

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
AKOUMBA Erica Ludivine
WICHOUD Nicolas

January 16, 2023

Contents

1	Introduction	1
2	Gestion du projet	3
2.1	Organisation du projet	3
2.2	Gestion du temps de travail	3
2.3	Gestion des tâches et répartition	4
3	Outils	7
3.1	Colabeler	7
3.2	Conversion de format	7
3.3	Comptage des labels	7
3.4	Folder Shortener	7
3.5	Fusion des dataframes	8
3.6	Tri des images	8
3.7	Réorganisation des données	8
4	Préparation des données	9
4.1	Acquisition des données	9
4.2	Stockage des données	9
4.3	Labellisation des données	10
5	Filtrage	12
5.1	Analyse des données labellisées	12
5.1.1	Analyse temporelle	13
5.1.2	Analyse météorologique	15
5.2	Détecteur de planches	16
6	Modèles	18
6.1	Les échecs	18
6.1.1	YOLOv5	18
6.1.2	Faster R-CNN avec Keras	19
6.1.3	SSD	20
6.2	Les réussites	21
6.2.1	Detectron2	21
6.2.2	FASTER R-CNN	21
6.2.3	RETINA net	24
6.2.4	<i>Detection Transformer (DE-TR)</i>	25
6.3	Modèles évalués	26

7 Analyses	27
7.1 Contexte	27
7.2 Lecture d'un benchmark COCO	28
7.3 L'entraînement	29
7.3.1 TensorBoard de faster RCNN	29
7.3.2 TensorBoard de RetinaNet	30
7.4 Évaluations des deux modèles sélectionnés	31
7.4.1 Scores des modèles	31
7.4.2 Modèle final choisi	33
7.5 Où est le problème	34
7.5.1 Analyse des images non détectés par Faster-RCNN	35
7.5.2 Analyse des images non détectées par RetinaNet	36
8 Prochaines étapes	38
8.1 Évaluation des modèles sur le set de test	38
8.2 Comptage	38
8.3 Comparaison avec le cahier des charges	39
8.3.1 Must have	39
8.3.2 Nice to have	39
8.4 Expériences futurs	40
9 Conclusion	42

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 des délais.

Le projet décrit ici est un projet existant ayant déjà été réalisé plusieurs fois par le professeur et de précédents étudiants : 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 des caméras équipées d'un capteur ont été installées. Après déclenchement, elles peuvent capturer une petite série de photos (*Figure 1.2*) lors de la détection de mouvement. De plus, une planche a été installée afin de faciliter la distinction des objets au sol. Comptant moins de caméras que de crapauducs, les caméras n'étaient pas rattachées à un crapauduc et comptent donc des images prises depuis différents crapauducs.



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

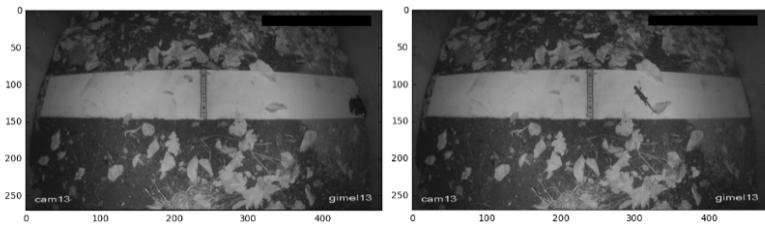


Figure 1.2: Exemples des images d'amphibiens prises par les cameras

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 inspecter 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 d'un 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).

Nous avons donc décidé d'aborder ce problème à l'aide de l'object detection en investiguant ainsi les différents modèles existants qui sont communément utilisés pour ce genre de tâche.

Chapter 2

Gestion du projet

2.1 Organisation du projet

Dès le départ, nous avons décidé de travailler avec git, plus précisément en utilisant le site <https://github.com>. Nous avons donc créé une organisation afin de séparer les différents dépôts. Nous en avons défini deux, 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 sous-ensemble d'images d'environ 0.5 Gib permettant le fine tuning.
- `utils`. Ce dépôt contient des scripts faisant des transformations ou des analyses sur les données. Il contient par exemple un script qui permet de convertir les annotations de csv à COCO.

De plus, nous avons créé un compte Google - ayant le doux nom de `student GML` - afin d'avoir un espace Google Drive de 15 GiB pour se partager aisément les données ainsi qu'une intégration facilitée dans le service `colab.research.google.com` de Google. Cependant, cette organisation n'a pas été complètement satisfaisante, comme détaillé dans les points suivants.

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, n'ayant pas d'autres cours que GML ce jour-là, un mardi typique se déroule comme suit :

- 8h00 - 13h15: Libre, préparation de la séance de l'après-midi la plupart du temps ;
- 13h15 - 15h: Appel Teams, où nous expliquons notre avancement. Les différents problèmes rencontrés durant la semaine doivent au possible être réglés 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 verbal des réunions est stocké sur le Google Drive de `student GML`.

La séance du mardi se résume donc essentiellement à un partage d'informations entre les différents groupes de travail composés de 1 à 3 étudiants. Le travail à proprement dit est - pour la plupart - effectué en dehors des réunions ; soit le mardi après la réunion, soit à un autre moment choisi 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 travaille sur quelle partie du projet. De plus, nous pouvons notamment voir les tâches en cours, les tâches terminées, ainsi que les tâches en attentes.

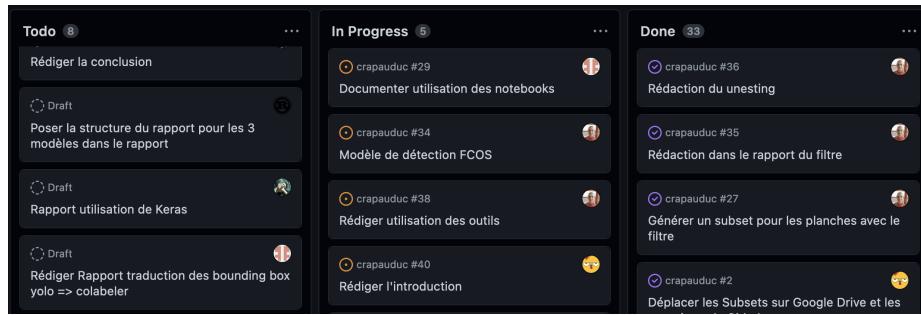


Figure 2.1: Nos tâches du Kanban réparties en 3 catégories : à faire, en cours et terminées.

Problèmes d'organisation: Nous avons décidé de ne pas élire de responsable au sein des étudiants et de garder une hiérarchie horizontale. Cette décision a bien fonctionné pour certains aspects du travail, tel que pour la prise des notes durant les réunions. Cependant, les appels Teams étaient généralement coordonnés par Olivier et Joris sans vraiment que cela ait été expressément prévu. Cette manière de fonctionner s'est imposée 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 a elle aussi subi 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, cette approche s'est avérée contenir plusieurs problèmes :

1. Échanges inefficaces ;
2. Tâches trop complexes pour une seule personne .

Notre organisation initiale fonctionnait bien au début du projet puisque nous avions beaucoup de petites tâches, ce qui nous a permis de bien avancer. Cependant, les tâches devenant de plus en plus grosses, les réunions ont pris de plus en plus de temps. Nous avons en effet rencontré beaucoup de problèmes qui étaient difficiles à résoudre seul ; nous en discutions donc durant les réunions, et celles-ci commençaient à prendre trop de temps. Après quelques séances peu efficaces, nous avons réalisé qu'il serait plus judicieux de séparer le 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 débloqué 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 (en fin de projet), 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 entravé le bon déroulement du projet et spécialement la fin de celui-ci 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 tourne. 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 ainsi tester et fine tuner rapidement des modèles. Nous avons donc entraîné plusieurs modèles sur Colab Pro, et les avons évalués avec le benchmark COCO - comme 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 éparpillé où il était difficile de retrouver la dernière version. Un autre point notable est le manque de crédit à la fin du projet, ce qui nous a empêché de réaliser 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 pu réaliser n'ont pas été concrétisées par manque de moyens. Nous voulions par exemple réaliser une centaine de prédictions sur un nouvel ensemble d'images et faire une fonction qui permet de filtrer certaines prédictions.

