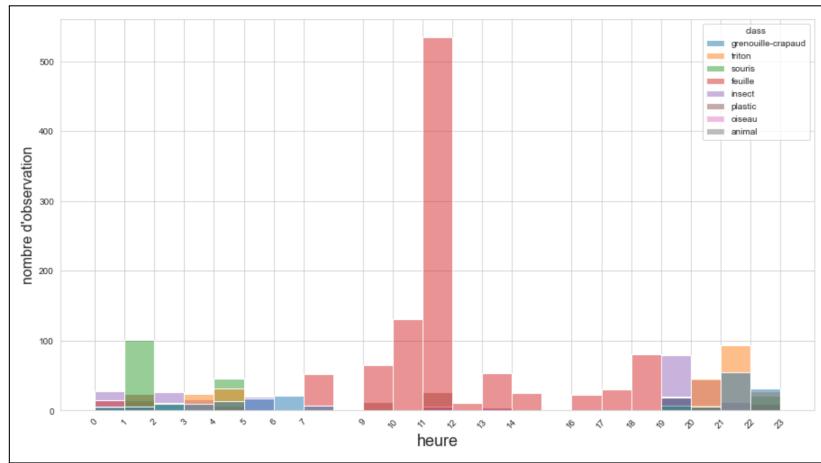


Figure 5.3: Fréquentation des objets en fonction de l'heure

On constate ici que le nombre de feuille étant plus grand que le reste d'objets observés, ceci nous empêche de pouvoir observer clairement la distribution d'observation d'objets. Visualisons donc les observations d'objets excepté les feuilles :

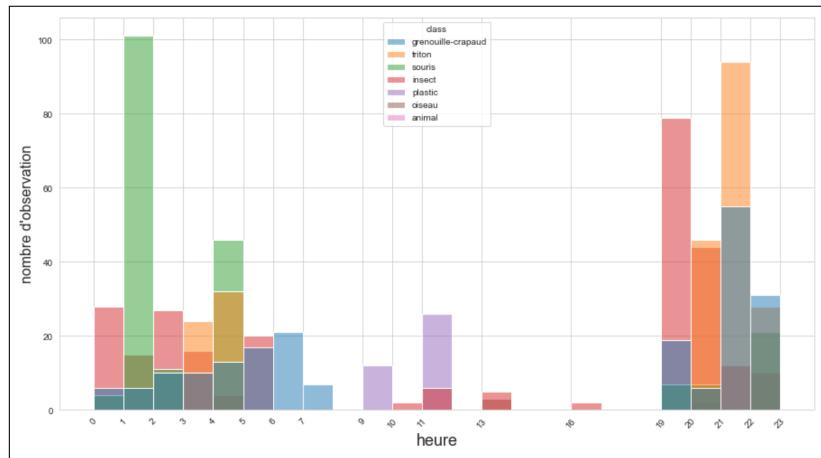


Figure 5.4: Fréquentation des batraciens observés en fonction de l'heure - sans les feuilles

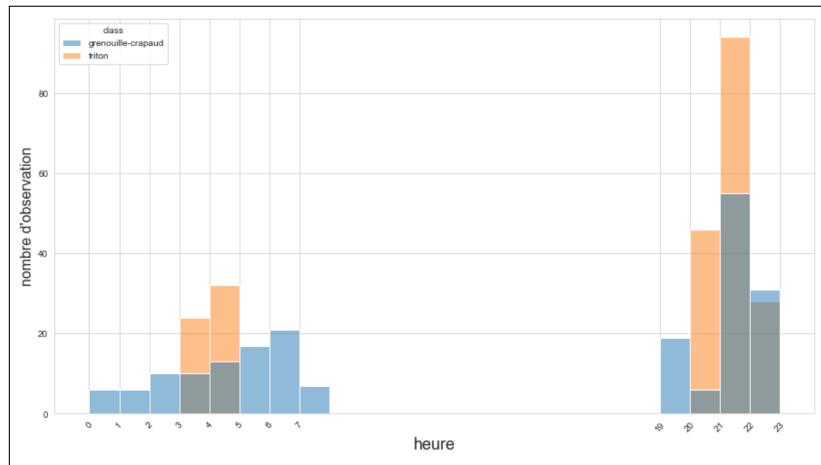


Figure 5.5: Fréquentation des batraciens observés en fonction de l'heure

D'après la figure 5.5 ci-dessus, on observe que les batraciens d'intérêt utilisent particulièrement - même uniquement, pour cet ensemble de données - les crapauducs entre 19h et 7h du matin, c'est-à-dire de nuit. Le reste des observations (figure 5.4) confirme premièrement la pertinence de cette observation, étant donné que nous avons une quantité élevée d'images prises tout au long de la journée parmi l'ensemble de données étudié ici.

Les quelques recherches faites sur la période de déplacement des batraciens à l'étang indiquant également qu'elle est particulièrement durant le crépuscule, on confirme ainsi la pertinence que peut avoir ce deuxième filtrage des images.

### 5.1.2 Analyse météorologique

Les données météorologiques additionnées des recherches en ligne ne sortent pas de particularité très prononcée quant à leur corrélation avec la fréquence d'observation de batraciens. Si l'on souhaite cependant citer les facteurs météorologiques qui pourraient être la plus déterministe, on citera l'humidité ; nous allons donc ici exposer nos observations la concernant. Notons que nous avons ici décidé de négliger les données labelisées "feuilles", comme elles forment du bruit et que nous avons fait un autre filtre s'en occupant si besoin.

## Humidité

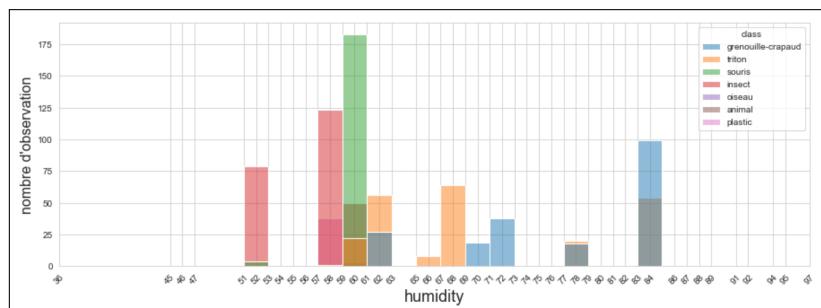


Figure 5.6: Fréquentation des objets en fonction de l'humidité - sans les feuilles

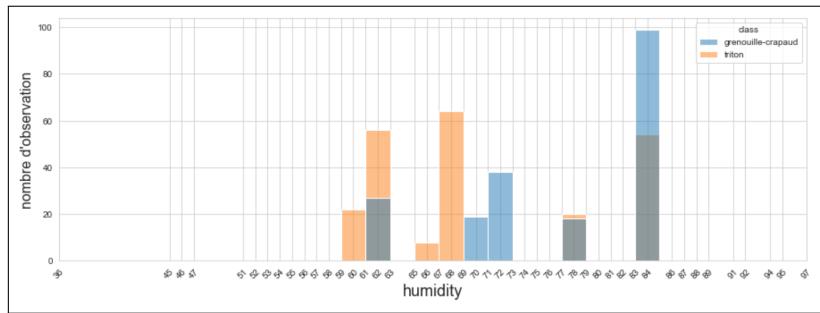


Figure 5.7: Fréquentation des batraciens en fonction de l'humidité - sans les feuilles

On voit ici qu'on peut imaginer prendre seulement les images prises lorsque l'humidité est au-dessus de 55.

## 5.2 DéTECTEUR de planches

Nous avons développé un réseau de neurones convolutif à l'aide de la librairie PyTorch. Ce classificateur binaire, prédit ou non la présence de planche.

