Détection et reconnaissance de chiffres manuscrits à partir de plans cadastraux anciens par apprentissage profond

Margot Glenaz

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

ECOLE SUPERIEURE DES GEOMETRES ET TOPOGRAPHES

MEMOIRE

présenté en vue d'obtenir

le DIPLOME D'INGENIEUR CNAM

SPECIALITE : Géomètre et Topographe

par

Margot GLENAZ

Détection et reconnaissance de chiffres manuscrits à partir de plans cadastraux anciens par apprentissage profond

Soutenu le 06 septembre 2021

JURY

Monsieur Mathieu BONNEFOND	Président du jury
Monsieur Frédéric DURAND	Maitre de stage
Monsieur Jean-Michel FOLLIN	Maître de stage
Madame Elisabeth SIMONETTO	Maitre de stage
Madame Nathalie THOMMERET	Enseignant référent

Remerciements

Je tiens à remercier toutes les personnes qui ont contribué à l'aboutissement de mon stage et à la rédaction de ce mémoire.

Je souhaite tout d'abord remercier mes maitres de stage, Monsieur Frédéric DURAND, Monsieur Jean-Michel FOLLIN et Madame Elisabeth SIMONETTO, pour leurs disponibilités et leurs conseils qu'ils m'ont prodigués tout au long de ce Travail de Fin d'Étude.

Je souhaite également remercier Madame Nathalie THOMMERET pour m'avoir conseillé et avoir relu et corrigé mon mémoire.

Je remercie aussi toute l'équipe enseignante de l'École Supérieure des Géomètres et Topographes, pour l'ensemble des connaissances apportées tout au long de mon cursus.

Je remercie aussi ma famille pour leurs encouragements et leur soutien tout au long de ma scolarité.

Liste des abréviations

AP : Average Precision

ARDIS : ARkiv Digital Sweden

CNN : Convolutive Neural Network (réseau de neurones convolutifs)

CPU : Central Processing Unit

Fast R-CNN : Fast Region based Convolutional Neural Network (réseau de neurones convolutifs rapide basé sur les régions)

Faster R-CNN : Faster Region based Convolutional Neural Network (réseau de neurones convolutifs plus rapide basé sur les régions)

GeF : Laboratoire Géomatique et Foncier

GPU : Graphical Processing Unit

IA : Intelligence Artificielle

IoU : Intersection over Union

mAP : mean Average Precision

MNIST : Modified National Institute of Standards and Technology

MSE : Mean Square Error (Erreur quadratique moyenne)

MSH : Maison des Sciences de l'Homme

NIST : National Institute of Standards and Technology

RBC : Red Blood Cell (Globule rouge)

R-CNN : Region based Convolutional Neural Network (réseau de neurones convolutifs basé sur les régions)

ReLU : Rectified Linear Unit (unité linéaire rectifiée)

RGB : Red Green Blue (Rouge Vert Bleu)

RMSE : Root Mean Square Error (racine carrée de l'erreur quadratique moyenne)

RNN : Recurrent Neural Network (réseau de neurones récurrents)

RoI : Region of Interest (région d'intérêt)

RPN : Region Proposals Network (réseau de propositions de régions)

SGD : Stochastic Gradient Descent (descente de gradient stochastique)

SVM : Support Vector Machine

USPS : United-States Postal Service

WBC : White Blood Cell (Globule blanc)

YOLO : You Only Look Once

Table des matières

Introduction

Depuis plusieurs décennies l'intelligence artificielle (IA) fascine autant les scientifiques que les scénaristes de science-fiction. Lorsque le sujet de l'IA est abordé, nous pensons tout de suite au futur et à des robots qui seront capables de remplacer l'Homme pour des tâches complexes et fastidieuses, voire de le surpasser. Cependant, les domaines d'application de l'IA sont plus variés que la seule robotique et peuvent aussi bien concerner la médecine, les transports que la finance. L'IA est définie par [Chollet, 2020] comme étant « l'effort d'automatisation des tâches intellectuelles normalement effectuées par des humains », ce qui est une définition qui concerne de nombreuses applications.

Parmi les tâches complexes et chronophages, il y a la vectorisation des cartes anciennes et scannées. Elle permet de retranscrire les données contenues dans les cartes (frontières, bâtiments, noms des villes…) au format vectoriel, qui sert à traduire l'information en géométries auxquelles des attributs peuvent être attachés. Les objets ne sont plus représentés par des pixels mais par une fonction qui, à chaque fois que nous agrandissons l'image par exemple, recalcule la géométrie de l'objet.

Les cartes anciennes constituent un patrimoine important de l'Histoire et permettent de représenter un territoire à différentes périodes. Elles sont aussi des outils intéressants pour étudier l'évolution et la politique d'aménagement des espaces, et pour comprendre comment et pourquoi le territoire a évolué dans sa configuration actuelle. Cependant, en plus d'être fastidieuse et répétitive, la vectorisation manuelle des cartes mobilise toute l'attention d'un opérateur, et malgré tout, il peut commettre des erreurs dans l'interprétation et la retranscription des données. C'est pourquoi, l'automatisation de la vectorisation des cartes anciennes à l'aide de l'IA est un sujet prometteur, et permettrait un gain de temps non négligeable.

Toutefois, l'IA est un domaine très large. Au sein de celle-ci, il y a la catégorie de l'apprentissage automatique, ou *machine learning* en anglais. Ce sous-ensemble répertorie les technologies qui permettent aux ordinateurs d'apprendre à effectuer une tâche, sans qu'ils aient reçu des consignes explicites. En effet, [Chollet, 2020] explique que la programmation dite « classique » consiste à fournir à un ordinateur des données d'entrée et des règles précises qu'il devra appliquer pour décider du résultat. Le principe de l'apprentissage automatique, lui, réside dans le fait que l'opérateur donne des valeurs d'entrée et, à la place des règles, les résultats attendus liés à ces valeurs d'entrée. Ainsi, l'ordinateur déduit les

règles à appliquer pour que ces prédictions soient cohérentes avec les résultats attendus. Nous apparentons cela à de l'apprentissage.

Quant à lui, le *deep learning*, ou apprentissage profond en français, est un type de *machine learning* qui met en place des réseaux profonds composés de couches de neurones superposées. Plus un réseau est composé de couches de neurone et plus il sera qualifié de profond. Les modèles *deep learning* sont de plus en plus utilisés notamment dans le domaine de la vision par ordinateur. Il semble que ce soit une piste prometteuse pour automatiser les tâches de reconnaissance d'objets dans les images.

Ce mémoire s'inscrit dans le cadre du projet GefVectoMoCad mis en place par le laboratoire Géomatique et Foncier (GeF) du Cnam au Mans. Dans sa globalité, ce projet consiste à vectoriser, géoréférencer et mosaïquer des images de planches cadastrales anciennes scannées. Et cela dans le but de créer une base de données répertoriant les caractéristiques d'un territoire au fil du temps. Les processus élaborés jusqu'ici sont appliqués aux plans cadastraux de 1873, 1850, 1972 et 1974 des communes de Vaas et d'Aubigné situées dans le Sud de la Sarthe. La chaîne de traitement est constituée d'étapes semi-automatiques élaborées avec des outils libres comme Python par exemple. Au travers de plusieurs travaux effectués par des étudiants (le TFE de M. Fahrasmane de 2016, le TFE de C. Odie de 2017, le TFE de JM Beveraggi de 2018, le TFE de A. Chalais de 2019 et plusieurs projets étudiants à partir de 2014), la chaîne de traitement a été améliorée. Le travail de [Chalais, 2019] a notamment permis de retranscrire l'entièreté du processus sous Python, afin d'éviter la multiplicité des logiciels utilisés. De plus, son travail a amélioré le processus de vectorisation des contours de parcelles en utilisant la méthode de détection de ligne nommée *Line Segment Detector*. Malgré de bons résultats obtenus à l'issue de cette chaîne de traitement, [Chalais, 2019] explique qu'une amélioration pourrait être apportée grâce à la détection des numéros de parcelle. En effet, si les numéros des parcelles étaient détectés, cela permettrait un contrôle concernant le nombre de parcelles à trouver. De plus, si plusieurs numéros sont trouvés au sein d'une même parcelle, cela pourrait indiquer que la détection a omis certains contours. Dans ce but, le laboratoire GeF souhaite explorer la piste du *deep learning* pour procéder à la détection des numéros de parcelles. En effet, l'utilisation du *deep learning* semble plus appropriée que les autres algorithmes d'apprentissage automatique car les numéros de parcelles sont manuscrits.

C'est pourquoi, l'objectif de ce mémoire est d'étudier la mise en place d'un réseau de *deep learning* chargé de détecter et de reconnaitre des numéros de parcelles manuscrits

dans des documents anciens. Ainsi, nous verrons dans une première partie les difficultés de la vectorisation des documents anciens ainsi que les notions générales qui régissent le *deep learning*. Dans une deuxième partie, nous verrons plus en détails la mise en place d'un réseaux Faster R-CNN et nous expliquerons l'élaboration du jeu de données utilisé dans le cas de la recherche de caractères numériques manuscrits. Enfin, la dernière partie concernera les résultats obtenus lors des tests de ce modèle ainsi que les perspectives envisagées.

I Mise en contexte

I.1 Problématiques soulevées lors de la vectorisation de cartes anciennes

I.1.1 Notions générales

La plupart du temps, les cartes anciennes originales sont conservées pour éviter leur détérioration. Celles-ci sont précieuses puisqu'il n'en existe qu'un seul exemplaire et que malgré les dispositions prises, le temps provoque des effets irréversibles sur le papier et l'encre ou le crayon. Cependant, pour la majorité d'entre elles et pour éviter de perdre les connaissances contenues dans ces cartes, elles ont été numérisées. La numérisation est définie par [ESRI] comme étant le processus qui « consiste à capturer les données de cartes imprimées ou d'images au format numérique à l'aide d'un appareil appelé scanner ». Ainsi, elle permet d'obtenir la carte au format raster, c'est-à-dire une image constituée de pixels.

Toutefois, le format raster présente des inconvénients. En effet, la conversion en image numérique peut engendrer des bruits comme par exemple le crénelage des contours ou un effet poivre et sel. De plus, en fonction du capteur photographique du scanner, la numérisation peut être de différentes qualités et ajouter des imperfections à la carte. Ainsi, la numérisation n'est pas une finalité en soi.

En revanche, pour effectuer des traitements sur des objets contenus sur la carte, il existe le format vectoriel. Un objet au format vecteur est une « représentation mathématique d'objets situés dans l'espace et portant donc leurs coordonnées dans un système de géoréférencement géographique. […] Le vecteur permet un changement d'échelle sans changement de résolution, car tout objet vecteur se ramène mathématiquement à des ensembles de points sans superficie. » [ALPAGE]. Ainsi, ce format est à privilégier pour l'exploitation et la conservation des cartes. La vectorisation est le procédé permettant de « passer d'une image raster à un objet vecteur » [ALPAGE].