Nous avons dès le début défini toutes les pièces nécessaires au bon fonctionnement de notre projet final. Nous avions par exemple défini une pipeline - visible dans la figure 2.2. Cependant, nous pensons qu'il s'agissait d'une forme de *premature optimization is the root of all evil*. En effet, nous avons passé beaucoup de temps à faire les composants de la pipeline de manières indépendantes, bien que le projet final n'utilise pas forcément les composants ainsi créés.

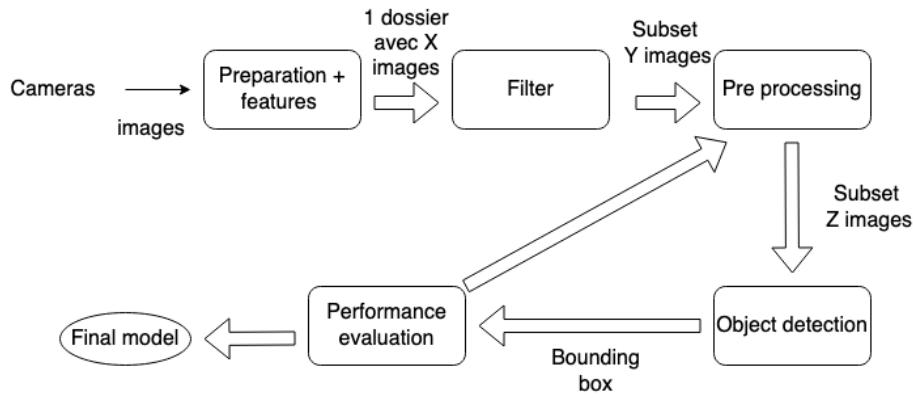


Figure 2.2: Pipeline du projet

En conclusion, nous avons rencontré de nombreux problèmes, la plupart dûs à notre faible connaissance du domaine. Néanmoins, nous sommes satisfaits de l'organisation et du déroulement de ce projet. Tous les membres du groupe ont travaillé sur des parties diverses et variées du projet et 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 fiers d'avoir su adapter notre organisation au cours du projet, afin de le mener à bien et ce malgré notre manque de connaissances 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 boxes 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 de la planche et dans la constitution de l'ensemble de données de test.

3.2 Conversion de format

Nous avons écrit plusieurs petit scripts Python permettant de convertir les labels en différents formats. Il existe plusieurs manières de définir les bounding boxes. Elles peuvent être définies comme un point d'ancre et une taille plus une hauteur ou simplement comme deux 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. Ainsi, nous avons créé plusieurs fonctions de conversion de csv à json, de xml à csv, ou encore de pascalvoc à csv, afin de pouvoir annoter les images avec l'outil colabeler.

3.3 Comptage des labels

Un petit script a été élaboré afin de compter les labels déjà effectués, ce qui permet d'avoir une liste des images déjà traitées et de constituer un sous-ensemble 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. Les chemins d'accès sont ainsi plus courts et plus lisibles, ce qui bénéficie à la compréhension du projet.

3.5 Fusion des dataframes

Nous avons également un script permettant de fusionner plusieurs fichier csv en un seul, ce qui permet de constituer un ensemble de données plus important afin de réaliser des analyses complètes. Par exemple, alors que les dossiers étaient partitionnés, notre fichier d'analyse regroupe ainsi tous les dossiers.

3.6 Tri des images

Nous avons également un script bash qui permet à partir d'une liste de fichier de déplacer tous les fichiers en une seule fois. Ainsi il est possible d'exporter un subset rapidement à partir d'une liste.

3.7 Réorganisation des données

Le script d'unnesting a permis de transformer la structure des données. Au début, la structure était partitionnée par caméra et par date, ce qui facilitait la navigation mais compliquait la gestion des fichiers. Ainsi, nous avons utilisé ce script pour changer le partitionnement uniquement par caméra, ce qui permet de gérer une arborescente moins profonde et regrouper le travail. De plus, ce script stocke les nouveaux résultats dans un csv ce qui permet de faire des analyses sur tout le dataset.

Chapter 4

Préparation des données

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 ; 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 chemin du fichier et non dans un fichier .csv externe. La nouvelle structure partitionnée par caméra 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 et sont toutes de la même taille, soit 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 en plusieurs images différentes. Nous avons donc considéré une séquence valide si sur la même caméra, les images sont prise à la suite dans un intervalle de temps inférieur à 2 secondes. Ce numéro de séquence 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 - comme introduit plus haut.

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 d'apprentissage supervisé, nous avons besoin de données labellisées manuellement 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 effectué avec le même outil de labellisation polyvalent, à savoir Colabeler, et sont décrites plus précisément dans les deux prochains paragraphes.

Classification Même si l'objectif final du projet n'est pas de classifier les images 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 principalement 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 crapauduc 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 (section 5.2).

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 des labels pour cette tâche de localisation que nous avons regroupés 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 certain 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 à certains moments. 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 toutefois important 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 grenouilles-crapauds 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 animal; l'ensemble de données compte en effet 425 images mais beaucoup moins de séquence puisque une séquence est composée de plusieurs images. 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

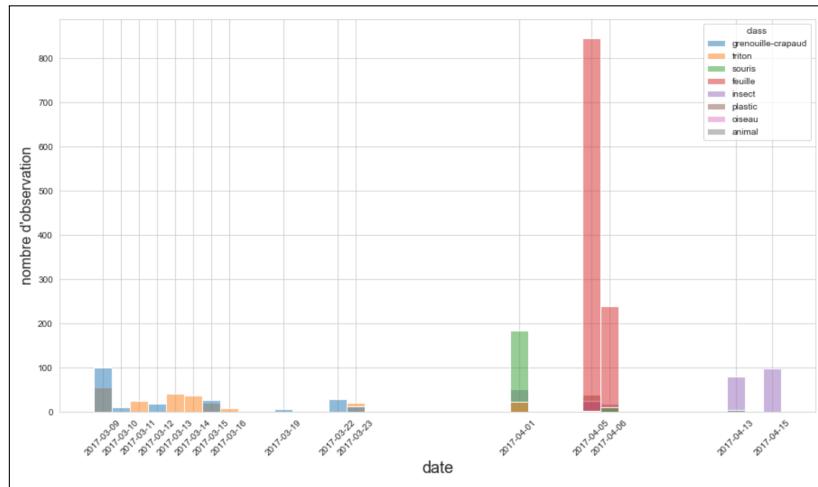


Figure 5.1: Fréquentation des animaux en fonction de la date

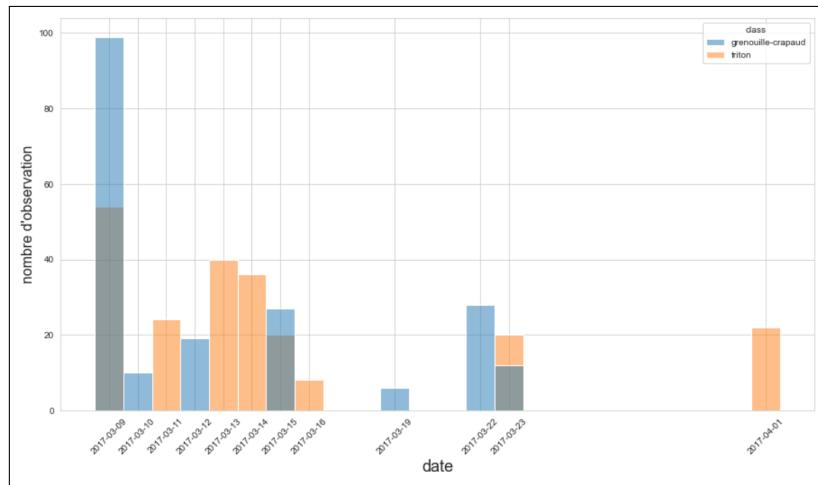


Figure 5.2: Fréquentation des batraciens observés en fonction de la date

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 ¹ sur internet, les batraciens se reproduisent en fin février-début mars. On peut donc considérer que la déduction de fréquentation plus élevée

¹<http://www.karch.ch/karch/home/amphibien/osservazione-di-anfibi.html>

des crapauds par les batraciens en mars peut être considérée pour un premier filtrage pertinent des images.