**Dataset d'entraînement** Nous avons extrait 600 images d'une même caméra et labellisé 359 non planches et 241 planches. Ensuite, nous avons développé un dataloader permettant d'intégrer nos labels et de charger des batchs de données directement dans la librairie PyTorch. Celui ci, utilise un pipeline d'entrée qui applique plusieurs transformations à l'image avant de pouvoir l'utiliser comme un tenseur.



Figure 5.8: Exemple de données d'entraînement

**Architecture du DéTECTEUR** Le détecteur est simplement constitué de 3 couches convolutives suivies de 2 couches entièrement connectées. Les channels d'entrée et de sortie des couches convolutives sont de : 3 - 32, 32 - 64, 64 - 128. Le nombre de neurones des couches fully connected sont de 128 et 1 pour le neurone de sortie. La fonction de coût utilisé est la BCELoss et l'optimiseur est Adam. Le réseau est entraîné pendant 10 epochs avec un learning rate de 0.001 et un momentum de 0.9 sur 3 epochs.

**Résultats** Le détecteur de planche a une précision de 1 et un recall de 0.98 sur la détection de planche. En revanche, la précision sur la détection de non planche est de 0.88 et un recall de 1. Ce qui veut dire que notre filtre est un peu trop efficace et a tendance à se tromper pour détecter les images sans planche. Comme les résultats sont satisfaisant pour dégrossir le travail, nous n'avons pas passé de temps supplémentaire à optimiser le réseau afin qu'il sépare mieux les images dotés d'une planche ou non. Comme, nous traitons une grande quantité de données, l'erreur est acceptable. Lancé sur la quasi intégralité du dataset,

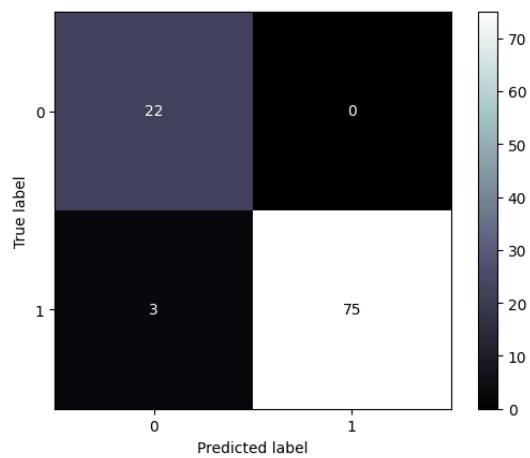


Figure 5.9: Matrice de confusion du détecteur de planche

le filtre a tourné pendant plus de 10h sur un ordinateur de bureau doté d'un processeur Ryzen9500X. Au final, le filtre a détecté 48910 images de non planches sur les 754543 images analysées.

# Chapter 6

## Modèles

### 6.1 Choix des modèles

Le choix des modèles a été un choix rapide.

Nous avons commencé par regarder les tutoriels sur les sites web des frameworks que nous utilisons. Nous avons donc regardé les tutotuels de pytorch, car nous voulions utiliser ce framework en particulier, mais avons aussi regardé les githubs de modèles dont nous avions entendu parler, comme YOLOv5. Il faut noter que notre compréhension du problème et du jargon utilisé dans le domaine s'est enrichi au fur et à mesure de nos recherches.

Ainsi, nos premiers choix peuvent sembler mauvais, mais lors de la prise de décision nous étions persuadés de faire les bons choix. Notre approche de départ se basait essentiellement sur un facteur: nous voulions des modèles pour lesquels il existe beaucoup de ressources en ligne. Cette méthodologie nous a amené à explorer une solution (YOLOv5) qui n'était pas adaptée à notre problème.

Après beaucoup d'essais infructueux, nous avons donc changé de méthodologie et nous nous sommes laissé la liberté d'utiliser des outils de plus haut niveau, tel que Detectron2 et de ne pas se restreindre uniquement à PyTorch. Une fois que nous avons pris en main ce framework, nous avons pu nous concentrer sur la performance du modèle. C'est aussi à ce stade que nous avons compris que le score sur le benchmark COCO2017 indiqué sur beaucoup de documentation de modèles était justement indiqué pour pouvoir comparer les modèles entre eux.

### 6.2 Les échecs

#### YOLOv5

YOLO est l'acronyme de You Only Look Once; il s'agit du premier modèle que nous avons essayé. Nous avons décidé de commencer avec ce modèle pour plusieurs raisons parmi lesquelles : une abondance de tutoriels sur le net et une solution qui nous semblait clé en main pour résoudre notre problème. Malgré ces signes positifs, il n'a pas été une solution adaptée.

En effet, YOLO, a été publié il y a plusieurs année et n'est plus forcément l'état de l'art actuel. De plus, ce modèle est adapté à du traitement en temps réel ce qui ne fait pas partie de notre problème. Cependant, le réel souci qui nous a fait abandonner cette solution est la structure spéciale du dataset qui ne correspondait pas à notre structure.

Durant une phase de réflexion visant à résoudre le souci de structure, nous avons ainsi réalisé

l'incapacité du modèle à gérer des images n'étant pas de 640x640 pixels. Nous aurions pu effectuer un réajustement de la taille des images mais ces derniers éléments nous ont fait réaliser que YOLO n'était pas la solution clé à laquelle nous nous attendions et avons décidé après quelques discussions de passer à un modèle plus adapté à notre problème initial. C'est ainsi que nous nous sommes lancé sur faster R-CNN, un modèle qui supporte des images de tailles arbitraires et qui est plus récent que YOLOv5.

## SSD

L'object detection étant une application nouvelle pour nous quand nous débutions le projet, après l'échec du modèle YOLO, nous avons décidé de nous lancer en parallèle sur différents modèles, le but étant de trouver celui ou ceux pouvant répondre efficacement à notre problématique. C'est dans cette optique que nous avons exploré le modèle SSD (Single Shot Multibox Détector). C'est un algorithme de detection d'objet dans une image qui au moment de la prédiction, divise l'image à l'aide d'une grille et génère des scores pour la présence de chaque catégorie d'objet dans chaque grille par défaut puis ajuste la grille pour mieux correspondre à la forme de l'objet. Le réseau combine ainsi les prédictions de plusieurs cartes de caractéristiques avec différentes résolutions pour traiter naturellement des objets de tailles diverses. Ce réseau se veut d'après la documentation, plus rapide que YOLO et aussi précis que FasterRCNN.

Nous avons trouvé un exemple d'implémentation sur [https://pytorch.org/hub/nvidia\\_deeplearningexamples\\_ssd](https://pytorch.org/hub/nvidia_deeplearningexamples_ssd). La mise en oeuvre de celui ci s'est faite sans trop de douleur. Par la suite, il était question de faire du transfert learning ou fine tuning selon les documents. En effet, le modèle a été entrainé sur le dataset COCO (MS COCO: Microsoft Common Objects in Context) qui est un jeu de données d'images à grande échelle contenant 328 000 images d'objets quotidiens et d'êtres humains. Nous souhaitons donc ré-entrainer une partie du réseau avec notre set d'images. Pour ce faire nous nous sommes aidé d'un tutoriel trouvé sur git à l'adresse <https://github.com/Coldmooon/SSD-on-Custom-Dataset>. Dans celle ci est expliqué une façon d'entrainer le modèle SSD sur un dataset personnalisé avec pour contrainte que les images doivent être de taille 300\*300 ou 512\*512. Travailler sur des images carrées était l'un des problèmes que nous avons rencontré avec YOLO, mais nous avons pensé résoudre ce problème avec les fonctions de resizing existantes. Nous avons commencé par essayer de reproduire le tuto sur le dataset proposé dans celui-ci. A de nombreuses reprises, nous avons fait face à des erreurs dont nous ignorions la provenance ainsi que la solution. Pour finir cela nous a pris beaucoup de temps pour au final ne pas arriver à entraîner le modèle sur le dataset en question. Nous n'avons donc pas pu aller au bout de cette implémentation. Mais nous restons convaincus que ça reste une alternative à la résolution de notre problématique.