I.1.2 La vectorisation de documents anciens : problématiques liées au *deep learning*

Comme vu en introduction, l'un des objectifs du laboratoire GeF est d'améliorer la chaîne de traitements déjà élaborée pour géoréférencer, vectoriser et mosaïquer des planches cadastrales anciennes. Pour cela, la détection des numéros de parcelles semble être un bon critère pour obtenir un contrôle des parcelles détectées avec la chaîne de traitements déjà produite. Cependant, la détection et la reconnaissance de caractères numériques manuscrits est une tâche difficile à mettre en place avec des méthodes semi-automatique. En effet, une

simple classification semble compliquée car, bien que des numéros soient écrits par une même personne (ce qui n'est pas toujours le cas dans les documents anciens), un chiffre ne sera jamais écrit exactement de la même manière. De plus, les travaux de [LeCun et al., 1989] étant les premiers à traiter du *deep learning*, et qui plus est de la reconnaissance de chiffres manuscrits, il nous a semblé judicieux d'explorer la piste du *deep learning* pour notre propre problématique.

Plus généralement, de nombreux travaux portent sur la vectorisation de cartes anciennes et avec les progrès technologiques, il serait dommage de ne pas essayer d'automatiser ce processus. Cependant, plusieurs problèmes peuvent perturber la vectorisation automatique de carte, ce qui rend difficile l'automatisation complète du processus et oblige malgré tout à conserver des étapes semi-automatiques.

L'un des premiers obstacles rencontré lors de la création d'un réseau d'apprentissage profond est le nombre d'échantillons à utiliser comme jeu de données. En effet, [Laumer et al., 2020], [Heitzler et al., 2020] ou encore [Dupas, 2021] sont d'accord pour dire que le jeu de données d'entraînement doit être suffisamment fourni. Bien évidemment, la taille du jeu de données dépend du nombre de classes que nous souhaitons identifier dans les données, mais aussi à quel point les classes sont différentes entre elles. Plus elles auront des caractéristiques communes et plus le nombre d'échantillons d'entraînements devra être important pour que le modèle apprenne à les différencier. Malgré tout, [Mitsa, 2019] indique que plus le jeu de données sera important et plus les performances du réseau d'apprentissage profond seront bonnes, ce qui n'est pas le cas des autres algorithmes de machine learning (voir Figure 1). En général, pour de la reconnaissance d'image, les jeux de données peuvent contenir entre 100 et plusieurs milliers d'échantillons par classe.

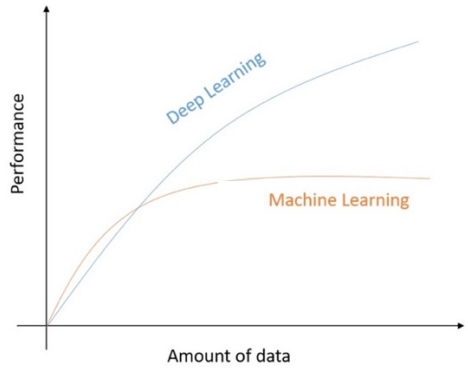

Figure 1 : Performance du deep learning et du machine learning en fonction du nombre de données
Source : Md Z.Alom, T.M. Taha, C. Yakopcic et al, *A State-of-the-Art Survey on Deep Learning Theory and Architectures*, 2019, Research Gate, [en ligne]. 8. 292. 10.3390/electronics8030292

Un autre problème qui concerne la mise en place du jeu de données est le temps de création de ce jeu. En effet, l'objectif premier de l'automatisation d'une tâche par l'utilisation d'un modèle *deep learning* est de ne plus avoir à référencer manuellement les données. Or, à moins d'avoir accès à un jeu de données déjà produit, il est nécessaire d'étiqueter soi-même la donnée, ce qui peut être aussi long et fastidieux. La plupart du temps, il est possible de rendre semi-automatique l'étiquetage des données en élaborant un programme Python par exemple. Cela peut donc constituer un frein dans le processus d'automatisation de la vectorisation de cartes anciennes.

Ensuite, une difficulté qui ne facilite pas non plus le processus est le fait que nous travaillons sur des documents anciens. Comme le précise [Chazalon, 2021] ou [Kusetogullari et al., 2020], des dégradations peuvent être dues au temps et rendre la lecture d'une carte difficile. Il peut y avoir une dégradation du papier, une bavure ou un effacement partiel de l'encre ou encore une partie du document qui est manquante. De plus, les documents anciens n'ont pas forcément les mêmes caractéristiques comme la texture de papier ou le style d'écriture manuscrite. Ce sont tous ces éléments qui rendent difficile l'automatisation complète du processus de reconnaissance d'objets au sein des documents anciens.

I.2 Généralités sur le deep learning

Le *deep learning* est une des méthodes de l'apprentissage automatique. Comme le montre la Figure 2, la grande différence entre la plupart des algorithmes de *machine learning* et de *deep learning* concerne l'extraction des caractéristiques. En effet, pour que l'apprentissage automatique fonctionne, il est nécessaire que l'algorithme ait connaissance des caractéristiques qui composent les données. Ces caractéristiques doivent être fournies en entrée de l'algorithme sous forme de cartes des caractéristiques (ou *feature map* en anglais). Une *feature map* est la sortie obtenue après avoir passé une donnée au travers d'un filtre. Il s'agit donc d'un traitement de la donnée, souvent utilisé en début de processus afin qu'elle soit au bon format et qu'elle contienne les éléments essentiels. Cependant, cette partie d'extraction des caractéristiques n'est pas réalisée automatiquement par les algorithmes de *machine learning* mais a été incluse dans le processus du *deep learning*.

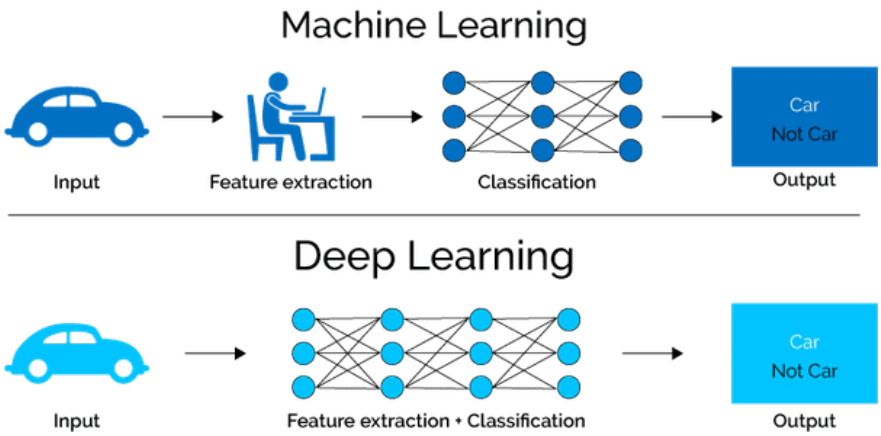

Figure 2 : Différence entre machine learning et deep learning Source : Julie LORENZINI, Le coin du digital, [en ligne]. Disponible sur : https://www.le-coin-du-digital.com/index.php/2018/07/11/deep-learning-vs-machine-learning/ (consulté le 14 juin 2021).

Les concepts introductifs qui régissent le *deep learning* ont vu le jour dans les années 50 avec la création du Perceptron, qui est le premier neurone artificiel créé. Cependant, ces concepts n'ont pas été exploités à cette époque à cause d'un manque de moyen pour faire fonctionner les réseaux profonds. Ce ne sont que dans les années 1980 que reprennent les travaux sur le *deep learning* et notamment celui de Yann LeCun. C'est en 1989 que [LeCun et al., 1989] adaptent les techniques de réseaux de neurones profonds à la reconnaissance de chiffres manuscrits. Ce projet avait été mis en place pour reconnaitre automatiquement les adresses pour le service postal des États-Unis. Ainsi, le premier réseau de *deep learning* nommé LeNet a été utilisé dans les années 1990. Depuis, de nombreux nouveaux réseaux ont vu le jour avec différentes architectures pour traiter toutes sortes de problèmes.

I.2.1 Fonctionnement d'un réseau de neurones

Revenons sur le fonctionnement d'un réseau de neurones comme nous l'entendons aujourd'hui. Tout d'abord, il faut savoir que la mise en place d'un réseau de *deep learning* se fait en deux temps : une phase d'entraînement du modèle et une phase de test. La phase d'entraînement correspond au moment où le modèle apprend. Pour cela, nous devons lui fournir des données, par exemple des images, auxquelles sont associées des étiquettes. Une étiquette contient les informations que doit apprendre le modèle quand il observe la donnée associée à celle-ci. Puis, une fois que l'apprentissage du réseau est jugé correct sur les données d'entraînement, nous pouvons tester le modèle. La phase de test consiste à montrer des nouvelles données au réseau pour qu'il fasse des prédictions de résultats sur celles-ci. Par exemple, dans le cas d'une classification binaire d'image où l'on souhaite savoir si une image contient un chien ou non, le test renverra la valeur 1 pour une image qui contient un

chien et la valeur 0 sinon. Cette phase de test permet d'analyser ses performances et d'évaluer la robustesse du réseau face à des données encore jamais rencontrées.

[Chollet, 2020] explique qu'un programme de *deep learning* est un réseau composé de plusieurs couches de neurones artificiels, dont le but est d'effectuer une tâche demandée. Chaque couche possède des paramètres qui modifient la donnée et la transmette à la couche suivante. Ces paramètres sont appelés poids et biais, et ce sont eux qui vont déterminer ce que va apprendre le modèle. Il existe différents types de couches qui n'auront pas le même effet sur la donnée. En fonction du choix des couches et de leur agencement, nous obtenons différentes architectures de réseaux.

Dans le but de trouver les bons paramètres pour toutes les couches, les données d'entrée passent plusieurs fois dans le réseau. Elle passe autant de fois que nécessaire pour que le modèle voit le jeu de données en entier à chaque époque. Une époque c'est quand un jeu de données est entièrement passer au travers d'un réseau une fois et que tous les poids ont été mis à jour une fois. Autrement, cela correspond au processus de la Figure 3 effectué une fois. Ainsi, plus le nombre d'époques est grand, plus le modèle aura vu le jeu de données dans sa totalité et aura ajusté ces paramètres. Cependant, pour que le modèle apprenne correctement, il faut qu'il puisse comparer ses résultats avec ceux attendus. Pour cela, l'opérateur définit une fonction de perte qui permet d'évaluer si l'algorithme est loin du résultat ou non. Cette fonction de perte renvoie un score qui traduit la distance entre la prédiction du modèle et le vrai résultat. De plus, afin que le modèle converge vers la solution, il est complété par un optimiseur dont la fonctionnalité est de modifier les poids et biais des couches en fonction du score de perte. L'objectif est donc de faire diminuer ce score au fur et à mesure des itérations. La Figure 3 résume ce processus de mise à jour des poids.

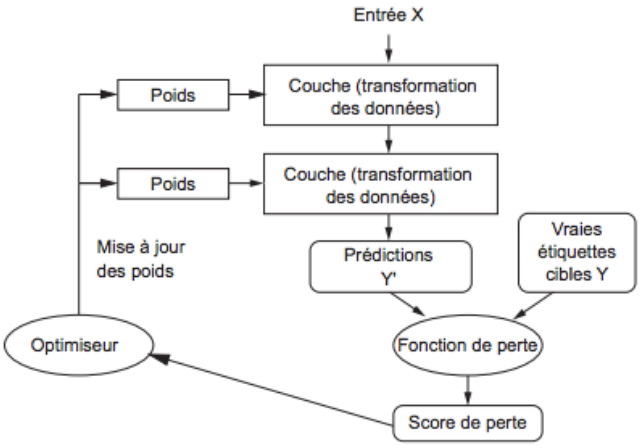

Figure 3 : Mécanisme de fonctionnement d'un réseau de neurones
Source : [Chollet F., 2020]

Pour connaitre le nombre de mise à jour des poids que doit faire le modèle à chaque époque, nous devons définir un paramètre supplémentaire appelé le *batch-size* ou taille de lot en français. Il correspond au nombre d'échantillons présents dans un lot. Donc plus la taille du lot est petite, moins il y a d'échantillons dans un lot et les poids sont mis à jour plus souvent. Cela implique aussi que si une donnée aberrante est présente dans les échantillons d'entraînement, elle aura un fort impact sur l'apprentissage.