Heure

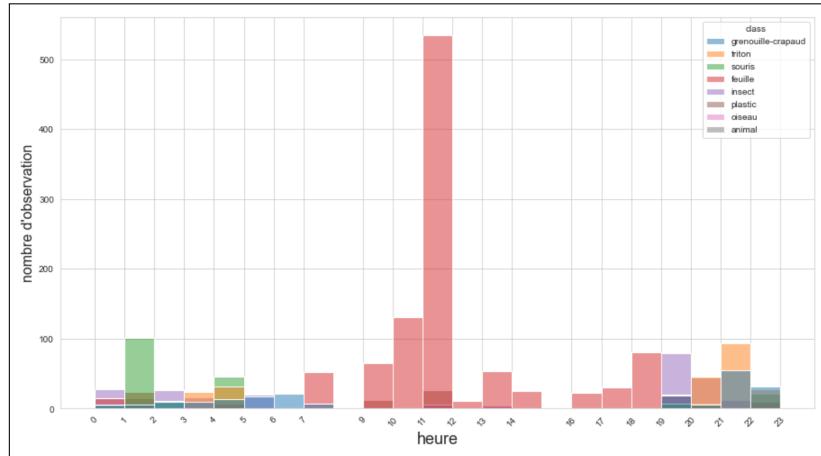


Figure 5.3: Fréquentation des animaux en fonction de l'heure

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 :

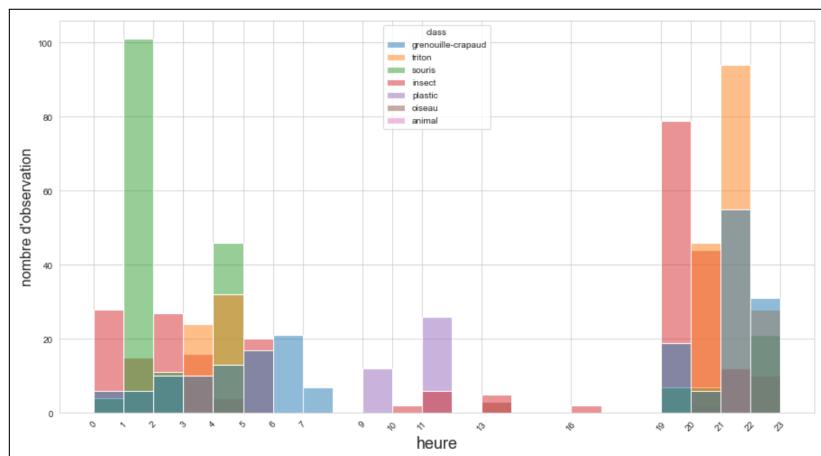


Figure 5.4: Fréquentation des batraciens observés en fonction de l'heure - sans les feuilles

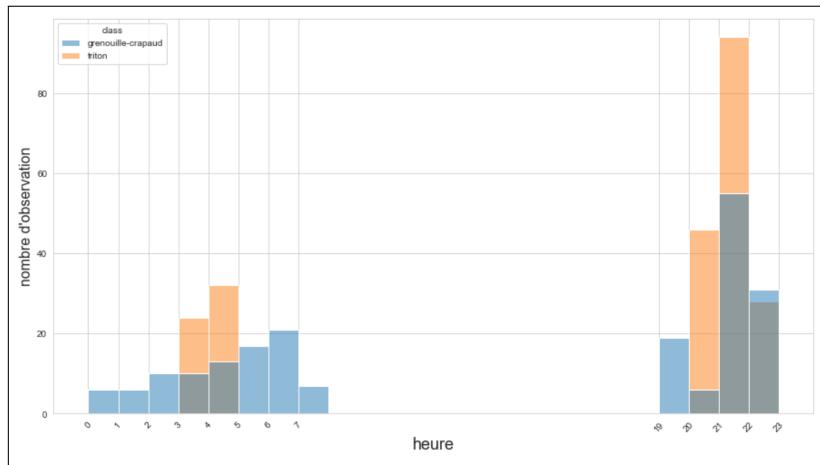


Figure 5.5: Fréquentation des batraciens observés en fonction de l'heure

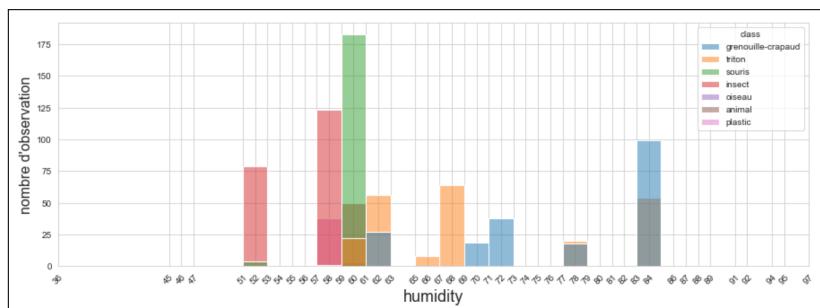
D'après la figure 5.5 ci-dessus, on observe que les batraciens d'intérêt utilisent particulièrement - même uniquement, pour cet ensemble de données - les crapauducs entre 19h et 7h du matin, c'est-à-dire de nuit. Le reste des observations (figure 5.4) confirme premièrement la pertinence de cette observation, 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



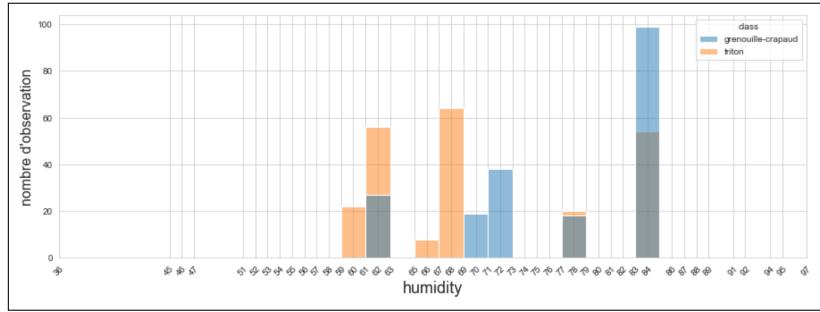


Figure 5.7: Fréquentation des batraciens en fonction de l'humidité - sans les feuilles

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 la présence ou non 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.