## Faster R-CNN avec Keras

Après l'échec de YOLOv5 en raison de la nécessité d'utiliser des images carrées de taille fixe, nous avons décidé de nous pencher sur une utilisation potentielle d'un réseau de neurones de type Faster R-CNN, plus précisément en utilisant la bibliothèque open-source Keras puisque c'est une bibliothèque que nous avions déjà utilisée auparavant dans le cadre du cours sur les réseaux de neurones. Pour atteindre cet objectif, nous avions à disposition une implémentation toute faite de Faster R-CNN utilisant Keras disponible à l'adresse suiv-

ante : [github.com/you359/Keras-FasterRCNN](https://github.com/you359/Keras-FasterRCNN). Malheureusement, nous avons rencontré un certain nombre de problèmes au moment d'utiliser l'implémentation fournie.

Tout d'abord, il s'agit d'un code plutôt ancien qui n'est pas forcément compatible avec les dernières versions des librairies utilisées en Python. En effet, ces librairies sont régulièrement mises à jour et certaines fonctions disponibles sont alors dépréciées, modifiées voire même définitivement supprimées. Il a donc fallu trouver par tâtonnement les bonnes versions des librairies à utiliser en créant de multiples environnements virtuels à l'aide de conda et en interprétant les divers messages d'erreur énigmatiques renvoyés à chaque nouvelle tentative. Finalement, nous avons réalisé que le repo github contenait un fichier texte indiquant les versions optimales des librairies à utiliser pour ce projet. Cependant, même en utilisant les versions recommandées, le code continuait à planter après quelques secondes pour d'obscures raisons.

Ensuite, le second problème réside dans la documentation de l'implémentation de Faster R-CNN qui contient ce qui semble être une grossière erreur quant à la version de Python à utiliser. En effet, nous avons appris au point précédent qu'il valait mieux lire attentivement la documentation disponible avant de se lancer corps et âme dans le code. Or cette documentation indique explicitement d'utiliser Python 2 pour faire tourner le code mis à disposition. Malheureusement, même en utilisant les bonnes versions des librairies et de python le code ne voulait définitivement pas fonctionner. Dans une tentative désespérée nous avons donc changé la version de Python pour Python 3 et là, comme par magie, le code commence à tourner et le réseau de neurones commence à s'entraîner.

Finalement, nous arrivons au problème principal que nous n'avons jamais réussi à résoudre et qui est donc la raison pour laquelle nous avons abandonné ce modèle. En effet, même si le code parvenait désormais à se lancer correctement, il plantait maintenant à des étapes aléatoires de l'entraînement du réseau de neurones, parfois après quelques secondes, parfois après quelques dizaines de minutes, mais toujours en renvoyant une grande quantité de messages d'erreur pratiquement incompréhensibles. Malgré de longues et intenses recherches sur de multiples sites internet et forums, personne ne semblait en mesure de trouver une solution à ce problème. En effet, d'autres utilisateurs rencontraient le même souci mais la solution adoptée au final était toujours la même : changer de modèle, souvent pour passer sur detectron qui possède une documentation beaucoup plus complète, ce que nous avons donc également fait par la suite. Nous avons tout de même réussi à finir un entraînement sans encombre, mais pour y parvenir il a fallu réduire drastiquement le nombre d'epochs afin de limiter la durée de l'entraînement, et à la fin de celui-ci le réseau de neurones n'était pas capable de reconnaître quoi que ce soit sur les images, probablement en raison du manque d'entraînement.

Pour conclure, on peut donc dire que le cœur du problème de l'implémentation utilisée est son manque de popularité dans la communauté du data science. En effet, comme les utilisateurs sont peu nombreux, des erreurs se glissent dans la documentation et passent inaperçues tandis que d'autres problèmes restent à jamais non-résolus car personne ne semble connaître la solution. Nous avons donc choisi d'utiliser par la suite des modèles plus populaires et par conséquent mieux documentés.

## 6.3 Les réussites

### 6.3.1 Detectron2

Detectron2 est une puissante plateforme de détection d'objets et de segmentation d'images développée par Facebook AI Research (FAIR). Il s'agit de la deuxième génération du système Detectron original, également développé par FAIR. Detectron2 est conçu pour être hautement modulaire et flexible, et il est utilisé pour un large éventail de tâches de computer vision, comme la détection d'objets, la segmentation d'instances, la segmentation panoptique, et plus encore. Il est construit au-dessus du framework PyTorch et présente une variété de modèles de pointe. L'utiliser nous a permis de facilement changer de modèle une fois que notre code d'entraînement était fonctionnel. En effet, beaucoup de configuration se font à travers le fichier de configuration, c'est dans lui qu'on spécifie les détails du processus d'entraînements tels que le set de données d'entraînement, le set de validation, le modèle à entraîner, le learning-rate etc. De plus, il intègre de nombreuses configuration de bases et grâce à ceci nous n'avons qu'à changer les paramètres qui nous intéressent comme : la backbone du modèle, les poids à charger pour faire du transfert learning. Detectron2 intègre directement dans la class `DefaultTrainer` une méthode qui affiche des statistiques sur l'entraînement en cours, comme le nombre d'epochs, le learning-rate, le temps restant, etc. Cela nous a permis de suivre l'entraînement de nos modèles. Il s'occupe aussi de faire de l'augmentation de données durant le processus d'entraînement. Finalement, Detectron2 fournit aussi pour tous ses modèles des poids après entraînement du modèle.

Entrainer un modèle avec Detectron2 se résume donc à faire un code similaire à celui visible en ci-dessous.

```
1 from detectron2 import model_zoo
2 from detectron2.engine import DefaultPredictor
3 from detectron2.config import get_cfg
4 cfg = get_cfg()
5 # On charge une configuration de base
6 cf_file="COCO-Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml"
7 cfg.merge_from_file(model_zoo.get_config_file(cf_file))
8 cfg.DATASETS.TRAIN = ("triton_train",)
9 cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url(cf_file)
10 # cfg.params... voir dans la documentation
11 trainer = DefaultTrainer(cfg)
12 trainer.resume_or_load(resume=False)
13 trainer.train()
14 predictor = DefaultPredictor(cfg)
```

En conclusion Detectron2, nous avons permis de facilement changer de modèle et de tester plusieurs modèles. Nous avons aussi pu facilement faire du transfert learning en chargeant les poids d'un modèle pré-entraîné.

### 6.3.2 FASTER R-CNN

Le modèle d'apprentissage automatique Faster R-CNN (R-CNN pour "Region-based Convolutional Network") vient à la base du modèle R-CNN, qui a été ensuite amélioré pour former le modèle Fast R-CNN, qui a lui-même finalement été optimisé pour créer le modèle Faster R-CNN étudié ici. R-CNN, Fast R-CNN et Faster R-CNN sont tous des modèles de

traitement de l'image utilisés pour la détection d'objets dans les images. Ils utilisent tous une approche en plusieurs étapes qui consiste à extraire des caractéristiques de l'image à l'aide d'un réseau de neurones convolutionnel (CNN), puis à identifier les régions de l'image qui pourraient contenir des objets à l'aide de la sélection de région d'intérêt (ROI) et enfin à prédire la classe de chaque région sélectionnée et à localiser l'objet dans l'image.