Les réseaux de deep learning peuvent être utilisés dans différents domaines et notamment dans celui de la classification et de la régression. Il existe trois grandes familles de problème :

- La classification binaire : comme son nom l'indique, il s'agit de classer des données en deux groupes. En général, il s'agit de dire si la donnée appartient ou non à une catégorie. Pour cela, le modèle calcule un score qui, s'il est supérieur à un seuil fixé, est considéré comme positif et la donnée est donc comptée comme appartenant à la classe.
- La classification multi-classe : ici, il s'agit de classer des données dans un nombre de classes supérieur à 2. Pour que le modèle fasse une prédiction, il calcule un score, en général compris entre 0 et 1, pour chacune des classes. La prédiction renvoyée par le modèle est la classe qui a obtenu le score le plus élevé.
- La régression : c'est un type de problème qui consiste à prédire une valeur numérique. Par exemple, quelle sera la température de demain en se basant sur des données météorologiques ? Ou à quel prix se vendra une maison en fonction de son état et de sa localisation ? L'évaluation de ces modèles repose sur des métriques comme l'erreur quadratique moyenne.

Maintenant que le principe d'élaboration d'un réseau de neurones est plus clair, nous pouvons passer au détail des couches, des fonctions et des paramètres utilisés dans un réseau de *deep learning*.

I.2.1.1 Fonctionnement d'un neurone

Tout d'abord, voyons quel est le fonctionnement d'un neurone. Comme l'explique [Patterson et al., 2018] dans leur ouvrage, un neurone transforme les données d'entrée en une valeur comprise, en général, entre 0 et 1. Pour cela, le neurone procède à la moyenne des valeurs d'entrée (ou les valeurs de la couche des neurones précédents, a_i) pondérée par

leurs poids (p_i). À cette moyenne est ajouté un biais (b), avant qu'elle soit injectée dans une fonction d'activation (voir équation n°1). Ce nom de fonction a été choisi car plus la valeur de sortie du neurone est proche de 1 et plus les poids de ce neurone seront pris en compte dans le calcul, donc le neurone sera actif.

$$sortie\ du\ neurone = fonction\ d'activation\left(\sum_{i=1}^{n} p_i \times a_i + b\right) \quad (1)$$

Il existe un certain nombre de fonctions d'activation parmi lesquelles figurent la fonction unitaire de rectification linéaire (ReLU), la sigmoïde ou encore la tangente hyperbolique (voir Figure 4 et équation n°2 à 4). Le choix de la fonction d'activation se fait en fonction du problème à résoudre et du nombre de sorties souhaitées.

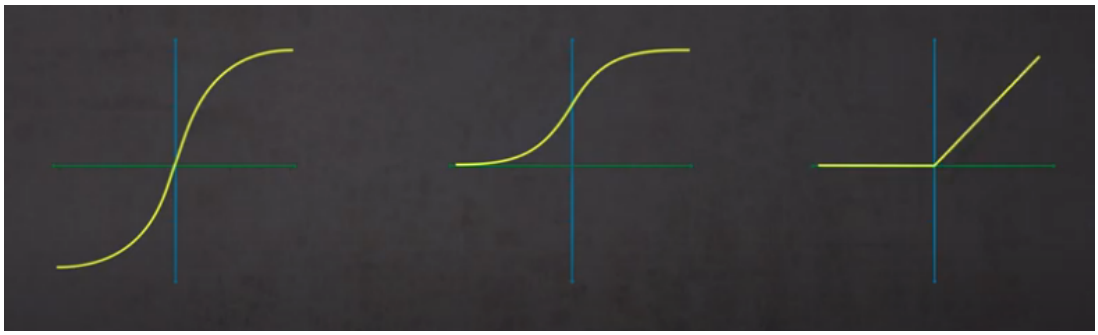

Figure 4 : À gauche : Fonction de tangente hyperbolique. Au centre : Fonction sigmoïde. À droite : Fonction ReLU
Source : Vidéo Youtube de la chaîne Chronophage. URL : https://www.youtube.com/watch?v=QuMabWInlAQ (consulté le 17 mars 2021)

$$tangente\ hyperbolique = \tanh(x) = \frac{\sinh(x)}{\cosh(x)} \quad (2)$$

$$sigmoïde(x) = \frac{1}{1 + e^{-x}} \quad (3)$$

$$ReLU(x) = \max(0, x) \quad (4)$$

De plus, le choix de la fonction d'activation de la dernière couche du réseau est important puisqu'en fonction de celle-ci la sortie sera différente. Par exemple, pour une classification multi-classe, nous choisirions la fonction *softmax* afin d'avoir un score de probabilité compris entre 0 et 1 pour toutes les classes.

I.2.1.2 La fonction de perte

Ensuite, comme autre composante importante dans la construction d'un modèle, il y a la fonction de perte. Elle calcule un score correspondant à la distance entre la prédiction

faite par le modèle et le résultat attendu. En fonction du choix de celle-ci, l'apprentissage sera différent puisque son objectif est de faire diminuer la valeur de la fonction de perte. Il en existe un grand nombre et il est aussi possible de définir sa propre fonction de perte. Parmi les plus connues, il y a la fonction d'entropie croisée binaire (*binary crossentropy*), l'entropie croisée catégorique (*categorical crossentropy*), l'erreur quadratique moyenne (MSE) etc….

[Chollet, 2020] conseille de choisir la fonction d'activation et celle de perte en fonction du problème à résoudre. Le Tableau 1 résume ses propositions.

Type de problème	Activation de la dernière couche	Fonction de perte
Classification binaire	Sigmoïd	Binary_crossentropy
Classification à plusieurs classes avec prédiction d'une seule étiquette	Softmax	Categorical_crossentropy
Classification à plusieurs classes avec prédiction de plusieurs étiquettes	Sigmoïd	Binary_crossentropy
Régression de valeurs arbitraires	Aucune activation	MSE
Régression de valeurs comprises entre 0 et 1	Sigmoïd	MSE ou Categorical_crossentropy

Tableau 1 : Choix de la fonction d'activation de la dernière couche et de perte en fonction du problème
Source : [Chollet, 2020]

I.2.1.3 Les métriques de performance

Pendant la phase d'entraînement, mais aussi pendant celle de test, d'autres valeurs peuvent être surveillées pour évaluer les performances du modèle. En effet, la fonction de perte permet de faire avancer l'entraînement dans le bon sens cependant, comme elle converge vers zéro, elle ne traduit pas bien la notion de performance. C'est pourquoi, il est possible de calculer une ou plusieurs valeurs à chaque itération comme la précision (équation n°5) ou le rappel (équation n°6) qui sont calculées pour chaque classe d'un problème et dont nous pouvons faire la moyenne pour obtenir la précision ou le rappel global. Dans le cas de la détection d'objet, il existe les métriques comme *l'intersection over union* (IoU, voir Figure 5), le *mean Average Precision* (mAP) ou encore dans le cas du Faster R-CNN le *mean overlapping bboxes*.

$$précision = \frac{Vrai\ positif}{Vrai\ positif + Faux\ positif} \quad (5)$$

$$rappel = \frac{Vrai\ positif}{Vrai\ positif + Faux\ négatif} \quad (6)$$

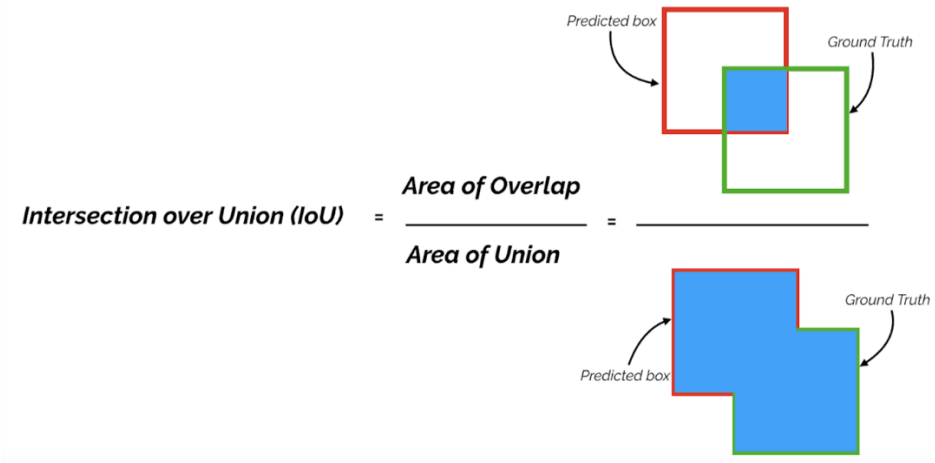

Figure 5 : Formule de l'IoU
Source : [Yohanandan S, 2020]

[Yohanandan, 2020] explique que le mAP est une métrique un peu plus complexe puisqu'il s'agit de la moyenne de l'AP. Et l'AP est calculée pour chaque classe et correspond à l'aire présente sous la courbe de précision-rappel.

Quant à lui, le *mean overlapping bboxes* est l'une des métriques de performance présentes dans l'architecture Faster R-CNN. Elle correspond au nombre moyen de boîtes englobantes correctement détectées dans une image. Ainsi, pour chaque image, le modèle compte le nombre de boîtes englobantes détectées dont le score IoU est supérieur au seuil fixé et une fois l'itération terminée, il calcule la moyenne. Cette valeur doit donc se rapprocher du nombre moyen de boîtes par image présentes dans le jeu de données.

I.2.1.4 L'optimiseur

Enfin, le dernier élément important d'une architecture est l'optimiseur. D'après [Doshi, 2019] « Les optimiseurs sont des algorithmes ou des méthodes utilisés pour modifier les attributs de votre réseau de neurones tels que les poids […] afin de réduire les pertes ». Ici aussi, il en existe un grand nombre dont le choix dépendra de la capacité des machines utilisées et de l'objectif à atteindre. Parmi eux, [Patterson et al., 2018] détaille notamment le fonctionnement de la méthode de la descente de gradient stochastique (SGD), le RMSProp et la méthode Adam. Le choix de l'optimiseur repose sur le taux d'apprentissage qui est un paramètre à déterminer en amont pour certains optimiseurs et qui est automatiquement déterminé pour d'autres. Il repose aussi sur la composition du jeu de données, à savoir si

elles ont une distribution hétérogènes ou non, ainsi que sur la capacité de l'algorithme à converger rapidement vers la solution sans être « coincé » dans un minima local.

I.2.2 Quelques problèmes du deep learning

Maintenant que la composition d'un modèle est définie, nous allons expliquer ce que sont les risques de surentraînement ou de sous-entraînement ainsi que le problème de manque de données.