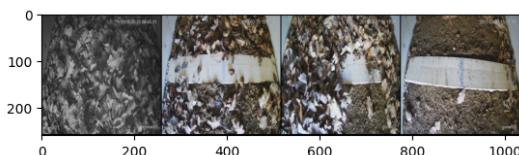


Figure 5.8: Exemple de données d'entraînement

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 respectivement 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 Nous avons calculé à partir de la figure 5.9 que 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. Ces résultats signifient donc 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

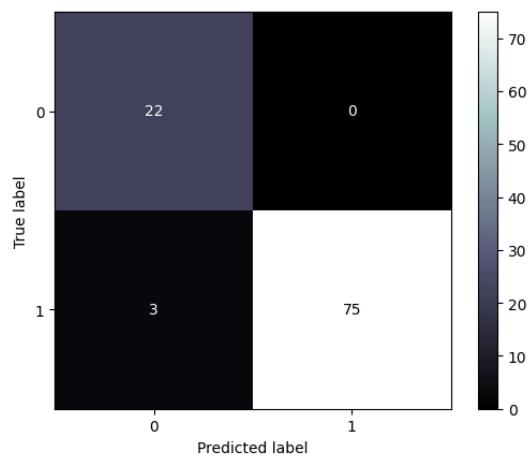


Figure 5.9: Matrice de confusion du détecteur de planche

acceptable. Lancé sur la quasi intégralité du dataset, 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. Il y a donc 10% d'images de non planches qui ne seront pas exploitables. Via ce filtre, nous pourrions les écarter lors d'une future utilisation.

Chapter 6

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.

6.1 Les échecs

6.1.1 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 années 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.

6.1.2 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 suivante : 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 du tutoriel que nous avons trouvé est d'une part, son ancienneté qui génère des erreurs de version ainsi que le manque d'assistance proposée. 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.

6.1.3 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 Détecteur). 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. 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 tutoriel 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.

6.2 Les réussites

6.2.1 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 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 classe `DefaultTrainer` une méthode qui affiche des statistiques sur l'entraînement en cours, comme le nombre d'epochs, le learning-rate, le temps restant, ce qui 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.

Entraîner 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 a permis de facilement changer de modèle et de tester plusieurs modèles rapidement. Nous avons aussi pu facilement faire du transfert learning en chargeant les poids d'un modèle pré-entraîné.

6.2.2 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.

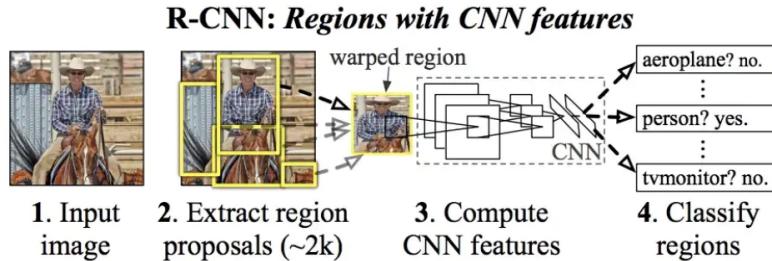


Figure 6.1: Architecture de R-CNN

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.

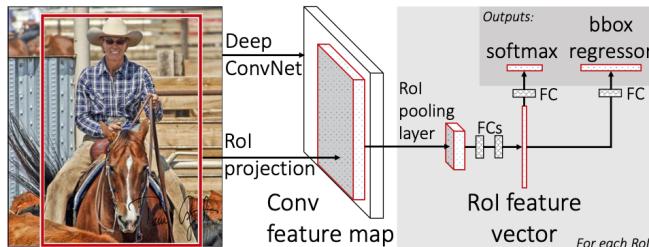


Figure 6.2: Architecture de Fast R-CNN

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.

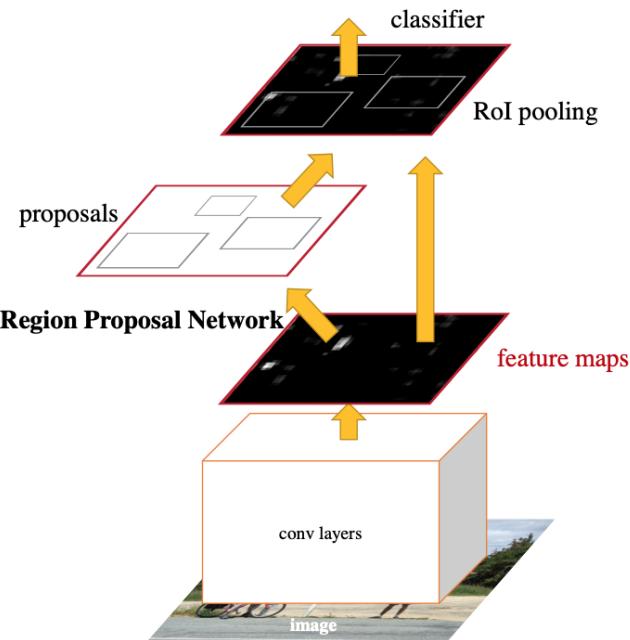


Figure 6.3: Architecture de Faster R-CNN

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 largeur 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 suivit d'un second réseau, FPN cette fois.

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 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.

6.2.3 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 suivi 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.

Il est intéressant de noter que ce modèle obtient des résultats extrêmement satisfaisants. En effet, il s'agit du modèle qui détecte les crapauds/grenouilles avec la plus grande précision. Les résultats précis de ce modèle sont détaillés dans la section "Evaluation" du chapitre 7 de ce rapport.

6.2.4 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 paru 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.

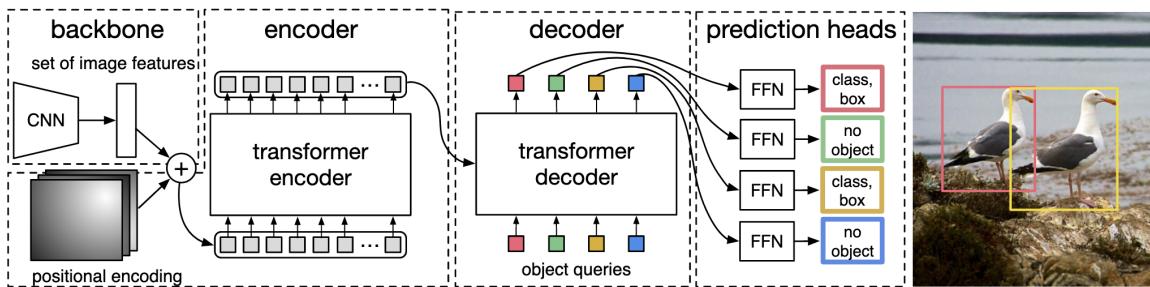


Figure 6.4: Architecture de DETR

Ce modèle est simple car il utilise peu de couches, contrairement à Faster R-CNN 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 celles visibles 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. Une version ayant tourné une fois sur un GPU premium est disponible sur notre repository 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



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

classe `COCO_Evaluator` de Detectron2. Nous avons testé sur quelques images et voyons de bons résultats. De plus, de bonnes métriques sont affichées durant l’entraînement. 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

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, c’est pourquoi nous avons donc décidé d’en rester là avec ce modèle.

6.3 Modèles évalués

D’après ces nombreuses recherches, nous avons donc implémenté et évaluer deux modèles fonctionnels dont nous avons pu comparer les scores d’évaluation sur des données de validation :

- 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 ainsi que le score COCO obtenu. 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 réelles (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 boxes 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

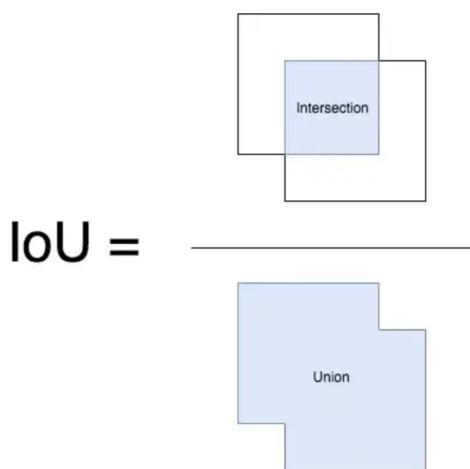


Figure 7.1: Intersection over Union

superposée sur la bounding box réelle, 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 avec l'API COCO¹. Il présente deux métriques 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 prédictive.