**R-CNN** est un modèle datant de 2014, dont on peut voir l'article scientifique *ici*. Voici premièrement son architecture en figure 6.1 ci-dessous.

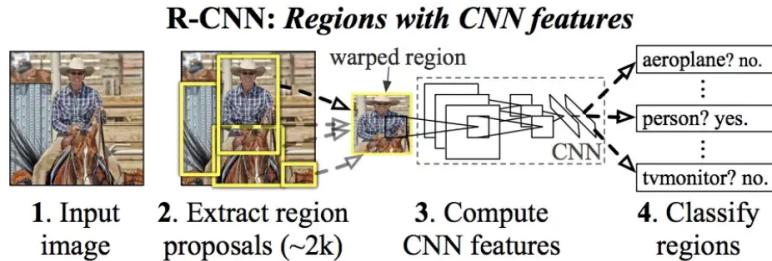


Figure 6.1: Architecture de R-CNN

Comme dit précédemment, il s'agit du modèle original. Il est très performant, mais aussi très lent, car il traite chaque région sélectionnée de manière indépendante et entraîne un modèle de classification séparé pour chaque région.

**Fast R-CNN** date d'une année plus tard, soit de 2015. On trouve son article scientifique *ici* et voici son architecture en figure 6.2 ci-dessous.

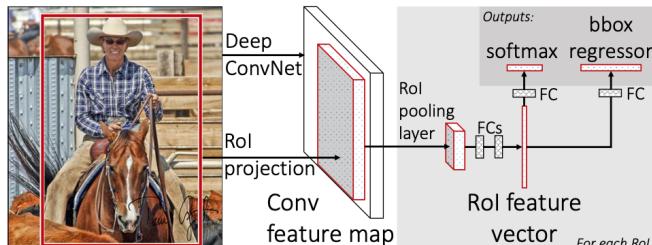


Figure 6.2: Architecture de Fast R-CNN

Fast R-CNN est ainsi une version améliorée de R-CNN, qui est beaucoup plus rapide. Au lieu de traiter chaque région sélectionnée de manière indépendante, Fast R-CNN utilise un seul modèle de classification pour toutes les régions sélectionnées dans l'image. De plus, il utilise une technique appelée "max pooling régional" pour réduire la dimension des régions sélectionnées avant de les passer au modèle de classification.

**Faster R-CNN** vient une année plus tard, en 2016. *Ici* se trouve son article scientifique, et voici finalement son architecture ci-dessous, en figure 6.3.

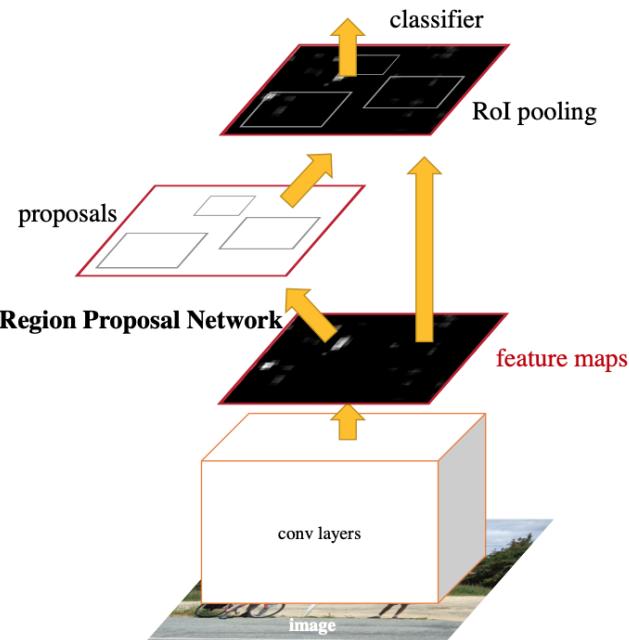


Figure 6.3: Architecture de Faster R-CNN

Faster R-CNN est effectivement un modèle encore plus rapide que Fast R-CNN. Il utilise une technique appelée "réseau de proposition de région" (RPN) pour identifier automatiquement les régions de l'image qui pourraient contenir des objets, sans avoir besoin de calculer explicitement toutes les régions de l'image comme le font R-CNN et Fast R-CNN. Cela permet à Faster R-CNN de traiter l'image de manière beaucoup plus rapide et de détecter des objets avec une précision comparable à celle de Fast R-CNN.

Nous avons décidé d'utiliser une backbone `X_101_32x8d_FPN` c'est à dire un réseau ResNeXt-101 avec une configuration de 32 groupes de large de 8, 32x8d fait donc référence à des hyper-paramètres utilisé par le réseau ResNeXt de la backbone. Nous n'avons pas nous même cherché ces valeurs, nous avons choisis cette configuration en copiant la configuration de Faster-RCNN donnant le meilleur score COCO selon la documentation de Detectron2. Ce réseau ResNeXt est suivi d'un second réseau, FPN cette fois, qui forment ensemble l'entièreté de la backbone que nous utilisons.

Pour des raisons évidentes de rapidité de traitement des images, nous avons donc investigué plus précisément ce dernier modèle - Faster R-CNN - pour notre problème. C'est d'ailleurs ce modèle que nous avons finalement sélectionné pour répondre à la problématique de ce projet, comme mentionné ci-après, en chapitre chapter 7 de ce rapport ("Evaluation"). Pour se faire, nous avons donc du adapter les exemples trouvés sur internet à notre problème, légèrement différent de ces dits exemples. Il a donc fallu assembler différents tutoriels, ce qui a impliqué beaucoup de temps d'analyse et de compréhension. Aussi, nous avons du nous habituer au jargon technique que nous n'avions pas - tel que mAP et les benchmarks coco pour en citer quelques uns. Il a également fallu passer du temps pour comprendre ce qu'était COCO (Common Object in COntext) : une base de données fournie par Microsoft qui contient des images annotées avec des informations sur les objets présents dans chaque image et qui fourni également un benchmark ; un ensemble de tests et de métriques utilisés

pour évaluer les performances des modèles de reconnaissance d'objets et de segmentation d'images.

### 6.3.3 RETINA net

#### Description

RetinaNet est un réseau de neurones convolutionnel utilisé en détection d'objets dans des images. Il a été présenté dans le papier "Focal Loss for Dense Object Detection" de Tsung-Yi Lin et al. en 2017.

Le principe de base de RetinaNet est de prédire des scores de confiance pour chaque classe d'objet à chaque position de l'image, ainsi qu'une boîte englobante pour chaque objet détecté. Pour cela, RetinaNet utilise une architecture de réseau de neurones à deux branches, l'une pour prédire les scores de confiance et l'autre pour prédire les boîtes englobantes.

La particularité de RetinaNet est qu'il utilise une "perte focale" pour lutter contre l'asymétrie des données dans les jeux de données de détection d'objets. Dans ces jeux de données, il y a souvent beaucoup plus de fond (c'est-à-dire des parties de l'image qui ne contiennent pas d'objets) que d'objets détectables. Cette asymétrie peut rendre difficile l'apprentissage pour le modèle, car il y a moins d'exemples d'objets à apprendre. La perte focale atténue cet effet en diminuant la contribution des exemples faciles (c'est-à-dire ceux où l'objet est facilement identifiable) à la perte totale, ce qui permet au modèle de se concentrer davantage sur les exemples difficiles.

Le modèle RetinaNet est souvent utilisé dans les tâches de détection d'objets pour améliorer la précision et réduire le nombre de faux positifs. Il a été largement utilisé dans de nombreuses applications, notamment la reconnaissance de la circulation routière et la reconnaissance de la faune.