Comment savoir si le modèle est correctement entraîné ? Dans un premier temps, nous pouvons suivre l'évolution des métriques de performances. Par exemple, nous pouvons exécuter le nombre d'époques suffisant pour que la précision du modèle dépasse 90% lors de la phase d'entraînement. Néanmoins, il peut arriver que le modèle ne soit pas performant lors de la phase de test malgré cette précision. En effet, comme l'explique [Chollet, 2020] cette imprécision sur des nouvelles données se justifie par un sur-apprentissage du modèle sur les données d'entraînement. Cela signifie que le modèle a appris par cœur les caractéristiques des données d'entraînement, au lieu de trouver celles qui sont communes à toutes les données, et d'en faire une généralisation. Ainsi, lorsque le modèle est testé sur des nouvelles données, il n'est pas capable d'atteindre une précision équivalente à celle de la phase d'entraînement. C'est pourquoi, il est important de trouver l'équilibre entre la généralisation et l'optimisation du modèle.

Ensuite, un mauvais entraînement peut-être causé par un manque de données d'apprentissage. Si l'ensemble de données est trop petit et trop peu représentatif, le modèle peut ne pas apprendre correctement la tâche qui lui a été demandée. Pour pallier à ce manque de données, il existe la méthode de la *data augmentation* (ou augmentation de données en français). [Chollet, 2020] explique qu'il s'agit de traitements aléatoires effectués pour déformer les données en utilisant des déformations horizontales, verticales et des rotations. Cela permet donc de créer des données supplémentaires en se basant sur celles existantes.

I.3 Les réseaux de neurones

Afin de mettre en place un modèle de *deep learning*, il faut d'abord choisir une architecture. L'architecture d'un réseau désigne la façon dont les couches de neurones sont agencées les unes par rapport aux autres. Le choix de l'architecture est surtout fait en fonction des données qui seront utilisées en entrée de ce réseau. Le Tableau 2 résume les trois grandes familles d'architectures à utiliser en fonction du type de données d'entrée, sur les conseils de [Chollet, 2020].

Type de données d'entrée	Réseaux entièrement connectés	Réseaux convolutifs (CNN)	Réseaux récurrents (RNN)
Vectoriel (tenseur 2D)	X		
Séries temporelles (tenseur 3D)		X	X
Image couleur (tenseur 4D)		X	
Vidéo (tenseur 5D)		X	X

Tableau 2 : Catégorie de réseau de neurones à utiliser en fonction du type de données d'entrée[1]

I.3.1 Les réseaux entièrement connectés

Ce type de réseaux ne traite que les données vectorielles et n'effectue pas de réelles hypothèses sur l'organisation des caractéristiques. Pour chaque couche de ces réseaux, tous les neurones sont connectés aux neurones de la couche suivante. Cela entraîne donc un nombre de paramètres important à calculer pour le modèle. L'avantage de ces réseaux est qu'ils sont facilement applicables à différents types de problème, cependant leurs performances, notamment de temps de calculs, peuvent être moins bonnes que celles d'autres architectures de réseaux.

I.3.2 Les réseaux récurrents (RNN)

De manière générale, lorsqu'un réseau analyse une donnée, il la traite dans son ensemble et en une seule fois. Il ne garde pas en mémoire une partie de la donnée qui lui permettrait de comprendre la suite de celle-ci. Cependant, les RNN permettent de garder en mémoire ce que le modèle a vu. En effet, [Chollet, 2020] explique que ces réseaux traitent des séquences de données « en parcourant les éléments de la séquence et en maintenant un état contenant des informations relatives à ce qu'il a vu jusqu'à présent ». Ce fonctionnement est mis en application en créant des neurones qui interagissent de façon non linéaires les uns avec les autres. Cela permet de créer des réseaux qui traitent des séries temporelles de données.

I.4 Les réseaux de neurones convolutifs (CNN)

Étant donné que nous travaillons avec des données images, nous allons utiliser un réseau CNN comme dans [Chollet, 2020].

[1] Données du tableau prises dans le livre de [Chollet, 2020]

I.4.1 La convolution dans les réseaux de deep learning

Les réseaux CNN, aussi appelés ConvNet, sont composés d'une succession de couches de convolution et de couches de *pooling* (regroupement en français). Nous allons détailler le rôle de ces deux types de couches afin de comprendre la spécificité des réseaux de neurones convolutifs.

Tout d'abord, [Saha, 2018] indique que « L'objectif de l'opération de convolution est d'extraire les caractéristiques de haut niveau ». En effet, l'objectif principal d'un CNN est d'extraire les caractéristiques de la donnée afin d'en faire une généralisation et de les apprendre. Pour ce faire, les couches de convolution sont utilisées afin d'obtenir en sortie une *feature map* (carte de caractéristiques). Les couches de convolutions possèdent plusieurs paramètres qui modifient le résultat que nous pouvons obtenir en sortie. Les paramètres sont :

- La taille de la donnée d'entrée.
- La taille du noyau. En général, elle est de 3x3 ou de 5x5.
- Le pas (ou *stride*) : c'est la distance de déplacement du filtre entre deux positions.
- Le *zero-padding* : c'est le nombre de zéro ajoutés en extrémité des lignes et des colonnes de la donnée d'entrée. Cela permet d'obtenir une *feature map* de même dimension que la donnée d'entrée. Voir un exemple d'application Figure 6.

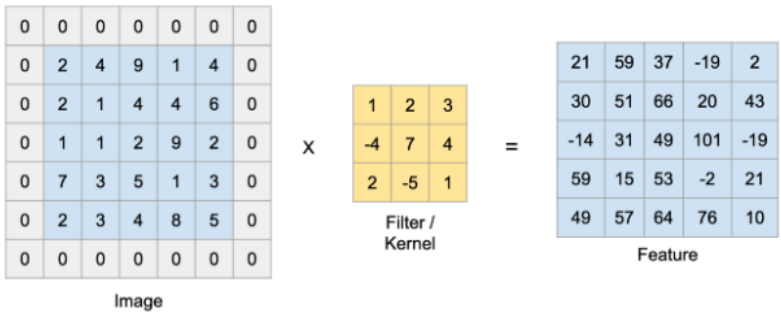

Figure 6 : Zero-padding appliqué à une image de taille 5x5 et un noyau 3x3 avec un pas de 1. Source : Patel, *Convolutionnal Neural Network - A Beginner's Guide sur toward data science*, 2019, towards data science [en ligne], https://towardsdatascience.com/convolution-neural-networks-a-beginners-guide-implementing-a-mnist-hand-written-digit-8aa60330d022 (consulté le 21 juin 2021).

De plus, une couche de convolution est souvent suivie d'une couche de *max pooling* ou *average pooling*, ou en français les couches de regroupement maximum ou moyen. Celles-ci permettent de réduire drastiquement la taille de la *feature map* et par conséquent

de réduire le nombre de paramètres calculés par le modèle. Ces couches servent aussi à détecter des caractéristiques à différentes échelles. [Dumoulin et al., 2018] explique qu'elles fonctionnent de la même manière que les couches de convolution. Cependant, au lieu d'utiliser « une combinaison linéaire décrite par le noyau », elles appliquent un sous échantillonnage par agrégation moyenne ou maximale (voir Figure 7).

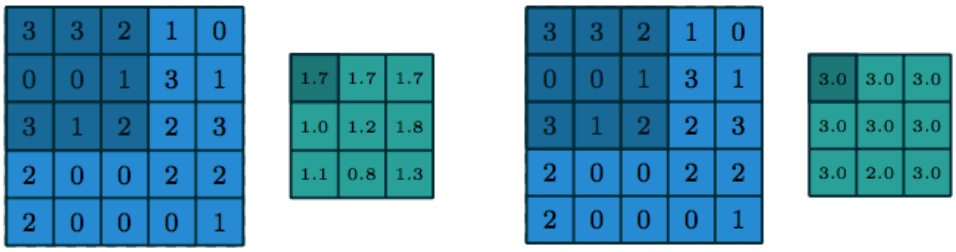

Figure 7 : À gauche : Average pooling. À droite : Max pooling, appliqué au coin haut gauche de la donnée
Source : [Dumoulin et al., 2018]

I.4.2 CNN basé sur les régions (R-CNN)

Les réseaux CNN sont connus pour faire de la classification d'images. Or, la classification d'images ne permet pas de trouver un ou plusieurs objets dans l'image et de leur attribuer à chacun une classe. En effet, [Gandhi, 2018] précise que les algorithmes CNN ne peuvent pas être directement appliqués pour faire de la détection d'objet à cause du nombre inconnu d'objets à trouver dans l'image. Par exemple, il peut y avoir plusieurs chats et un chien dans une image et un seul chat dans une autre. Cela pose problème pour les paramètres de la couche de sortie du modèle. Le principe qui permet de détecter plusieurs objets dans une image sont les régions d'intérêts. Ce sont des zones délimitées par un carré ou un rectangle, dans lesquelles pourraient figurer un objet que nous recherchons.

Afin de remédier à cela, [Girshick et al., 2014] ont mis au point le réseau R-CNN. Dans un premier temps, celui-ci consiste à appliquer une recherche sélective sur l'image. La recherche sélective a été mise en place par [R. R. Uijlings et al., 2013] en combinant les avantages de la recherche exhaustive et de la segmentation, et cela permet au réseau R-CNN de trouver 2 000 régions d'intérêt (RoI) de dimensions différentes. Le nombre de régions d'intérêt a été fixé par [Girshick et al., 2014]. Ces régions d'intérêts permettent d'avoir des boîtes candidates pouvant contenir un objet recherché.

Les régions d'intérêt sont ensuite fournies à un réseau CNN afin d'en extraire les caractéristiques. Puis, l'algorithme de classification SVM (Support Vector Machine) est

utilisé pour attribuer une classe à chaque région. Enfin, une régression des boîtes englobant l'objet est effectuée dans le but d'améliorer les contours de la région proposée.

[Gandhi, 2018] soulève plusieurs inconvénients de cette méthode. Tout d'abord, cette méthode nécessite l'utilisation de quatre modèles pour aboutir : la recherche sélective, un CNN, un algorithme de classification SVM et la régression des boîtes englobantes. De plus, le traitement de 2 000 régions d'intérêt pour chaque image engendre un temps de calcul assez conséquent. Et enfin, la qualité du résultat dépend fortement des régions proposées par la recherche sélective, or étant donné qu'elle ne fait pas partie du réseaux CNN, aucun apprentissage ne peut être fait. Ainsi, le CNN, l'algorithme SVM et la régression peuvent s'améliorer mais si les propositions de boîtes ne sont jamais correctes, le R-CNN n'aura jamais de bons résultats.

I.4.3 Fast R-CNN

Partant de cette première tentative pour mettre en place un réseau de détection et de reconnaissance d'objet à partir d'un CNN, le même auteur a voulu résoudre les principaux problèmes de la méthode R-CNN. Comme l'explique [Girshick, 2015], le nouveau modèle nommé Fast R-CNN (ou R-CNN rapide en français) est neuf fois plus rapide que le R-CNN lors de la phase d'entraînement et 213 fois lors de la phase de test. Cette progression est notamment due aux modifications apportées au début du processus.

En effet, au lieu de sélectionner 2 000 régions et de toutes les traiter au sein d'un CNN, le Fast R-CNN applique un réseau convolutif directement sur l'image entière pour en extraire une *feature map*. Puis, la recherche sélective est appliquée sur cette *feature map* pour trouver des régions d'intérêt. Enfin, pour chaque région, le modèle prédit une boîte englobante ainsi que la probabilité d'appartenance de cette boîte à une classe.

Cette méthode est plus performante que la précédente, cependant elle fait toujours appelle à la recherche sélective qui, d'après [Gandhi, 2018], est lente et fastidieuse.