- Les trois premières lignes sont l'AP en considérant différents seuils 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 documentation¹, 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.

¹cocodataset.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

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 un 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 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 L'entraînement

Nous avons entraîné les modèles les uns après les autres sur un set de test composé de 176 tritons, 170 crapaud-grenouilles ainsi que de 150 souris. Ce dernier animal n'était pas demandé dans la tâche, mais nous nous sommes dit que ça permettrait d'ajouter de la diversité dans le dataset. L'évaluation a été effectuée sur un set composé de 48 tritons, 47 crapaud-grenouilles et 30 souris. Les labels ont été fournis par le professeur. Nous n'avons pas changé les paramètres généraux de Detectron2, uniquement les paramètres relatifs aux différents modèles, les entraînements se sont déroulés sur une GPU premium dont la spécification est visible dans le listing 7.1

```

1 NVIDIA-SMI:      460.32.03
2 Driver Version: 460.32.03
3 CUDA Version:   11.2
4 GPU Name:       A100-SXM4-40GB

```

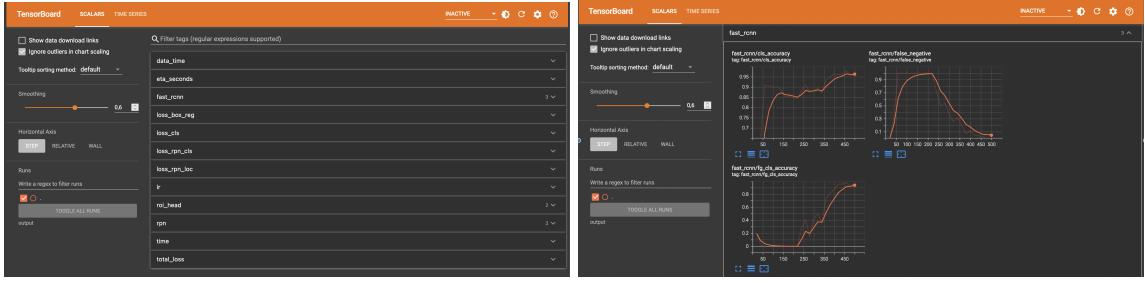
Listing 7.1: Résultat de la commande nvidia-smi

Les versions de Pytorch, ainsi que de Python étaient celles par défaut sur colab au moment de l'entraînement.

Nous allons maintenant détailler les différents entraînements effectués.

7.3.1 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.



(a) Aperçu du widget Tensorboard

(b) Aperçu de cls.accuracy, false_negative et foreground_cls_accuracy durant l'entraînement.

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

Nous avons pris quelques captures d'écrans affichées dans les figures 7.4a et 7.4b Detectron2 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 model et de l'over/under fitting. Comme nous pouvons le voir dans la figure 7.5, nous n'avons ni d'overfit, ni d'underfit.

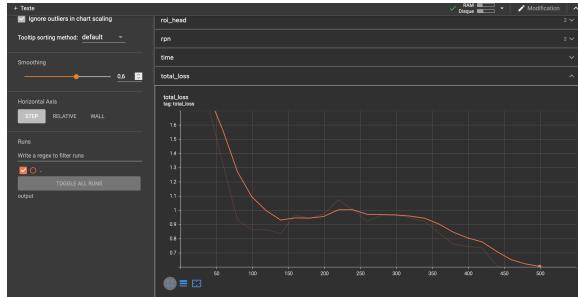
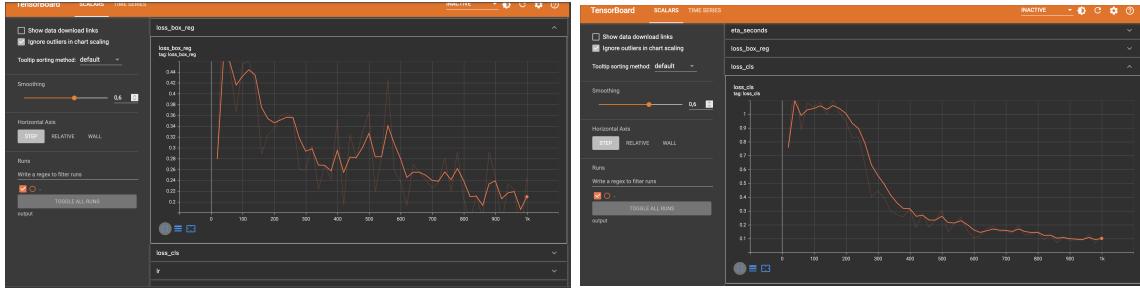


Figure 7.5: Aperçu de la loss total enregistrée dans TensorBoard

7.3.2 TensorBoard 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 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 calcule.



(a) Loss des bounding boxes

(b) Loss pour la classification

Figure 7.6: Les métriques de RetinaNet durant un entraînement

7.4 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.

7.4.1 Scores des modèles

Voici donc les scores obtenus avec Faster R-CNN (figure 7.7) et RetinaNet (figure 7.8) :

```
Average Precision (AP) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.347 ← (1)
Average Precision (AP) @[ IoU=0.50 | area= all | maxDets=100 ] = 0.773
Average Precision (AP) @[ IoU=0.75 | area= all | maxDets=100 ] = 0.283
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 ] = 0.025
Average Precision (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.357
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 1 ] = 0.424
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 10 ] = 0.426
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.426 ← (2)
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 ] = 0.100
Average Recall (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.435
[01/10 23:26:10 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
| AP | AP50 | AP75 | APs | APM | APl |
|:-----:|:-----:|:-----:|:-----:|:-----:|:-----:|
| 34.703 | 77.265 | 28.251 | nan | 2.525 | 35.700 |
[01/10 23:26:10 d2.evaluation.coco_evaluation]: Some metrics cannot be computed and is shown as NaN.
[01/10 23:26:10 d2.evaluation.coco_evaluation]: Per-category bbox AP:
| category | AP | category | AP | category | AP |
|:-----:|:-----:|:-----:|:-----:|:-----:|:-----:|
| None | nan | grenouille-crappaud | 39.811 | souris | 23.824 |
| triton | 40.475 | | | | ← (3)
```

Figure 7.7: Scores obtenus avec Faster R-CNN

```

Average Precision (AP) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.260
Average Precision (AP) @[ IoU=0.50 | area= all | maxDets=100 ] = 0.480
Average Precision (AP) @[ IoU=0.75 | area= all | maxDets=100 ] = 0.292
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 ] = 0.000
Average Precision (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.260
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 1 ] = 0.294
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 10 ] = 0.294
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.294
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 ] = 0.000
Average Recall (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.294
[12/09 13:05:28 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
| AP | AP50 | AP75 | APs | APm | APl |
|-----|-----|-----|-----|-----|-----|
| 25.978 | 48.050 | 29.236 | nan | 0.000 | 25.978 |
[12/09 13:05:28 d2.evaluation.coco_evaluation]: Some metrics cannot be computed and is shown as NaN.
[12/09 13:05:28 d2.evaluation.coco_evaluation]: Per-category bbox AP:
| category | AP | category | AP | category | AP |
|-----|-----|-----|-----|-----|-----|
| None | nan | grenouille-crapaud | 49.401 | souris | 28.531 |
| triton | 0.000 | | | | |

```

Figure 7.8: Scores obtenus avec RetinaNet

D'après le chapitre 7.2 vu plus tôt "Lecture d'un benchmark COCO", c'est la première ligne de ces résultats qui est la plus importante, c'est-à-dire la moyenne des mAP des bounding boxes qui ont un IoU entre 50 et 95 pourcent. On va donc ici observer ces valeurs, ainsi que l'average recall pour le même intervalle de IoU, avec la même valeur (=100) pour maxDets - nombre maximal d'objets détectables sur une image. On peut observer ces valeurs sur les figures 7.7 et 7.8, en (1) pour l'AP et en (2) pour l'AR ; les voici dans ce tableau :

modèle	average precision	average recall
Faster R-CNN	0.347	0.426
Retinanet	0.260	0.294

On voit donc premièrement que les deux scores sont meilleurs pour Faster R-CNN que pour RetinaNet. Aussi, en observant les AP par classe ((3) sur les figures 7.7 et 7.8), on a les résultats suivants :

modèle	AP grenouille-crapaud	AP triton
Faster R-CNN	39.811	40.475
Retinanet	49.401	0.000

On voit ici que 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.