Nous avons décidé d'utiliser une backbone R\_101\_FPN c'est à dire un réseau ResNet-101 suivit d'un réseau FPN. Il s'agit selon la documentation de Detectron2 de la backbone qui donne un modèle RetinaNet le plus précis, c'est à dire donnant le meilleur score sur le benchmark COCO. Nous avons donc choisi ce modèle pour avoir un modèle précis et qui puisse être utilisé pour répondre à notre problématique de base.

En résumé, RetinaNet est un modèle de détection d'objets qui utilise une architecture à deux branches et une perte focale pour améliorer la performance de détection dans les jeux de données asymétriques.

## Results

Ce modèle obtient des résultats extrêmement satisfaisants. En effet, il s'agit du modèle qui détecte les crapauds/grenouilles avec la plus grande accuracy. Les résultats précis de ce modèle sont par ailleurs détaillés dans la section "Evaluation" du chapter 7 de ce rapport.

### 6.3.4 Detection Transformer (DE-TR)

DETR est un modèle sorti en 2020 qui est extrêmement simple à implémenter et fournit des scores (mAP, IoU etc) sur le benchmark COCO qui surpassent ceux des modèles existants de quelques points. C'est pourquoi, en plus de son architecture innovante, nous avons voulu l'essayer. L'architecture, visible en figure 6.4, se base sur un transformer, un modèle de deep learning parut dans le fameux papier de 2017 de Google, *Attention is all you need*. On peut observer que le modèle commence par générer des features à partir de l'image d'entrée en utilisant une backbone, c'est-à-dire un modèle pré-entraîné visant justement à extraire ces features à l'aide d'un réseau neuronal convolutif. Ensuite, le modèle utilise la partie encodeuse du transformer suivi de son équivalent décodeur et finalement d'un "prediction feed-forward networks" (FFN). C'est le réseau FFN qui génère les possibles bounding boxes et les classes associées.

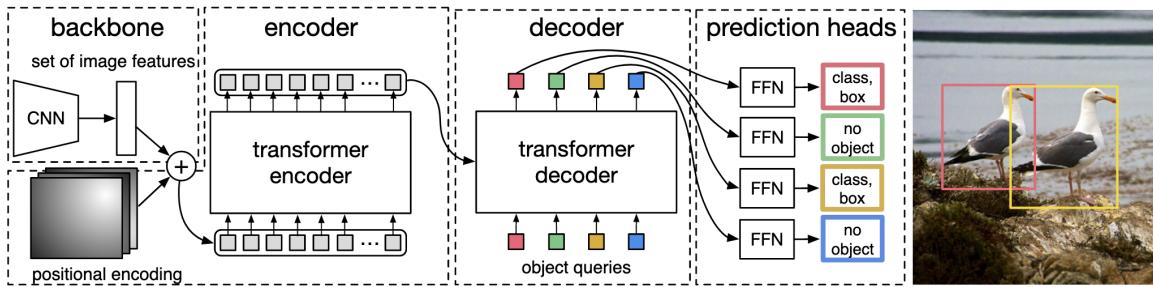


Figure 6.4: Architecture de DETR

Ce modèle est simple car il utilise peu de couches contrairement à FasterRCNN par exemple. Une implémentation PyTorch est faisable en 60 lignes. Cependant, l'entraînement est complexe et nous nous y sommes repris plusieurs fois pour réussir un entraînement. DETR n'est pas un modèle fourni dans les configurations de bases de Detectron2. Sur le dépôt github du projet DETR, du code permettant de l'intégrer avec Detectron2 est fourni, néanmoins ce wrapper est complexe à utiliser et nous n'avons pas réussi à obtenir des résultats satisfaisants, notre hypothèse est qu'il n'y a pas de threshold sur les probabilités de classes. Ainsi nous avons des prédictions similaires à celle visible sur la figure 6.5 qui illustre le problème.

Nous pouvons observer que le triton au centre de l'image est correctement reconnu avec une probabilité de 63% mais qu'il y a trop de bounding boxes. Nous voulions implémenter un système de filtrage après le modèle. Cependant nous n'avons pas réussi car le modèle ayant beaucoup de poids, il nécessitait un GPU puissant pour être entraîné. Nous avons déjà dépensé beaucoup d'argent de notre poche sur le service google colaboratory et nous n'avions pas les moyens de payer à nouveau pour entraîner et expérimenter sur un GPU puissant. Nous avons donc abandonné l'idée de filtrer les bounding box. Vous pouvez néanmoins retrouver notre implémentation de DETR ici [https://colab.research.google.com/drive/1nJC4tI83L1\\_sLhfFUMZPk4bFN0Ui3Jfi?usp=sharing](https://colab.research.google.com/drive/1nJC4tI83L1_sLhfFUMZPk4bFN0Ui3Jfi?usp=sharing). Une version ayant tourné une fois sur un GPU premium est disponible sur github sous le même nom.

Nous avons tenté une seconde implémentation de DETR en codant en PyTorch le modèle puisque on peut le faire en 60 lignes. L'entraînement était optimisé avec PyTorch Lightning. Nous avons réussi l'entraînement cependant en n'utilisant pas Detectron2, nous n'avons cette fois pas réussi à obtenir des scores sur le benchmark COCO fourni dans la classe `COCO_Evaluator` de Detectron2. Nous avons testé sur quelques images et voyons des bons



Figure 6.5: Prédiction de DETR après un entraînement sur un GPU premium

résultats, de plus durant l'entraînement de bonnes métriques sont affichées cependant sans benchmark COCO, il est difficile de comparer ce modèle aux autres. Vous pouvez retrouver notre implémentation ici: [https://github.com/student-GML/crapauduc/blob/main/model\\_finaux/DETR\\_kaggle\\_fine\\_tune.ipynb](https://github.com/student-GML/crapauduc/blob/main/model_finaux/DETR_kaggle_fine_tune.ipynb)

En résumé : Nous n'avons pas réussi par manque de ressources à filtrer les probabilités faibles avec le wrapper Detectron2 de DETR. Nous avons cependant, avec PyTorch Lightning, réussi à avoir de bons scores durant l'entraînement. Ces résultats ne sont pas suffisants pour être comparés aux autres modèles. Et nous avons donc décidé d'en rester là avec ce modèle.

## 6.4 Modèles évalués

D'après ces nombreuses recherches, nous avons donc implémenté et évalué deux modèles fonctionnels dont nous avons pu comparer les scores d'évaluation sur des données tests :

- Faster R-CNN avec Detectron2 ;
  - Retinanet avec Detectron2.

# Chapter 7

## Analyses

Nous allons commencer par expliquer les différentes métriques utilisables lors d'une tâche d'object detection. Ensuite, nous allons expliquer comment lire un benchmark COCO. Enfin nous allons analyser l'entraînement de nos différents modèles. Et pour finir, nous allons observer et analyser des images qui présentent des erreurs de prédictions.