I.4.4 Faster R-CNN

Jusqu'ici les améliorations apportées ont permis d'accélérer le processus de reconnaissance des objets, mais cela n'est toujours pas suffisant pour traiter des images en temps réel. En effet, si nous souhaitons faire de la reconnaissance d'objet sur des vidéos de surveillance par exemple, il faut que le traitement soit plus rapide que la fréquence d'acquisition des images.

Pour cela, [Ren et al., 2016] proposent une nouvelle approche. Celle-ci consiste à remplacer l'algorithme de recherche sélective par un réseau entièrement convolutif. Ce réseau est appelé *Region Proposals Network* (RPN), littéralement, réseau de proposition de régions en français. Ensuite, après une opération de *RoIPooling*, [Ren et al., 2016] utilisent un Fast R-CNN qui reprend les propositions de régions du RPN comme données d'entrée pour trouver les boîtes englobantes et leur probabilité d'appartenance à une classe (voir Figure 8).

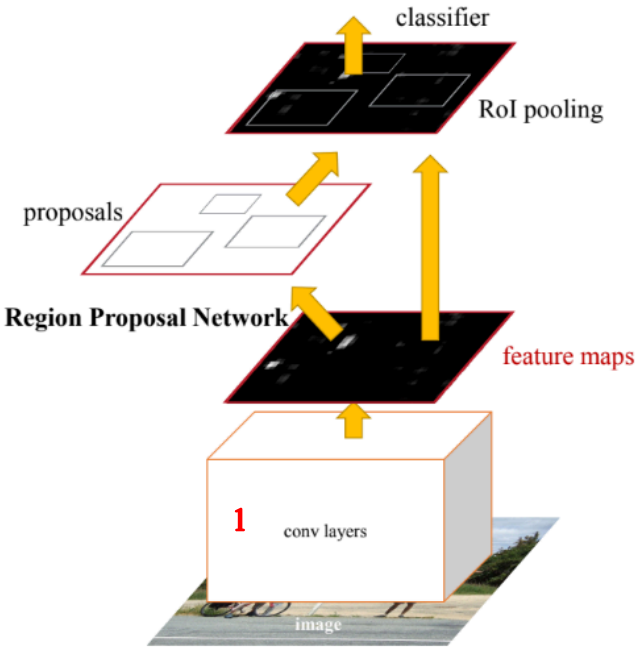

Figure 8 : Fonctionnement du Faster R-CNN
Source : [Ren et al., 2016]

D'après [Sambasivarao, 2019], la couche de *RoIPooling* permet de mettre en commun toutes les régions d'intérêt trouvées via une opération de *max pooling*. Cela permet de traiter tous les RoI en une seule fois et de produire une *feature map* de taille fixe à partir de données non uniformes. Elle prend donc en entrée deux choses : la *feature map* retournée par le réseau CNN et les RoI détectées par le RPN. Ensuite, pour chaque RoI, cette couche prend la section de la *feature map* correspondante au RoI et divise cette section en bloc en fonction des paramètres renseignées dans la couche. Puis, sur chaque bloc est appliquée l'opération de *max pooling*, ce qui renvoie une *feature map* de sortie de taille fixe. (voir les opérations à la Figure 9).

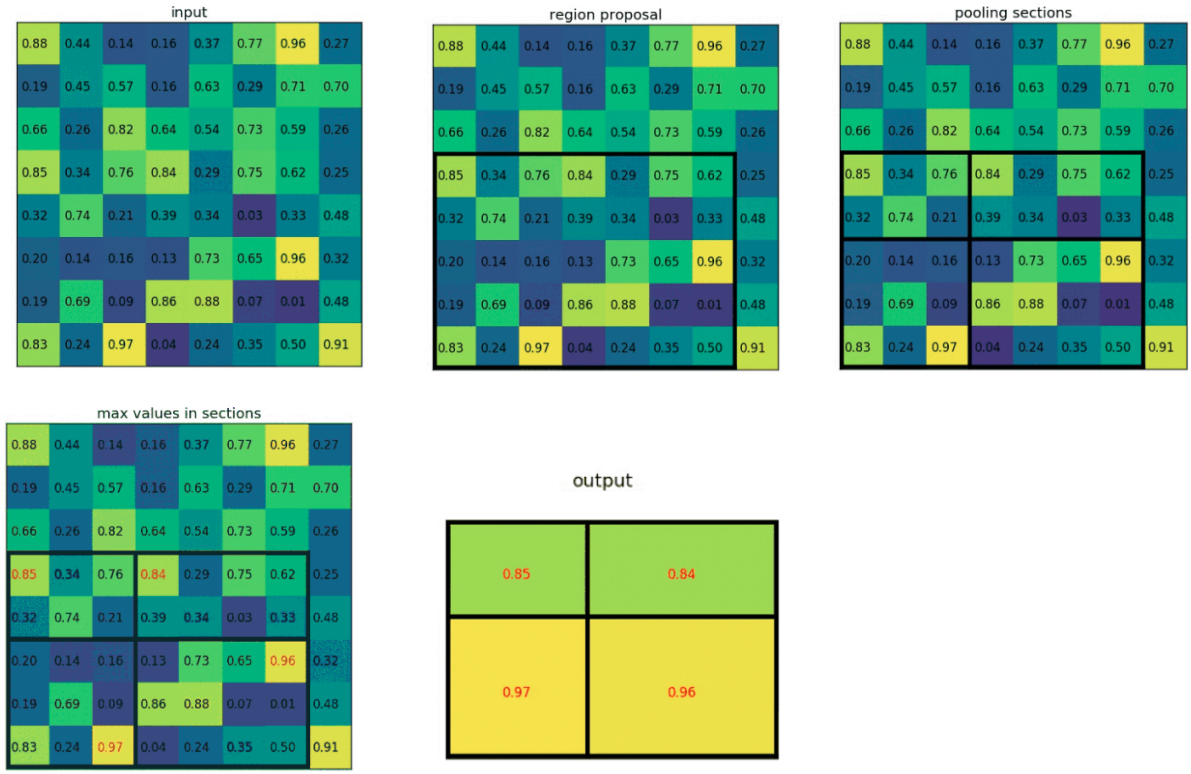

Figure 9 : Opération de la couche de *RoIPooling*
Source : [Sambasivarao K., 2019]

I.4.4.1 Le Region Proposal Network (RPN)

Le RPN est la partie chargée de la recherche des propositions de régions, pour ensuite les fournir au Fast R-CNN dans le but de les classer. Étant donné que les deux réseaux sont utilisés conjointement, [Ren et al., 2016] ont choisi d'utiliser un réseau de couches convolutives partagées (voir 1 Figure 8). Cela permet de mettre en commun les couches similaires entre les deux modèles et de les entraîner simultanément.

Une fois que la *feature map* est générée par la dernière couche du réseau partagé, le RPN fait glisser une fenêtre pour appliquer un mini-réseau sur les données contenues dans cette fenêtre. L'objectif de ce mini-réseau est de déterminer d'une part les coordonnées de la boîte englobante, c'est-à-dire le coin haut gauche et bas de droit de la boîte, et d'autre part deux scores qui estiment la probabilité que la boîte proposée contienne un objet ou non. Le mini-réseau est composé d'une première couche convolutive dont les dimensions correspondent à la taille de la fenêtre. Puis cette couche est suivie de deux autres couches convolutives de taille 1x1, une pour chaque partie à déterminer à savoir : les coordonnées de la boîte et les scores de probabilité.

Le principe novateur qui caractérise le RPN est l'utilisation des ancres. En effet, pour chaque fenêtre examinée par le modèle, un certain nombre d'ancres sont calculées. Les

ancres sont des propositions de boîtes faites pour chaque position de la fenêtre et dont l'échelle et le rapport largeur/hauteur ont été prédéfinis. Par exemple, si nous choisissons de rechercher des boîtes de taille 50x50 pixels et de 50x100 pixels, nous indiquerons une échelle d'ancre de 50 et deux ratios largeur/hauteur de 1/1 et 1/2. Cela donne donc deux ancres pour chaque fenêtre. Dans le cas de l'étude de [Ren et al., 2016], il a été choisi trois échelles et trois rapport largeur/hauteur, ce qui fait un total de neuf ancres pour chaque fenêtre. Le paramétrage des ancres permet au modèle d'apprendre à détecter des objets de différentes taille et proportions pré-définies, chose qui n'était pas possible avant.

I.4.4.2 Entraînement du Faster R-CNN

Étant donné que le Faster R-CNN est composé de deux modèles unifiés grâce aux couches convolutives partagées, l'entraînement ne peut pas être fait de façon classique.

Avant de s'intéresser à l'entraînement du Faster R-CNN, nous allons nous intéresser à la notion de modèle pré-entraîné. En effet, [Marcelino, 2018] explique qu'un modèle pré-entraîné est un modèle déjà mis en place dans la littérature, préalablement entraîné pour une tâche similaire à notre problème. Les modèles pré-entraînés ont été formés sur des grands ensemble de données afin de permettre à des utilisateurs de reprendre les poids de ces modèles sans avoir à les ré-entraîner, ce qui est un réel gain de temps et de ressources informatiques. Ces poids sont utilisés comme base de départ pour la formation du notre propre modèle.

La méthode de formation du modèle choisie par [Ren et al., 2016] est celle de l'entraînement en alternance en quatre étapes (*4-step alternating training*). Celles-ci consistent à :

- Entraîner le RPN seul. Celui-ci utilise les poids d'un modèle pré-entraîné sur le jeu de données ImageNet[2]. [Ren et al., 2016] se sont servis du modèle VGG16 et du modèle mis en place par [Zeiler et al., 2014].
- Entraîner le Fast R-CNN seul en utilisant les régions trouvées précédemment par le RPN comme données d'entrée. Il est aussi initialisé avec les poids du modèle pré-entraîné sur ImageNet.

[2] Le dataset ImageNet est un jeu de données qui contient environ 14 millions d'images classées dans plus de 20 000 catégories. Les catégories peuvent être « chiens », « ballons », « fraises » etc…

- Conserver les couches partagées du Fast R-CNN. Ces couches sont fixées afin que leurs poids ne se mettent pas à jour. Puis, les poids du RPN sont ajustés pour s'adapter aux résultats obtenus en sortie du Fast R-CNN.
- Affiner les poids des couches du Fast R-CNN, toujours en gardant fixe les poids des couches partagées.

L'architecture VGG16 est un réseau convolutif mis en place par [Simonyan et al., 2015]. Il est plus performant que le réseau AlexNet car au lieu d'avoir des couches avec des grands filtres, il a plus de couches (donc le réseau est plus profond) mais les filtres sont plus petits (voir Figure 10). Ce réseau a notamment remporté la première et la deuxième place du challenge de reconnaissance visuelle à grande échelle ImageNet (ImageNet Large-Scale Visual Recognition Challenge = ILSVRC).