Il est également ici intéressant de noter que le fait que la classe triton ait une AP nulle avec RetinaNet a un impact important sur le résultat de mAP. Il serait donc important et intéressant d'investiguer plus profondément sur la raison de l'échec complet de détection de triton ; comme discuté plus tard en section 8.3.1 de ce rapport.

7.4.2 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.DATALOADER.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.SOLVERIMS_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
```

Les paramètres choisis ont été trouvés et repris de la plupart des documentations trouvées en ligne. L'hypertuning s'est focalisé sur certains paramètres, en particulier ‘MAX_ITER’.

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

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

Le seuil choisi pour le test ci-dessus indique donc qu’on considérera une image comme étant prédite d’une certaine classe si le modèle détermine une probabilité supérieure ou égale à 70% que sa classification soit correcte.

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



Figure 7.9: Prédiction de Faster R-CNN - crapaud-grenouille



Figure 7.10: Prédiction de Faster R-CNN - triton

On observe donc ici une bonne prédiction du modèle, qui est quasi certain de son classement (96% et 97%). Il est cependant intéressant et surtout important de noter que le modèle a été entraîné sur des images qui peuvent être très semblables à celles sur lesquelles il a été évalué ensuite. En effet, les images que nous avons prises pour l’entraînement et pour l’évaluation proviennent d’un ensemble d’images pouvant provenir de la même séquence ou du même crapauduc, ce qui peut provoquer de l’overfitting. Cette problématique sera discutée plus en profondeur dans le chapitre 8.

7.5 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 7.3.1 ainsi que 7.3.2, 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 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 DETR¹ atteint 64.5 de mAP 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 - qui a 60 millions de paramètres alors que leur modèle en a 600 millions.

7.5.1 Analyse des images non détectés par Faster-RCNN

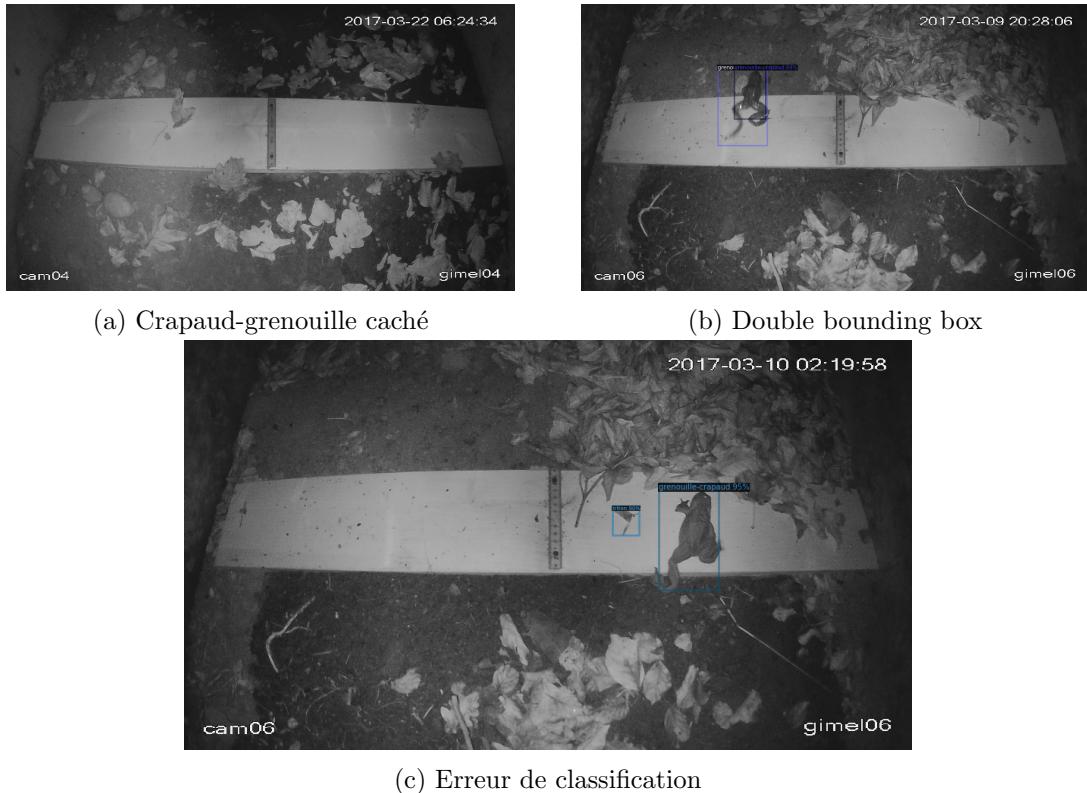


Figure 7.11: Exemple d'images mal prédites par Faster-RCNN

La plupart des images qui présentent des erreurs pouvant expliquer le score de Faster-RCNN ressemble aux images de la figure 7.11. 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 feuilles pourrait expliquer cela. Un préprocessing pourrait vérifier l'hypothèse de la luminosité. Cependant, nous ne pouvons pas vraiment faire de préprocessing pour la présence de feuille.

La seconde est aussi assez fréquente, il s'agit de la présence de multiples bounding boxes 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 seuil de confiance (threshold) pour réduire ce genre d'erreur.

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

La dernière est la seule occurrence de ce type observée sur le set de test ; 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 R-CNN est vraiment bon et les erreurs ne présentent pas de graves erreurs d'entraînement.

7.5.2 Analyse des images non détectées par RetinaNet

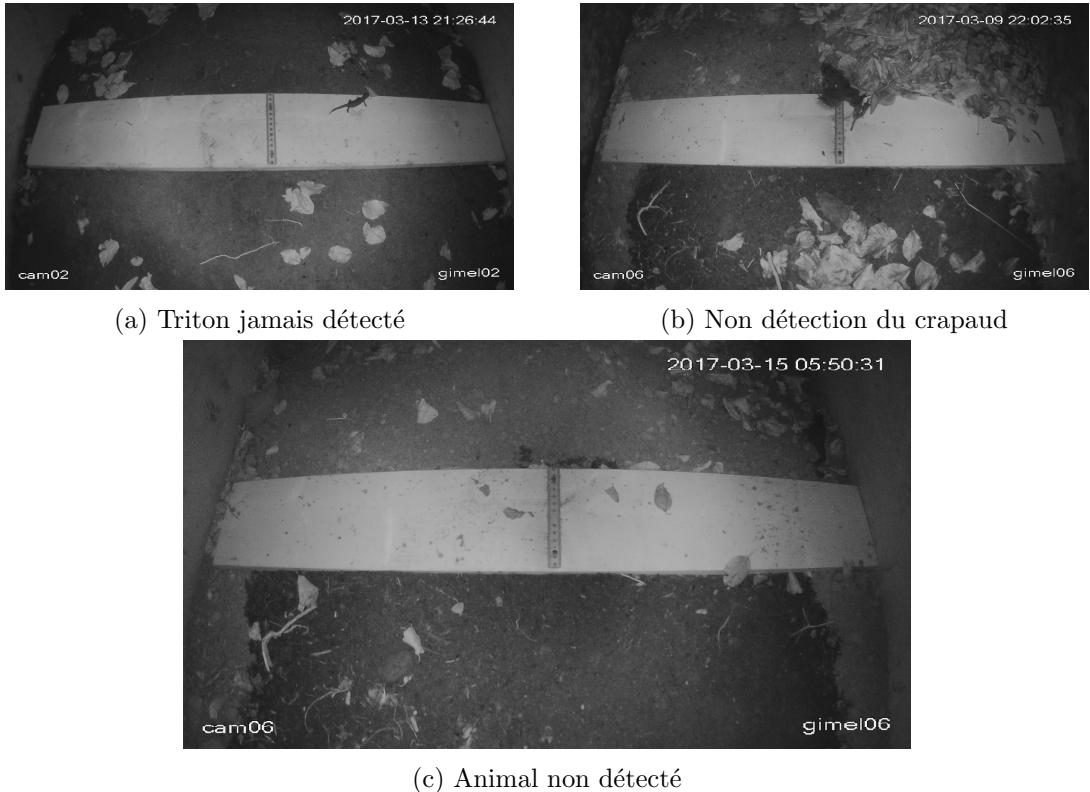


Figure 7.12: Exemple d'images mal prédites par RetinaNet

Nous pouvons voir dans la figure 7.12 que RetinaNet ne détecte jamais les tritons, ce qui confirme les scores du benchmark COCO vu précédemment. Nous avons pensé que c'était dû à la petite taille de ces derniers, cependant nous pouvons observé que l'image 7.12b 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 détectés par RetinaNet. L'image 7.12c nous montre un problème similaire à celui vu précédemment sur l'image 7.11a, ce qui tend à confirmer l'hypothèse du manque de lumière sur les cotés.

Après avoir regardé toutes les prédictions sur les images du set de test pour les deux modèles, nous avons remarqué que plusieurs images semblent provenir de la même séquence. Ainsi, lorsque la détection ne fonctionne pas, elle ne fonctionne pas plusieurs fois! Ceci implique que les erreurs sont comptabilisées à double. Cependant, nous ne pouvons pas crier victoire et dire que nos scores sont bien en dessous de leur vrai valeurs car nous observons aussi ceci sur les prédictions correctes! Ainsi les bons scores et mauvais scores à double se compensent.

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 placées au centre de l'image, que les modèles détectent mieux les animaux au centre de l'image de par leur entraînement. Néanmoins, un ajustement de la luminosité serait une piste de réflexion.

Chapter 8

Prochaines étapes

8.1 Évaluation des modèles sur le set de test

La phase de test est l'une des plus importantes dans tout projet de machine learning. Cette dernière tend à confronter le modèle à la réalité du terrain. Donc il est important de ne pas introduire de biais dans les données utilisées dans celle-ci. Par biais nous entendons par exemple tester le modèle sur des données très similaires à celles utilisées durant le train ; le faire pourrait en effet certes nous donner de bons résultats, mais ils seraient biaisés.

Dans un premier temps, nous avions évalué le modèle sous la base d'un simple test split effectué sur des données issues de mêmes caméras. Le faire sous-entend que la plupart des images du set de test et du set de train utilisés font partie des mêmes séquences. Et comme vu avant, cette ressemblance rend discutable les résultats obtenus. Pour pallier cela, nous avons donc construit deux nouveaux jeux de test avec des images issues des caméras 1 et 3, deux caméras jamais utilisées dans le cadre de notre projet. Nous avons labelisé un total de 371 images provenant de la caméra 1 et 172 images de la caméra3 avec l'outils colabeler. Le but étant de tester les modèles sur chacun d'eux et confronter les résultats et ainsi confirmer ou infirmer les résultats actuels.