### 7.1 Contexte

Dans une tâche d'object detection, nous utilisons le score IoU qui signifie Intersection over Union. C'est un score qui compare les bounding boxes prédites par le modèle avec les bounding boxes réels (ground-truth). Une tâche d'object detection comprend deux sous-problèmes: la classification et la localisation. Ainsi nous avons plusieurs métriques pour analyser la performance d'un modèle. IoU se concentre sur la localisation des bounding box prédites tandis que la métrique mAP (mean Average Precision) se concentre sur la classification. Un score IoU de 1 signifie que la bounding box prédite est parfaitement

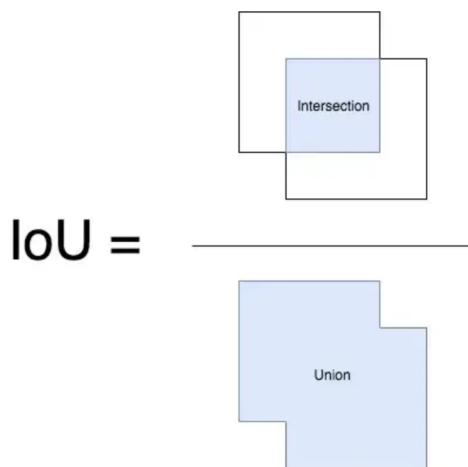


Figure 7.1: Intersection over Union

superposée sur la bounding box réel, tandis qu'un score de 0 signifie qu'il n'y a pas d'air en commun. Idéalement on espère donc avoir un score IoU de 1 pour toutes nos bounding boxes

## 7.2 Lecture d'un benchmark COCO

Un benchmark COCO peut être affiché dans la console comme présenté dans l'image 7.2. Ce benchmark est réalisé en suivant l'API COCO<sup>1</sup>. Il présente deux métrics la précision ( $\frac{TP}{TP + FP}$ ) et le rappel ( $\frac{TP}{TP + FN}$ ). La précision moyenne, (AP-Average Precision)

```
Accumulating evaluation results...
DONE (t=0.04s).
IoU metric: bbox
Average Precision (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.321
Average Precision (AP) @[ IoU=0.50   | area=   all | maxDets=100 ] = 0.842
Average Precision (AP) @[ IoU=0.75   | area=   all | maxDets=100 ] = 0.132
Average Precision (AP) @[ IoU=0.50:0.95 | area= small  | maxDets=100 ] = -1.000
Average Precision (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.067
Average Precision (AP) @[ IoU=0.50:0.95 | area= large  | maxDets=100 ] = 0.330
Average Recall    (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 1 ] = 0.399
Average Recall    (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=10 ] = 0.409
Average Recall    (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.409
Average Recall    (AR) @[ IoU=0.50:0.95 | area= small  | maxDets=100 ] = -1.000
Average Recall    (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.067
Average Recall    (AR) @[ IoU=0.50:0.95 | area= large  | maxDets=100 ] = 0.418
```

Figure 7.2: Exemple de benchmark COCO dans la console

présente dans les résultats COCO est calculée sur toutes les catégories, elle correspond traditionnellement au mean Average Precision (mAP). Les résultats sont divisé en 4 catégories qui dépendent du score IoU ou de l'air de la bounding box prédite.

- Les trois premières lignes sont l'AP en considérant différents seuil d'IoU pour sélectionner les boudings boxes à évaluer.
- Les trois suivantes sont l'AP en considérant des surfaces de tailles différentes pour sélectionner les boudings boxes à évaluer
- Les trois suivantes sont le Rappel Moyens (AR) en considérant plusieurs IoU pour sélectionner les boudings boxes à évaluer.
- Les trois suivantes sont l'AR en considérant des surfaces de tailles différentes pour sélectionner les boudings boxes à évaluer.

Selon la documentation, la première ligne est la plus importante. A la place de considérer les bounding boxes qui ont un IoU plus grand que le seuil, il est calculé sur cette ligne la moyenne des mAP des bounding boxes selon 10 seuils différents (de 0.5 à 0.95 par pas de 0.05). Il faut noter que s'il n'existe pas de bounding boxes répondant aux critères, un score de -1 est affiché. On observe donc qu'il n'existe pas de bounding box de petite taille (32x32 pixels) puisque l'AP ainsi que l'AR où l'air est petite vaut -1.

Avec Detectron2, un benchmark COCO est fait après chaque epoch. C'est à dire lorsque le modèle à vu une fois le dataset en entier. Ce benchmark est effectué sur une fold du set d'entraînement appelé validation. Cependant, afin de véritablement tester les résultats, nous avons aussi effectué un benchmark COCO sur un set de test d'images jamais vues

---

<sup>1</sup>[cocodataset.org/#detection-eval](http://cocodataset.org/#detection-eval)

category	AP	category	AP	category	AP
triton	33.819	grenouille-crapaud	44.724	souris	17.668

Figure 7.3: Exemple de résumé COCO

par le modèle. Detectron2 sauvegarde les résultats dans un fichier json, ce fichier peut être lu par un widget TensorBoard afin de montrer l'entraînement. Les résultats peuvent aussi être visualisés de manière résumées, par défaut ce n'est pas fait durant l'entraînement uniquement lors d'un benchmark COCO complet. Le résumé s'affiche comme dans l'image 7.3. On voit très facilement que les AP sont désormais calculés par classes sans distinction de IoU.

### 7.3 Tensorboard de faster RCNN

Les widgets tensorboard permettent durant tous les entraînements d'avoir un aperçu des différentes métriques enregistrées. Nous pouvons donc savoir quand arrêter l'entraînement pour avoir la meilleure performance selon une métrique, cela permet aussi de ne pas overfit. Nous avons pris quelques captures d'écrans affichées dans les figures 7.4a et 7.4b Detectron2

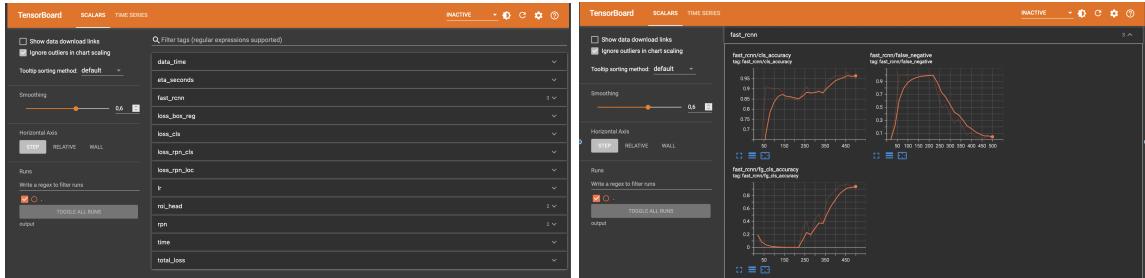


Figure 7.4: Aperçu de tensorboard et des métriques de faster RCNN

affiche aussi une loss total, nous n'avons pas réussi à déterminer la manière dont elle est calculée, mais elle reste un bon indicateur de la performance du modèle et de l'over/under fitting. De plus comme nous avons plusieurs modèles, nous pouvons les comparer entre eux puisque toutes erreurs seraient identiques sur tous les modèles. Comme nous pouvons le

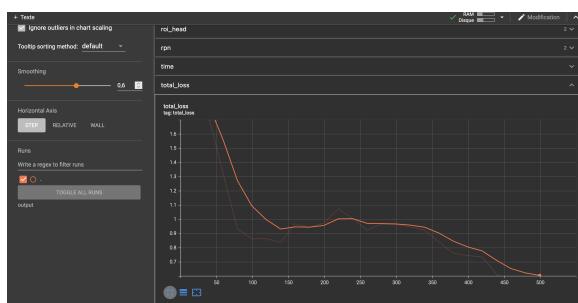


Figure 7.5: Aperçu de la loss total enregistrée dans TensorBoard

voir dans la figure 7.5, nous n'avons ni d'overfit, ni d'underfit.