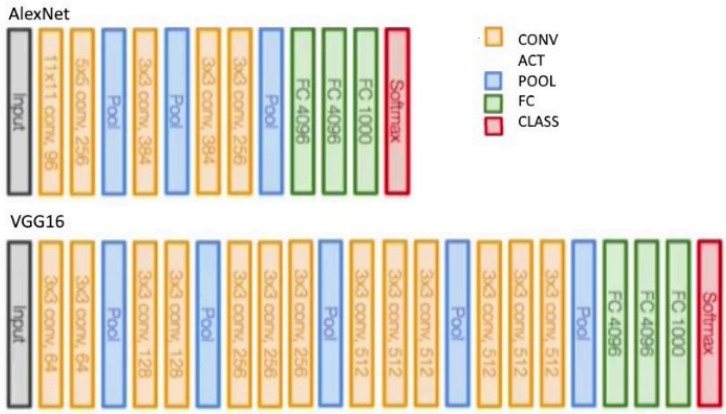

Figure 10 : Comparaison entre les architectures AlexNet et VGG16
Source : A. Lumini et L. Nanni, Ocean Ecosystems Plankton Classification: Theories and Applications, 2019, Research Gate [en ligne], 10.1007/978-3-030-03000-1_11. https://www.researchgate.net/publication/329685943_Ocean_Ecosystems_Plankton_Classification_Theories_and_Applications (consulté le 1er juillet 2021).

Le Tableau 3 récapitule les architectures rencontrées et leurs spécificités.

Architectures	Articles	Composition	Avantages	Inconvénients
R-CNN	[Girshick et al., 2014]	Recherche sélective, CNN, classification SVM, régression de boîte	Permet la détection et la reconnaissance d'objet.	Architecture lente à cause des 2000 RoI par image. Aucun apprentissage possible lors des propositions de RoI.
Fast R-CNN	[Girshick, 2015]	CNN pour feature map, recherche sélective, CNN, régression de boîte	Proposition de région incluses dans l'apprentissage. Gain de temps d'entraînement et de test.	Présence de la recherche sélective qui ralentit le processus.

Faster R-CNN	[Ren et al., 2016]	CNN partagé, RPN, CNN, régression des boîtes.	Utilisation des ancres accélère le temps de calcul.	Entraînement en 4 étapes imbriquées donc dépendance de ces étapes entre elles.

Tableau 3 : Synthèse des architectures R-CNN

I.5 Les travaux existants dans le domaine de la reconnaissance de caractères

La reconnaissance de caractères numériques écrits à la main est un des premiers cas traité par les réseaux de neurones convolutifs. [LeCun et al., 1989] a traité le cas de la reconnaissance des chiffres pour le service postal des États-Unis (voir l'introduction de la partie I.2). Nous savons aussi que le jeu de données a une importance capitale pour l'élaboration d'un modèle *deep learning* autant lors de la phase d'entraînement que de test. Avec l'apparition de nouveaux jeux de données de chiffres manuscrits, la mise en place de réseaux de neurones pour les reconnaitre a pris de l'ampleur et notamment avec l'organisation de compétitions comme celles du site Kaggle[3] par exemple.

I.5.1 Les jeux de données existants

Afin que le modèle ne soit ni surentraîné ni sous-entraîné, le jeu de données doit être adapté au problème. En effet, si un modèle est entrainé à reconnaitre des images de chats, il ne sera pas capable de reconnaitre une vache, sauf si nous utilisons un modèle pré-entraîné. Cela paraît logique mais cette notion peut se transposer à des niveaux plus fins. C'est pourquoi, il est préférable que le jeu de données d'entraînement soit proches des données qui seront utilisées en phase de test.

Ainsi, c'est dans cette optique que de nombreux jeux de données ont été créés et sont disponibles en open data. Ils sont en général téléchargeables sur Internet et peuvent être organisés de différentes façons. Parmi elles, les données peuvent être organisées dans des dossiers portant le nom de la classe ou encore, les images peuvent être en vrac et leurs étiquettes sont renseignées dans un fichier texte. Dans le cadre de la reconnaissance de chiffres manuscrits, plusieurs jeux de données ont été mis en place au fil du temps et peuvent tous présenter des avantages pour notre problématique.

I.5.1.1 MNIST

Le jeu de données Modified National Institue Standards and Technology database (MNIST) est un ensemble de 70 000 images collectées par le laboratoire américain NIST

[3] https://www.kaggle.com/

contenant chacune un chiffre manuscrit. Ces images sont codées en nuances de gris et ont une dimension de 28x28 pixels. Il y a 10 catégories de chiffres allant de 0 à 9. Ces 70 000 images sont subdivisées en deux ensemble : un ensemble d'entraînement dans lequel il y a 60 000 images et un ensemble de test contenant les 10 000 images restantes. À chacun de ces ensembles est associé un fichier répertoriant les labels pour chaque image.

Ce jeu de données a été créé par l'Institut national des normes et de la technologie. [MNIST] explique qu'il a été produit grâce à l'union de deux anciens jeux de données. En effet, à l'origine l'Institut avait mis en place plusieurs *special dataset* et le jeu de données MNIST est la combinaison du *special dataset 1* et du *special dataset 3*. A l'origine, le *special dataset 3* servait pour la phase d'entraînement et le *special dataset 1* pour la phase de test. Cependant, ces deux jeux de données étaient foncièrement différents car ils ne provenaient pas de la même source. Le premier a été produit par les employés de Census Bureau et le deuxième par des élèves du secondaires. Il en est de même pour l'ensemble de test. En Figure 11 quelques exemples d'images provenant du jeu de données MNIST.

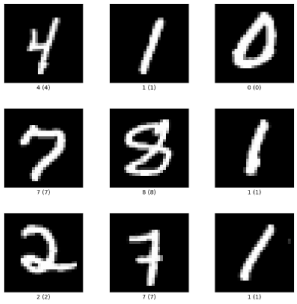

Figure 11 : Exemples d'images de chiffre du jeu de données MNIST
Source : https://www.tensorflow.org/datasets/catalog/mnist

I.5.1.2 ARDIS

Le jeu de données ARkiv Digital Sweden contient aussi des images de chiffres écrits à la main. [Kusetogullari et al., 2019] ont extrait ces chiffres de registres religieux suédois. Ces registres ont été écrits par différents prêtres du 19ème au 20ème siècles. De ce fait, l'écriture varie d'un registre à un autre et permet une plus grande représentabilité des chiffres. De plus, la calligraphie n'étant pas la même que celle d'aujourd'hui, ce jeu de données peut permettre une meilleure reconnaissance appliquée aux documents anciens.

Ce jeu de données est divisé en 4 ensembles dont le contenu est illustré à la Figure 12 :

- *Dataset* 1 : contient 10 000 images, chacune composée d'une chaîne de 4 chiffres (donc un nombre à 4 chiffres) directement extraite des documents, donc sans retraitement. Les images sont codées en RGB et au format JPG mais ne sont pas normalisées, c'est-à-dire qu'elles n'ont pas les mêmes dimensions. Ce jeu de données est lui-même divisé en 3 parties qui correspondent à des dates différentes.
- *Dataset* 2 : contient 7 600 images de chiffres non retraitées, c'est-à-dire directement extraites des documents. Elles sont aussi codées en RGB et au format JPG.
- *Dataset* 3 : contient 7 600 images de chiffres dont le fond a été nettoyé c'est-à-dire que les éventuelles bavures d'encres par exemple ont été enlevées.
- *Dataset* 4 : contient 7 600 images de chiffres répertoriées dans des fichiers au format .csv et sont en noir et blanc. Il y a 4 dossiers comportant les images d'entraînement et de tests ainsi que les labels correspondants. Elles ont été mises au même format que la bases de données MNIST afin de pouvoir les comparer.

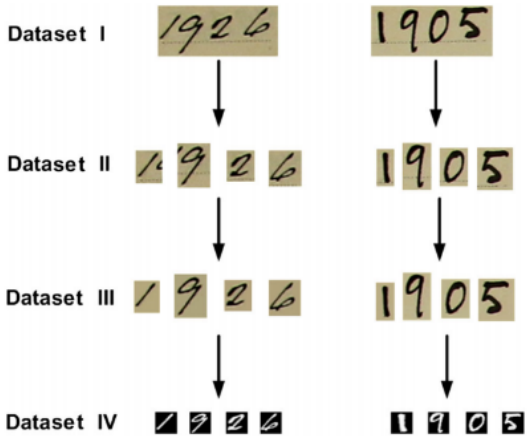

Figure 12 : Exemples d'images des 4 jeux de données de ARDIS
Source : [Kusetogullari et al., 2019]

I.5.1.3 USPS

Enfin, nous avons le jeu de données United-States Postal Service qui regroupe des chiffres mais aussi des lettres et des mots. C'est ce jeu de données qui a été utilisé par [LeCun et al., 1989] pour mettre au point l'architecture LeNet. Il contient un total de 9298 images réparties en deux ensembles : un ensemble d'entraînement qui contient 7291 images et un ensemble de test qui contient les 2007 images restantes. Les images sont codées en nuances de gris (voir quelques exemple Figure 13).

Figure 13 : Exemples d'images du dataset USPS
Source : Hull J.J., *A Database for Handwritten Text Recognition Research*, in IEEE *Transactions on Pattern Analysis and Machine Intelligence*, vol. 16, n°5, pp. 550-554, May 1994, doi: 10.1109/34.291440. [en ligne] https://ieeexplore.ieee.org/abstract/document/291440 (consulté le 22 mars 2021)

I.5.2 La méthode DIGITNET

Le travail de [Kusetogullari et al, 2020] est fortement lié avec le sujet de ce mémoire, puisque le sujet est la détection et la reconnaissance de chiffres et de nombres dans des images de documents anciens. Ces documents anciens sont datés entre 1800 et 1940, ce qui offre un large panel de documents et d'écritures différents. L'objectif est d'automatiser cette tâche avec de l'apprentissage profond.

I.5.2.1 Création d'un jeu de données adapté au problème

Afin de mettre en place leur projet, [Kusetogullari et al, 2020] ont décidé de créer leur propre jeu de données. Comme ils l'expliquent dans leur article, les jeux de données existants tel que MNIST ou USPS ne conviennent pas pour l'entraînement de leur modèle. En effet, ces données sont normalisées alors que la taille des caractères numériques n'est jamais fixe dans les documents anciens. De plus, les images sont en noir et blanc alors que les documents qui seront testés sont en couleur. Enfin, les chiffres de ces jeux de données ont été nettoyés de toutes dégradations, alors que le but ici est d'apprendre au modèle à s'adapter aux difficultés des documents anciens. Un jeu de données qui pourrait convenir serait le *dataset* 2 d'ARDIS, cependant [Kusetogullari et al, 2020] jugent qu'il ne contient pas un nombre suffisant de caractères pour entraîner un modèle.

C'est après cette analyse qu'ils ont décidé de créer leur propre jeu de données, nommé DIDA, qui est une extension de celui d'ARDIS. Une partie de ce jeu de données a été mise en ligne à partir du 14 juin 2021 ce qui a rendu impossible le test de ce jeu de données dans le temps imparti de ce TFE. Ce nouveau jeu de données est découpé en trois datasets :

- *Dataset* 1 : ce jeu contient 200 000 chiffres manuscrits répartis uniformément en classe allant de 0 à 9. Ils ont été extraits à partir de 100 000 documents historiques suédois.
- *Dataset* 2 : ce jeu-ci contient 200 000 images de chaînes de chiffres (donc des nombres). Ce jeu est lui-même divisé en deux sous-ensembles qui sont : un jeu qui est composé de 130 000 nombres de quatre chiffres uniquement et un jeu de 70 000 nombres d'une longueur comprise entre deux et dix chiffres.
- *Dataset* 3 : celui-ci est composé d'images extraites de 100 documents historiques sur lesquelles ont été insérées des chiffres du *dataset* 1. En même temps que leur insertion, la classe du caractère, les coordonnées X et Y du centre du chiffre, ainsi que sa largeur et sa hauteur ont été enregistrés comme label de vérité terrain.