8.2 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 série d'images est prise. De ce fait plusieurs images par exemple de tritons à différentes positions peuvent représenter le passage d'un 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 à leur comptage. En d'autres termes, nous allons commencer par compter les animaux par images, puis regrouper les détections par prise. Par la suite, nous n'allons conserver pour chaque prise

et pour animal, que le nombre maximal de ses occurrences comptées sur les images de cette prise. Ce nombre in fine va représenter le nombre d'animaux de ce type pour cette prise.

8.3 Comparaison avec le cahier des charges

Lors de l'élaboration du cahier des charges de ce projet, nous avons fixé un certain nombre d'objectifs que nous souhaitions atteindre. Ces objectifs sont divisés en deux grandes catégories, à savoir les "must have" et les "nice to have". Chacune de ces catégories contient trois objectifs distincts, et on va détailler ci-dessous dans quelle mesure ces différents objectifs ont été atteints ou à l'inverse ont échoué.

8.3.1 Must have

Il s'agit des objectifs à atteindre obligatoirement avant la fin du projet dans la mesure du possible.

Dessiner une bounding box autour des tritons

Comme on peut le constater dans la partie 7.4, les deux modèles obtiennent des résultats très différents concernant les tritons. En effet, le modèle RetinaNet est malheureusement incapable de détecter le moindre triton sur les images. En revanche, le modèle Faster R-CNN obtient un AP score plutôt correct de 42.58 sur les tritons. Comme nous avons choisi ce dernier modèle en tant que modèle final, on peut considérer que ce premier objectif obligatoire est atteint.

Compter les tritons

Comme expliqué dans la section 8.2, le comptage des tritons et plus globalement des animaux en général s'est avéré être une tâche plus complexe que prévu initialement. En effet, même si cette tâche peut sembler en apparence relativement simple une fois les animaux détectés sur les images, plusieurs problèmes explicités dans le chapitre 8.2 sont intervenus et nous ont malheureusement empêché d'atteindre cet objectif.

Détecter les crapauds-grenouilles indépendamment des tritons

Comme on peut le constater dans la partie 7.4, les deux modèles obtiennent des résultats relativement similaires quant à la détection des crapauds-grenouilles sur les images. En effet, ils obtiennent tous les deux un AP score situé entre 40 et 50. Le modèle RetinaNet est toutefois légèrement meilleur puisqu'il parvient à obtenir un AP score de 49.40, contre seulement 42.03 pour Faster R-CNN. Cependant, comme expliqué ci-dessus, RetinaNet ne détecte aucun triton et nous avons donc préféré Faster R-CNN pour notre modèle final. Toujours est-il que le résultat obtenu par Faster R-CNN sur les crapauds-grenouilles reste plutôt satisfaisant et on peut donc considérer que ce troisième objectif est une réussite.

8.3.2 Nice to have

Il s'agit des objectifs qu'il serait intéressant d'atteindre si le temps le permet mais qui sont d'importance secondaire.

Distinguer les tritons, les crapauds et les grenouilles

Même si la formulation n'est pas très explicite, l'objectif ici était surtout de distinguer les crapauds des grenouilles puisque la distinction entre les tritons et les crapauds-grenouilles était déjà sous-entendue par le dernier objectif "must have". Hélas, cet objectif est un échec puisque dans notre modèle final décrit dans la section 7.4.2, nous avons gardé la catégorie crapaud-grenouille en raison des difficultés rencontrées pour différencier ces deux animaux. En effet, les différences entre ces deux espèces sont minimes et même pour un être humain, il est difficile de les distinguer sans être un expert. La marche était donc tout simplement trop haute pour nos réseaux de neurones.

Compter le nombre de tritons, crapauds et grenouilles qui ont traversé le tunnel dans une période donnée

Cet objectif était en quelque sorte le prolongement de l'objectif "must have" détaillé dans la partie 8.3.1. Cet objectif initial n'ayant pas pu être atteint, nous n'avons par conséquent pas pu faire aboutir sa version améliorée pour les mêmes raisons. Sans entrer dans les détails de ces raisons qui sont expliqués ci-dessus, cet objectif peut donc être considéré comme un nouvel échec.

Déterminer le sens de la traversée d'un animal

Cet objectif était pertinent principalement pour compter le nombre d'animaux qui traversent les tunnels dans chaque sens. Comme nous n'avons pas été en mesure d'atteindre l'objectif de comptage explicité ci-dessus, nous n'avons pas vraiment essayé de déterminer le sens de la traversée des animaux. De plus, au vu des difficultés rencontrées pour distinguer les crapauds des grenouilles, cette tâche aurait probablement été un peu trop difficile par rapport au bénéfice rapporté par la connaissance du sens de traversée. On peut donc considérer que cet objectif n'a pas été atteint ni même commencé.

8.4 Expériences futurs

Comme dit dans la section 7.5, nous avons remarqué que les animaux sur les côtés de l'image sont souvent mal détectés. Nous avons pensé à deux expériences permettant de possiblement résoudre ce problème. Premièrement, nous pensons qu'il serait judicieux de faire une correction non linéaire de la luminosité comme illustré en figure 8.1 et observer après traitements les scores. Il est selon nous nécessaire que la correction soit non linéaire afin de ne pas saturer les zones claires, car il s'agit du centre de l'image, où passent de nombreux animaux.