## 7.4 Loss de RetinaNet

Dans cette section nous allons observer les métriques du modèle RetinaNet. Ce modèle enregistre moins de métriques, ceci est du au fait qu'il ne possède pas une architecture aussi complexe que Faster-RCNN. Alors que Faster-RCNN présentait des métriques pour certaines parties spécifiques du réseau, les RPN (Region Proposal Network) par exemple, RetinaNet ne présente que deux métriques relatives à l'entraînement : la loss de classification ainsi que la loss des bounding boxes. Nous pouvons observer que RetinaNet doit être plus

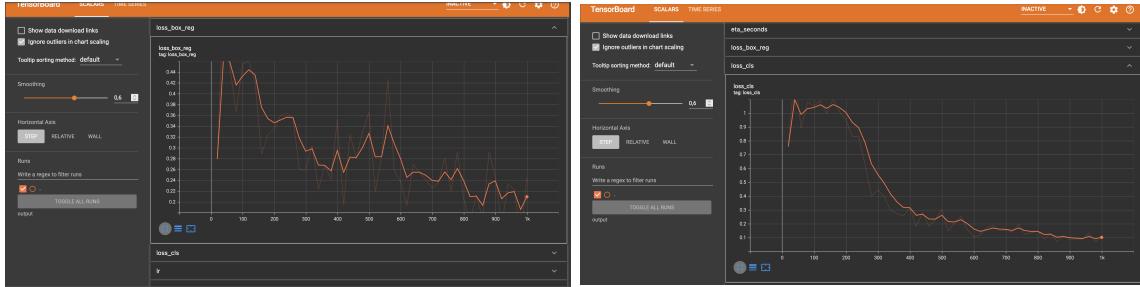


Figure 7.6: Les métriques de RetinaNet durant un entraînement

entraîné, 1000 itérations contre 500 pour Faster-RCNN. Il ne faut pas confondre itérations et epoch. Une epoch représente un passage sur l'entièreté du dataset, tandis qu'une iteration est un pas vers la recherche d'un minimum durant l'optimisation de l'erreur. Nous avons déterminé ces hyperparamètres grâce à tensorboard qui permet durant l'entraînement d'avoir un aperçu des métriques. Fait intéressant, RetinaNet est plus rapide à entraîner (environ 10 minutes de moins que Faster-RCNN) ceci est sûrement du à son architecture plus simple qui nécessite donc moins de calculs.

## 7.5 Évaluations des deux modèles sélectionnés

Après avoir testé plusieurs approches comme mentionnées dans le chapitre 6, nous avons retenu deux modèles dont nous allons ici comparer les scores d'évaluation :

- Faster R-CNN avec Detectron2 et
- Retinanet avec Detectron2.

Voici donc un tableau résumant les average precision (AP) scores de chaque catégorie suivant le threshold, ainsi

Scores par classe :

modèle	catégorie	AP score
Faster R-CNN	crapaud-grenouille	42.03
	triton	42.58
Retinanet	crapaud-grenouille	49.40
	triton	0.00

Bien que le score pour la classe crapaud-grenouille soit plus élevé à l'aide du modèle Retinanet, le score pour la classe triton est de zéro. Rappelons que la tâche initiale de ce

projet est de compter le nombre de triton/crapaud-grenouille qui utilisent les crapauduc ; ainsi, nous avons choisi l'algorithme Faster R-CNN comme modèle final pour cette tâche de classification.

### 7.5.1 Modèle final choisi

**Entraînement** Voici donc comment nous avons premièrement entraîné notre modèle Faster R-CNN, avec Detectron 2 :

```

1 | cfg = get_cfg()
2 | cfg.merge_from_file(model_zoo.get_config_file("COCO-Detection/
3 | faster_rcnn_X_101_32x8d_FPN_3x.yaml"))
4 | cfg.DATASETS.TRAIN = ("triton_train",)
5 | cfg.DATASETS.TEST = ()
6 | cfg.DATA_LOADER.NUM_WORKERS = 2
7 | cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-Detection/
8 | faster_rcnn_X_101_32x8d_FPN_3x.yaml")
9 | cfg.SOLVER.IMS_PER_BATCH = 2
10 | cfg.SOLVER.BASE_LR = 0.00020
11 | cfg.SOLVER.MAX_ITER = 500
12 | cfg.SOLVER.STEPS = []
13 | cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 128
14 | cfg.MODEL.ROI_HEADS.NUM_CLASSES = 4

```

**Test** Et voici donc le modèle construit pour tester nos données d'évaluation :

```

1 | # on recuperation des poids du modele qu'on vient d'entrainer
2 | cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model_final.pth")
3 | # determination d'un threshold pour le test
4 | cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.7
5 | # construction du predicteur
6 | predictor = DefaultPredictor(cfg)

```

Le threshold choisi pour le test ci-dessus indique donc qu'on considérera une image comme étant prédite d'une certaine classe si le modèle est sûr à au moins 70% de sa classification.

Voici deux exemples de résultats de classification par notre modèle ainsi créé :



Figure 7.7: Prédiction de Faster R-CNN - crapaud-grenouille



Figure 7.8: Prédiction de Faster R-CNN - triton

Validation (résultats) important de dire que images sont similaires à celles déjà vues blabla – Joris (précision sur ce qui est dit..)

## 7.6 Où est le problème

Nous avons donc des résultats qui approchent ceux de l'état de l'art en 2019 selon <https://paperswithcode.com/sota/object-detection-on-coco>. Nous avons aussi des modèles qui sont correctement entraînés puisque, nous l'avons vu dans la partie 7.3 ainsi que 7.4, les métriques indiquent qu'ils sont ni sous-entraînés ni sur-entraînés. Il nous faut donc observer des mauvaises classifications afin de comprendre les erreurs et de déterminer d'autres facteurs sur lesquels nous pouvons agir afin d'avoir de meilleurs résultats. En effet à l'heure où nous écrivons ces mots, un modèle basé sur DETR<sup>1</sup> atteint 64.5% d'AP sur le benchmark COCO, ce qui est un score très élevé. Il existe donc des modèles SOTA bien meilleurs mais nous n'arrivons pas à les reproduire. Nous allons donc observer les erreurs de classification afin de comprendre ce qui ne va pas, mais avant nous aimeraions faire remarquer qu'une explication possible est simplement la taille du Modèle. DETR, notre plus gros modèle, a 60 millions de paramètres alors que leur modèle en a 600 millions.

### 7.6.1 Analyse des images non détectées par Faster-RCNN

La plupart des images qui présentent des erreurs pouvant expliquer le score de de Faster-RCNN ressemble aux images de la figure 7.9. Nous avons ici trois types d'erreurs.

La première, qui est aussi une des plus fréquentes, est la non détection d'animaux (souvent sur les bords de l'image). Le manque de luminosité ou la présence de feuille pourrait expliquer cela. Un pré-processing pourrait vérifier l'hypothèse de la luminosité. Cependant, nous ne pouvons pas vraiment faire de pré-processing pour la présence de feuille.

La seconde est aussi assez fréquente, il s'agit de la présence de multiples bounding box sur un même animal. Même si c'est étonnant, cette erreur ne devrait pas affecter les scores sur le benchmark. Nous avons aussi tuner le threshold de confiance pour éviter ce genre d'erreur.