Ainsi, si nous récapitulons les jeux de données de caractères numériques manuscrits rencontrés jusqu'alors, nous obtenons le Tableau 4.

Nom du jeu de données	Articles	Nombre de subdivision	Nombre d'images
MNIST	[MNIST]	1	70 000
ARDIS	[Kusetogullari et al., 2019]	4	32 800
USPS	United-States Postal Service	1 Contient des chiffres et des lettres.	9 298
DIDA	[Kusetogullari et al., 2020]	3	Environ 500 000

Tableau 4 : Synthèse des jeux de données de caractères numériques manuscrits rencontrés dans la littérature

I.5.2.2 Fonctionnement du modèle DIGITNET

Le modèle mis en place par [Kusetogullari et al, 2020] est composé en deux parties, comme le réseau Faster R-CNN. Dans un premier temps, ils ont mis en place le DIGITNET-dect chargé de la détection des chiffres dans le document. Et dans un deuxième temps, le DIGITNET-rec qui lui a pour mission de classer les caractères numériques détectés précédemment (voir son architecture Figure 14).

Ce qui diffère du modèle Faster R-CNN est la construction du modèle. En effet, les deux parties du réseau DIGITNET ne sont pas entraînées conjointement et ne sont pas non plus construites de la même façon. La partie DIGITNET-rec est construite en se basant sur l'architecture YOLOv3. Il s'agit d'une architecture CNN qui prédit les boîtes englobantes ainsi que la probabilité d'appartenance à une classe de nombre pour chaque boîte.

L'architecture YOLOv3 est plus performante que le Faster R-CNN car elle n'a pas besoin de deux parties à entraîner conjointement en quatre étapes, c'est donc un gain de temps et de ressources informatiques.

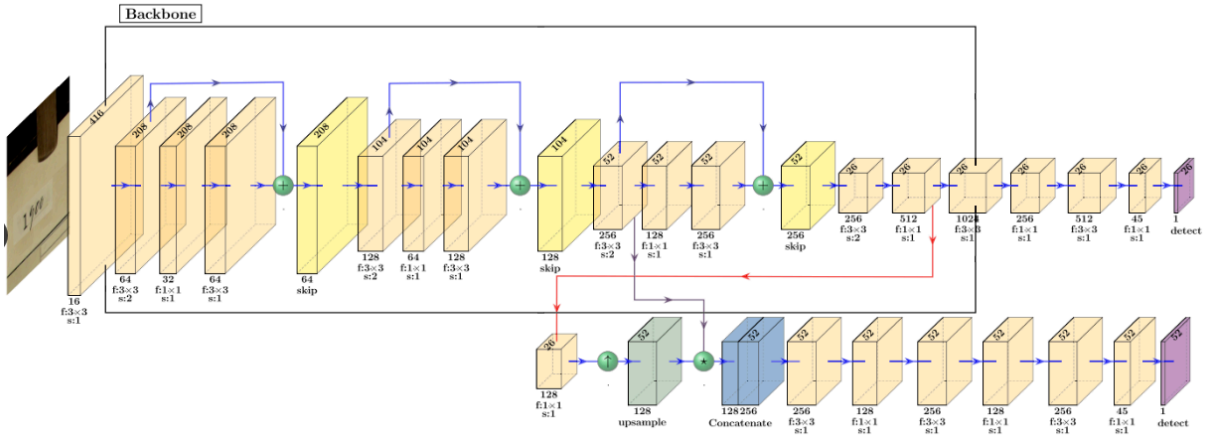

Figure 14 : Architecture du réseau DIGITNET
Source : [Kusetogullari et al, 2020]

I.5.2.3 Résultats obtenus

Grâce à leur modèle, [Kusetogullari et al, 2020] ont obtenu des résultats très concluants. Ils ont dans un premier temps comparé l'efficacité de leur modèle de détection sur différents jeux de données. Et à la suite de ce test, c'est avec le jeu de données DIDA que les résultats semble être les meilleurs.

Ensuite, ils ont mis à l'épreuve leur modèle en testant des données qui contenaient des dégradations comme l'effacement d'une partie des chiffres ou des chiffres qui se touchent entre eux. Et ils ont effectué ces mêmes tests avec d'autres modèles de détection. Au final, c'est le modèle DIGITNET-dect qui a obtenu les meilleures performances.

II Mise en place d'un réseau Faster R-CNN

Dans cette partie, nous détaillons les étapes qui ont permis de mettre en place une architecture Faster R-CNN. Cela dans le but de procéder à la détection et à la reconnaissance de caractères numériques dans des planches cadastrales anciennes, notamment les numéros de parcelles. En effet, après plusieurs recherches concernant les architectures utilisées pour procéder à de la reconnaissance de chiffres, aucun ne traite de l'architecture Faster R-CNN. C'est pourquoi nous avons choisi d'étudier cette architecture afin de comparer ses performances avec celles des autres études.

Dans un premier temps, nous allons présenter la préparation de l'outil de travail qui a permis de réaliser des traitements *deep learning*. Puis, nous avons appliqué un exemple d'architecture Faster R-CNN qui traite de la détection et de la classification des cellules du sang en se servant d'images prises au microscope. Cela permet de tester l'installation des différents outils et de vérifier notre bonne utilisation de ce modèle.

Ensuite, nous mettons en place notre propre jeu de données afin d'entraîner un modèle applicable aux planches cadastrales anciennes. En effet, au moment de ce TFE, seuls les jeux de données composés d'images sur lesquelles ne figurent qu'un seul chiffre sont disponibles en open data. Il est donc nécessaire d'élaborer une base d'images composées de plusieurs chiffres, ainsi que les étiquettes contenant les coordonnées des boîtes englobantes des objets.

II.1 Préparation de l'outil de travail

II.1.1 L'environnement de travail

Nous travaillons sous le système d'exploitation Linux. Pour être en capacité de lancer les programmes et d'avoir un temps de calcul raisonnable, nous avons mis en place un serveur de calcul composé d'une carte graphique (ou GPU) Nvidia Quadro RTX6000. Cette carte a une mémoire de 24 Go et un total de 700 cœurs. Pour faire fonctionner ce GPU, il est nécessaire d'installer l'interface de programmation CUDA qui est un ensemble de pilotes permettant au GPU d'exécuter des calculs parallélisés, ainsi que cuDNN qui est « une bibliothèque de primitives hautement optimisées pour l'apprentissage profond » [Chollet, 2020].

Afin de faire des calculs *deep learning* en langage Python, il est préférable dans un premier temps d'utiliser certaines bibliothèques comme Keras et tensorflow. Ce sont deux

bibliothèques très utilisées car elles sont libres d'accès et leurs contenus sont bien documentés[4]. Cependant, il faut faire attention aux versions téléchargées. En effet, les deux bibliothèques sont liées (voir Figure 15) et donc les versions doivent correspondre pour que le programme soit fonctionnel. Ici, nous avons utilisé la version 2.0.3 de Keras et la version GPU de tensorflow 1.15.0. D'autres bibliothèques sont nécessaires pour faire fonctionner le réseau Faster R-CNN et la mise en place de l'environnement est expliqué en Annexe 1.

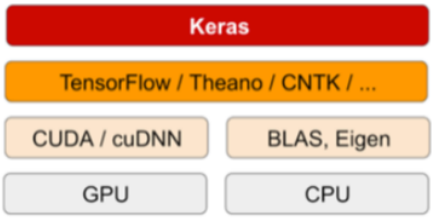

Figure 15 : La pile logicielle et matérielle du deep learning
Source : Pandey P., *My Journey into Deep Learning using Keras,* toward data sciences [en ligne], 2018. Disponible sur : https://towardsdatascience.com/my-journey-into-deeplearning-using-keras-part-1-67cbb50f65e6 (consulté le 8 juillet 2021)

II.1.2 Mise en œuvre du Faster R-CNN : exemple des cellules dans le sang

Dans cette partie, nous détaillons l'utilisation du Faster R-CNN dans le cas de la reconnaissance et de la classification de plusieurs types de cellules dans le sang. L'objectif ici est de comprendre le fonctionnement de ce modèle et de l'appliquer pour pouvoir ensuite l'adapter à notre propre problématique. L'ensemble de l'architecture a été obtenu via un site web[5] de la communauté des professionnels de l'analytique et de la science des données Analytics Vidhya, qui explique certaines parties de cette architecture.

[4] Documentation Keras : https://keras.io/guides/
Documentation Tensorflow : https://www.tensorflow.org/versions/r1.15/api_docs/python/tf?hl=fr
[5] https://www.analyticsvidhya.com/blog/2018/11/implementation-faster-r-cnn-python-object-detection/

II.1.2.1 Présentation de la structure du dossier contenant l'architecture Faster R-CNN

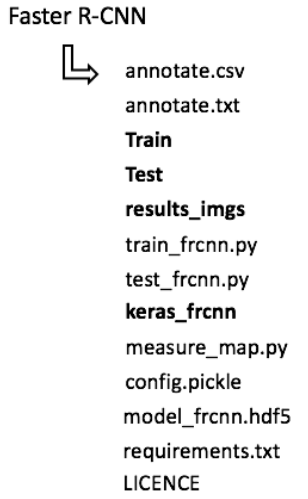

Figure 16 : Arborescence du dossier Faster R-CNN

L'une des particularités du dossier Faster R-CNN est qu'il est composé de plusieurs dossiers, fichiers et scripts Python (.py). En effet, la Figure 16 illustre l'arborescence du Faster R-CNN.

Les noms en gras sont des dossiers. Les dossiers **Train** et **Test** contiennent des images destinées à l'entraînement ou au test du modèle. Le dossier **results_imgs** est le dossier de sauvegarde des images traitées par le modèle. Donc, sur les images de **results_imgs** figurent les boîtes englobantes dessinées, ainsi que la classe attribuée à ces boîtes. Le dernier dossier intitulé **keras_frcnn** est composé de 12 scripts python. Ces scripts sont des définitions de fonctions ou de paramètres qui sont utilisés lors de l'entraînement ou du test du modèle.

Ensuite, les fichiers **annotate.csv** et **annotate.txt** répertorient les étiquettes des données d'entraînement. Pour chaque cellule de sang identifiée, une ligne avec les informations suivantes est écrite : le chemin de l'image dans laquelle est identifiée la cellule, les coordonnées du coin haut gauche et bas droit de la boîte englobante (dans l'ordre xmin, ymin, xmax et ymax) ainsi que la classe de la cellule de sang. Seul le fichier annotate.txt sert à la phase d'entraînement, cependant le fichier annotate.csv est plus simple à écrire lors de sa création, ce qui explique la présence des deux formats.

De plus, afin d'exécuter l'entraînement ou le test du modèle, nous avons besoin des scripts python **train_frcnn.py** et **test_frcnn.py**. Ce sont eux qui seront lancés en ligne de commande dans le terminal.

Au début de l'entraînement, la configuration des principaux paramètres est chargée et conservée dans le fichier **config.pickle**. Et une fois l'entraînement terminé, ce sont les poids des couches du modèle qui sont enregistrés dans le fichier **model_frcnn.hdf5**. Ce dernier est dans un format permettant sa réutilisation lors de la phase de test.