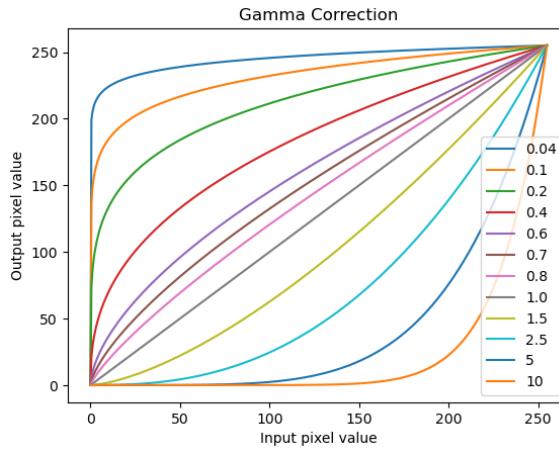


Figure 8.1: Correction non linéaire de la luminosité $O = \left(\frac{I}{255}\right)^{\gamma} 255$

Une seconde options que nous n'avons, elle non plus, pas eu le temps de mettre en oeuvre est l'utilisation d'une Grad-Cam afin de détecter les régions d'intérêts sur les images. Cette méthode permet de détecter les régions d'intérêt sur les images et de les mettre en évidence. Nous pensons que cela permettrait de savoir si les côtés sont *observés* par le modèle.

Analyse de la sensibilité Une expérience que nous aurions voulu faire est une technique inspirée de la data augmentation. Il s'agit d'occulter des parties de la bounding box afin de voir les parties de l'objet qui déclenche une détection par le modèle.

Let's zoom La grosse différence entre les classes présentes sur COCO - le dataset d'entraînement originel - et notre dataset de fine-tuning est la taille des objets. En effet, les animaux que nous avons sont beaucoup plus petits que ceux du dataset COCO et nous aimerais ainsi savoir si faire la prédiction en deux étapes - d'abord sur l'image entière puis une seconde fois sur un zoom de la zone prédictive - permettrait d'améliorer les scores.

Bootstrapping Vu que nos modèles se trompent rarement - l'erreur la plus commune étant la non détection - il serait intéressant d'essayer de les entraîner depuis 0 en utilisant un plus gros dataset, annoté par la version précédente du modèle. Cela permettrait de voir si le modèle apprend mieux et si les scores augmentent.

Resizing Durant notre périple, nous avons - comme mentionné précédemment - exploré différents modèles. Pour la plupart d'entre eux, la taille des images demandée en entrée était de 300*300, 500*500 pour SSD, 512*512 YOLOv5 - Cas des images carrées. Les deux modèles précités pour plusieurs raisons dont celle de la taille des entrées constituent des échecs dans notre projet. Car nous rappelons que nous travaillons avec des images de taille 1920*1080. Nous pensons donc que pour de futures expériences, il serait intéressant de redimensionner les images non seulement pour permettre que celles-ci s'adaptent à plusieurs modèles, mais aussi pour accélérer le temps de traitement. Passer d'une taille d'image de 1920*1080 à une taille de 300*300 par exemple ne constitue pas pour la tâche que nous faisons une perte d'informations handicapante. À confirmer avec une expérience.

Chapter 9

Conclusion

En conclusion, on peut estimer que le bilan global de ce projet est en demi-teinte. En effet, nous ne sommes pas parvenus à atteindre tous les objectifs fixés par le cahier des charges. Plus précisément, nous avons atteint seulement deux objectifs parmi les trois "must have" et aucun objectif parmi les trois "nice to have". Ces résultats contrastés sont principalement dûs aux difficultés inattendues rencontrées lors du comptage des animaux. Cependant, l'objectif principal restait de détecter les animaux présents sur les photos ainsi que de différencier les tritons des crapauds-grenouilles. Cet objectif ayant été atteint, non sans difficulté, on peut considérer que le bilan du projet sur le plan technique est plutôt correct.

Au-delà de l'aspect technique, la gestion de projet représentait également une partie importante de ce projet. Sur ce plan-là, nous pensons tous avoir beaucoup appris au fur et à mesure de l'avancement des différentes étapes du projet. En effet, même si au début la gestion du projet était plutôt chaotique, nous avons réussi à mettre en place des méthodes de travail et de collaboration qui nous ont permis d'avancer de manière plus efficace et qui nous seront probablement utiles pour nos projets futurs et la suite de nos parcours en tant qu'ingénieurs des données. Au vu de cette amélioration au fil du projet, on peut considérer que la gestion de projet est plutôt une réussite même si ce n'était pas gagné d'avance.

Une conséquence malheureuse de cette entame de projet laborieuse concerne la reproductibilité des expériences effectuées. En effet, comme nous avions tendance à partir dans tous les sens au début du projet, l'organisation des différents fichiers sur le github ou le google drive est relativement chaotique et difficilement compréhensible pour un nouveau venu. Si le projet était à refaire, nous mettrions probablement l'accent sur cet aspect afin de livrer une hiérarchie de fichiers plus lisible et réutilisable par la suite. Malheureusement, nous sommes partis sur de mauvaises bases et ce genre de problème est difficile à résoudre en cours de projet. La reproductibilité du projet représente donc à nos yeux l'un des des échecs de ce projet.

Finalement, on peut dire que nous avons rencontré un nombre relativement important de difficultés, tant sur le plan technique que sur le plan de la gestion de projet. En effet, nous n'avons pas réussi à atteindre tous les objectifs fixés et la gestion de projet fut parfois extrêmement laborieuse. Cependant, nous avons énormément appris sur ces deux aspects et si le projet était à refaire dans les mêmes conditions, nous obtiendrions probablement des résultats beaucoup plus consistants. Comme il s'agit d'un projet effectué dans une

école, la visée est avant tout pédagogique et l'objectif principal est d'apprendre de nos erreurs pour ne pas les reproduire dans le monde du travail. Sur ce point-là, nous sommes donc extrêmement satisfaits car toutes les erreurs effectuées ne se reproduiront plus et nous permettent à toutes et tous d'aller de l'avant et de rejoindre plus sereinement le monde de l'industrie après la fin prochaine de nos études.