---

<sup>1</sup><https://arxiv.org/pdf/2211.03594v1.pdf>

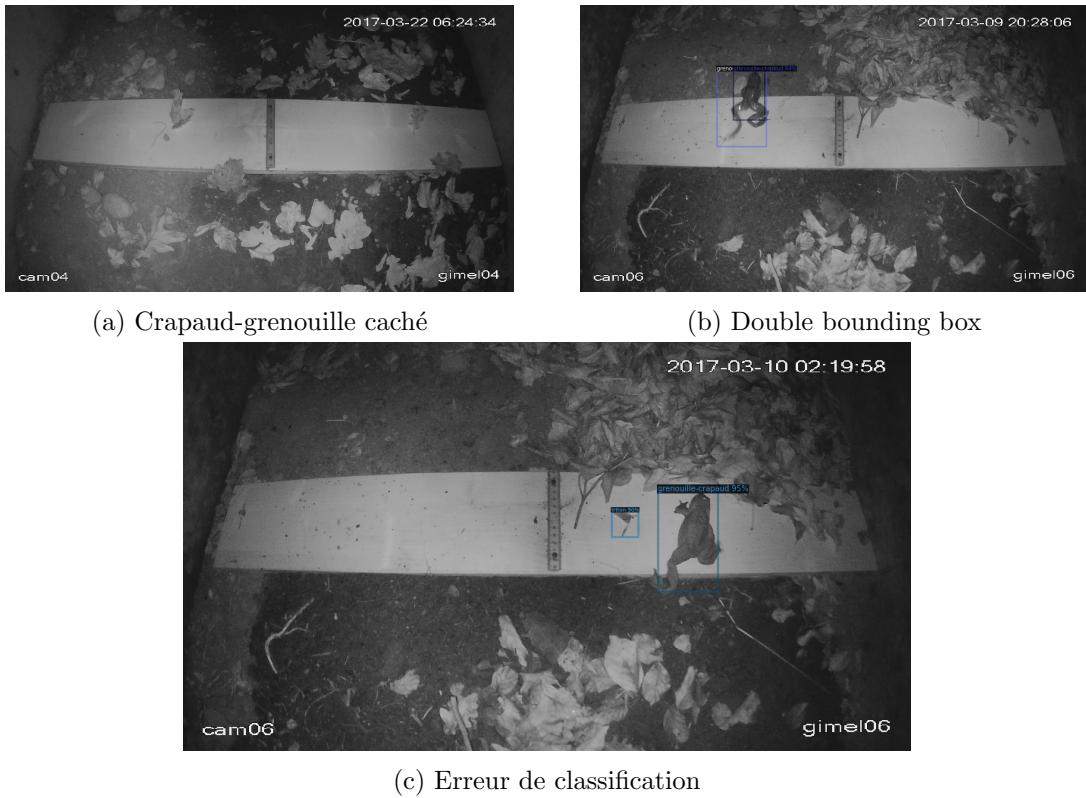


Figure 7.9: Exemple d'images mal prédites par Faster-RCNN

La dernière est la seul occurence de ce type observée sur le set de teste, on peut voir qu'un bout de feuille est mal classifié avec pourtant une probabilité de 90%. Et ceci sans raisons visibles. En conclusion, Faster-RCNN est vraiment bon et les erreurs ne présentent pas de grâve erreur d'entrainement.

### 7.6.2 Analyse des images non détectés par RetinaNet

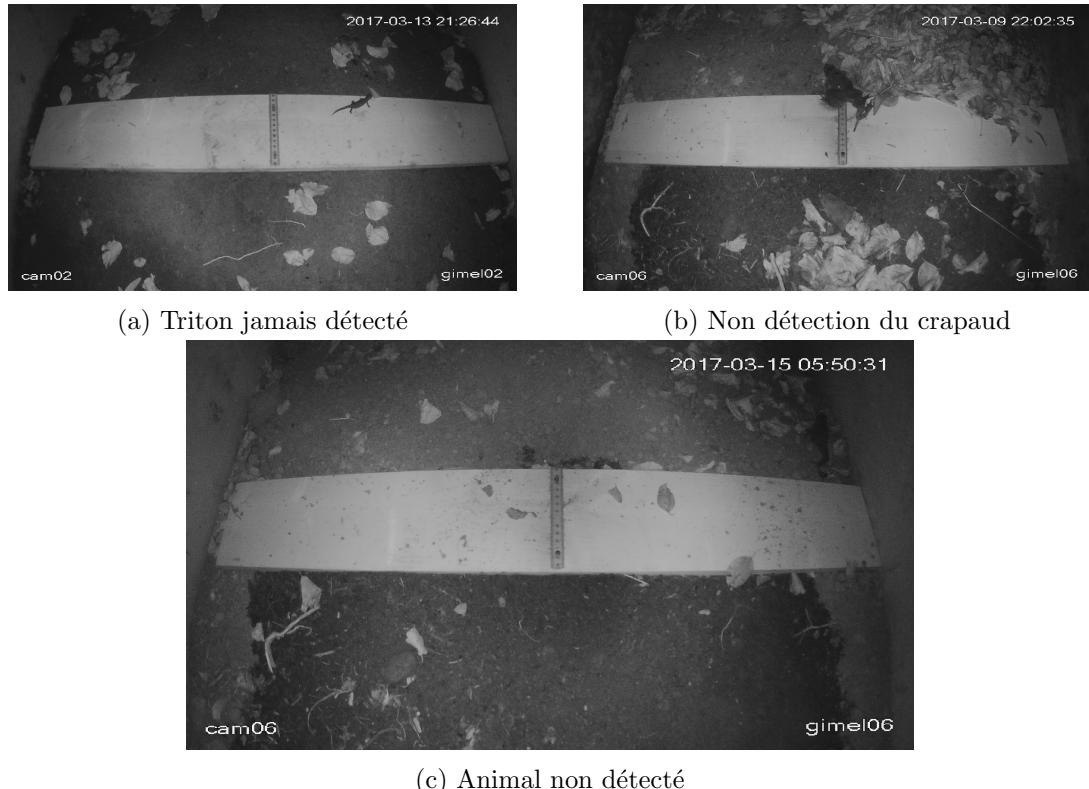


Figure 7.10: Exemple d'images mal prédites par RetinaNet

Nous pouvons voir dans la figure 7.10 que RetinaNet ne détecte jamais les tritons. Nous avons pensé que c'était du à la petite taille de ces derniers, cependant nous pouvons observé que l'image 7.10b est un gros crapaud qui n'est pas non plus reconnu. Il faut remarquer que du set de test, il s'agit de l'unique occurrence d'une telle erreur. On peut noter que les crapauds-grenouilles sont toujours très bien détecté. L'image 7.10c nous montre un problème similaire à celui vu précédemment sur l'image 7.9a ce qui tend à confirmer l'hypothèse du manque de lumière sur les cotés.

**Conclusion** Nos deux modèles semblent avoir des difficultés avec les animaux qui passent par les cotés, il se peut, au vu des nombreuses annotations d'entraînement bien centrées, que les modèles aillent apprendre à détecter les animaux au centre de l'image. Néanmoins, un ajustement de la luminosité serait une piste de réflexion.

# Chapter 8

## Prochaines étapes

### 8.1 Comptage

Le comptage des animaux traversant le tunnel est la tâche principale pour laquelle ce projet a été mis sur pieds. Maintenant que nous avons des résultats probants pour l'object detection, l'étape suivante consiste en la mise en oeuvre d'une stratégie permettant de compter le nombre de tritons et grenouilles-crapauds traversant le tunnel. Ceci est une tâche assez complexe dans la mesure où une fois les capteurs enclenchés, une longue série d'images est prise. De ce fait plusieurs images par exemple de tritons à différentes positions peuvent représenter le même triton. Pour pallier cette difficulté, nous avons pensé à regrouper les images par prise. Les prises ont été séparées entre elles par un intervalle de 2s afin de réduire au plus la probabilité de compter plusieurs fois le même animal.

Une fois les animaux détectés par notre modèle, nous allons procéder au comptage en regroupant les détection par prise. Nous n'allons donc conserver qu'un seul animal de chaque type qui aura été détecté par prise.

# **Chapter 9**

## **Conclusion**

Ceci est un acronyme k-nearest neighbor