Pour finir, le script python **measure_maps.py** est un exécutable permettant de calculer la mAP pour chaque classe. Enfin, le fichier **requirements.txt** est la liste des bibliothèques python nécessaires pour exécuter les scripts.

II.1.2.2 Présentation du jeu de données

Le jeu de données d'entraînement comporte un total de 325 images avec 4 300 cellules référencées dans le fichier annotate.txt (voir un extrait du fichier à la Figure 17).

Les cellules peuvent appartenir à trois classes différentes : les plaquettes (platelets), les globules blancs (WBC) et les globules rouges (RBC). Ici, nous avons un total de 320 plaquettes, 3648 globules rouges et 332 globules blancs. Nous constatons que ces données ne sont pas équitablement représentées ce qui pourrait entraîner des difficultés pour la reconnaissance des plaquettes et des globules blancs. Cependant, étant donné qu'il y a beaucoup plus de globules rouges dans le sang (environ 4,5 million/mm^3) que de globules blancs (environ 10 000/mm^3) et de plaquettes (environ 300 000/mm^3) et que les images tests ont les mêmes caractéristiques (proportions des cellules, couleurs, tailles) que les images d'entraînements, cet écart de proportion ne semble pas poser problème à l'auteur de l'exemple.

```
1 Train\BloodImage_00000.jpg,260,177,491,376,WBC
2 Train\BloodImage_00000.jpg,78,336,184,435,RBC
3 Train\BloodImage_00000.jpg,63,237,169,336,RBC
4 Train\BloodImage_00000.jpg,214,362,320,461,RBC
```

Figure 17 : Extrait du fichier annotate.txt

II.1.2.3 Compréhension des paramètres du code

Le code train_frcnn.py permet d'entraîner le modèle avec la méthode Faster R-CNN. Dans ce code, il y a un grand nombre de paramètres détaillés ci-après.

Le premier paramètre concerne le format des données d'entrée. Le fichier à renseigner en entrée est celui contenant les étiquettes (ou annotations) des objets. Dans notre cas, nous n'avons utilisé que le format texte (extension .txt), donc l'option à indiquer en ligne de commande est « -o simple ». Cependant, il existe des jeux de données comme Pascal VOC 2007 ou Pascal VOC 2012 qui utilisent un autre formalisme pour les annotations. [Khandelwal, 2019] indique que ces jeux de données sont au format XML et comporte plus d'informations que le format texte :

- Le dossier contenant les images.
- Le nom du fichier présent dans le dossier, c'est-à-dire le nom de l'image par exemple.
- Les coordonnées des boites englobantes désignées par xmin, ymin, xmax et ymax

- La taille de l'image indiquée par sa largeur, sa hauteur et sa profondeur. Si l'image est en nuance de gris alors la profondeur est de 1, si elle est en couleur (RGB) elle sera de 3.
- Les détails de l'objet qui permettent de spécifier la classe de l'objet, si l'objet est tronqué, la difficulté à reconnaitre l'objet et le cadre de délimitation.

Dans le cas de données Pascal VOC, nous devons le spécifier dans la ligne de commande avec le paramètre « -o pascal_voc ».

Ensuite, nous devons indiquer où se trouve le fichier qui contient les annotations. Dans le cas de notre arborescence vue à la Figure 16, nous devons spécifier « -p annotate.txt » car le fichier annotate.txt se situe au même niveau que train_frcnn.py dans l'arborescence.

Nous devons aussi informer le modèle de l'architecture choisie pour les couches partagées du Faster R-CNN. Faster R-CNN prend en compte deux architectures qui sont le VGG16 et le ResNet50 et celle sélectionnée par défaut est le ResNet50.

Les paramètres output_weight_path et output_val sont les paramètres pour sauvegarder respectivement les poids du modèle dans model_frcnn.hdf5 et les valeurs de sortie à chaque époque dans historique.csv. Quant à lui, l'élément input_weight_path permet de charger des poids d'un autre modèle.

C'est aussi dans la ligne de commande que nous allons indiquer le nombre d'époques que le modèle doit effectuer. Il faudra trouver le bon nombre d'époques pour éviter tout sous-entraînement ou sur-entraînement comme expliqué en partie I.2.2.

Dans ce programme, il est possible de faire de l'augmentation de données. Par défaut, nous avons choisi de ne pas faire d'augmentation de données.

Il est aussi possible d'indiquer le nombre de RoI souhaitées (définit dans la partie I.4.2). Ce paramètre est important pour la suite car il a une forte influence sur les résultats. Le nombre de RoI sert à déterminer en combien de zones vont être découpées les fenêtres avant d'appliquer une opération de *RoI pooling* (voir la partie I.4.4).

Ensuite, nous nous intéressons aux paramètres à modifier dans le code intitulé config.py situé dans le dossier keras_frcnn. Ce fichier contient les principaux paramètres techniques du traitement Faster R-CNN. Nous retrouvons au début de ce code les paramètres pour sélectionner l'architecture du réseau partagé ainsi que ceux concernant la *data*

augmentation. Ensuite viennent les paramètres concernant les ancres. En fonction des échelles et des rapport largeur/hauteur renseignés, le modèle n'apprendra pas la même chose. Il faut que les échelles correspondent à la taille des objets que l'on cherche. Par exemple, si les objets font une taille de 256x512 pixels et d'autres objets font 128x128 pixels, nous devons renseigner deux échelles qui sont [128, 256] et deux rapports largeur/hauteur [[1,1], [1,2]]. Il peut aussi être intéressant de mettre des boîtes un peu plus petites et un peu plus grandes afin d'être sûr de trouver les objets dans l'image. Comme autre paramètre, il faut renseigner la taille à laquelle les images seront redimensionnées dans le cas où elle ne seraient pas toutes de même dimension. Nous retrouvons également le nombre de RoI qui peut être directement renseigné ici. Enfin, c'est ici que sont indiqués les seuils pour le score d'IoU qui permettent de valider ou non une proposition de régions, ainsi que les seuils de probabilité d'appartenance à une classe pour la classification des RoI.

D'autres paramètres doivent être pris en compte et notamment ceux inscrits en dur dans le code. Cela signifie qu'il faut ouvrir le programme et directement modifier leurs valeurs si besoin. C'est le cas notamment de la largeur d'une époque (*epoch_length* dans le code). Cet élément se situe à la ligne 153 du code train_frcnn.py et est lié au *batch-size*. Un forum de discussion[6] fournit la définition suivante : « Vous pouvez donc dire epoch_length=1000, ce qui signifie qu'après un entraînement sur 1000 exemples/lots, vous considérez l'époque terminée et vous en commencez une nouvelle ». Donc il s'agit du nombre d'échantillons analysés avant que l'époque ne soit interrompue. Ce paramètre a donc une forte influence sur les échantillons analysés.

II.1.2.4 Les valeurs de sorties obtenues à la fin de l'entraînement

Hormis les paramètres estimés du modèle enregistrés dans le fichier model_frcnn.hdf5, l'algorithme renvoie des valeurs de perte et de précision à la fin de chaque époque. Ces valeurs sont stockées dans le fichier historiques.csv et nous détaillons leur signification ici.

En sortie du calcul de l'entraînement, nous obtenons quatre valeurs de perte. Deux d'entre elles correspondent à la perte liées au RPN et les deux autres sont liées à la partie Fast R-CNN. Les pertes du RPN sont celles qui concernent la classification des boîtes, c'est-à-dire si une proposition de région contient un objet ou de l'arrière-plan, et la régression des

[6] https://stackoverflow.com/questions/59067368/in-training-a-faster-r-cnn-model-what-does-epoch-length-mean

boîtes, c'est-à-dire si la boîte trouvée recouvre la véritable boîte avec un recouvrement supérieur au seuil indiqué. Tandis que les pertes du Fast R-CNN correspondent à la *categorical crossentropy* pour la partie régression et à une combinaison linéaire de *binary crossentropy* pour la partie classification.

En plus de ces valeurs de perte, l'algorithme calcule deux métriques supplémentaires. Tout d'abord, il y a la valeur *mean overlapping bboxes*, et la précision de la classification des boîtes dans la bonne classe de chiffre. Ces valeurs ont été définies dans la partie I.2.1.3.

II.1.2.5 Difficultés rencontrées pour exécuter l'entraînement

Nous avons rencontrées plusieurs difficultés avant de réussir à exécuter entièrement le programme train_frcnn.py.

Le premier problème auquel nous avons été confrontés lors de l'entrainement test a été l'installation de la bibliothèque tensorflow. Nous avons besoin de la version gpu 1.15.0 de tensorflow pour qu'elle soit compatible avec la version 2.0.3 de Keras. Cependant, lors de l'exécution de l'entraînement, des conflits de fonctions apparaissent et empêchent le bon fonctionnement de tensorflow. Nous nous sommes donc rendus compte que l'ordre de téléchargements de ces deux bibliothèques peut entraîner des conflits et empêcher le bon fonctionnement des bibliothèques. Pour ne pas rencontrer ce problème, voici la procédure qu'il faut suivre :

- Étape 1 : installer Keras version 2.0.3 avec la commande : `pip install keras==2.0.3`
- Étape 2 : installer tensorflow de la même façon que précédemment : `pip install tensorflow-gpu=1.15.0`

Si la bibliothèque tensorflow a été installée avant Keras, il suffit de la désinstaller et de la réinstaller une fois que Keras a été téléchargée.

Le deuxième problème concerne le format des données. Lorsque les annotations du fichier annotate.txt sont lues, aucune image n'est chargée. Cette erreur vient du chemin des images qui n'est pas correctement orthographié. Il faut donc remplacer « \ » par « / », pour tous les chemins du fichier annotate.txt. Cela donne donc « Train/BloodImage_00000.jpg » au lieu de « Train\BloodImage_00000.jpg ».

Le troisième obstacle concerne le chargement des poids des modèles pré-entraînés sur le jeu de données ImageNet. Dans le cas où aucun chemin n'est renseigné pour le

paramètre input_weight_path, le modèle cherche les poids au même niveau de l'arborescence que le fichier train_frcnn.py. Lors de l'exécution de l'entrainement, les poids du modèle ResNet50 ne sont pas chargés par le modèle. Un message d'erreur (non bloquant pour la suite de l'exécution du script) est renvoyé dans lequel est indiquée une adresse web[7] sur laquelle ces poids sont téléchargeables. Les poids sont téléchargeables pour différentes architectures. Ceux des architectures ResNet50 et du VGG16 sont ceux pris en charge par notre modèle. Cette solution a permis d'utiliser les poids du ResNet50 pré-entraîné sur ImageNet au début du modèle Faster R-CNN. Cependant, les poids téléchargés pour le modèle VGG16 ne semblent pas fonctionner et aucune solution n'a pu être trouvée pour ce modèle.

A l'issue de la résolution de ces trois principaux problèmes, nous avons pu exécuter le programme entièrement. D'après le site à partir duquel nous avons téléchargé le programme[8], l'entraînement est assez long et tous les paramètres ne sont pas renseignés de façon claire. Cependant, les poids du modèle entraîné sont disponibles pour pouvoir les charger et tester le modèle sur les images Test et le code test_frcnn.py. Ainsi, nous avons testé le modèle et obtenu des résultats très corrects comme le montre l'exemple de la Figure 18. En effet, plusieurs globules blancs et le globule rouge ont été reconnus par le modèle avec des scores de probabilité d'appartenance à une classe élevés (au-dessus de 80% de fiabilité).

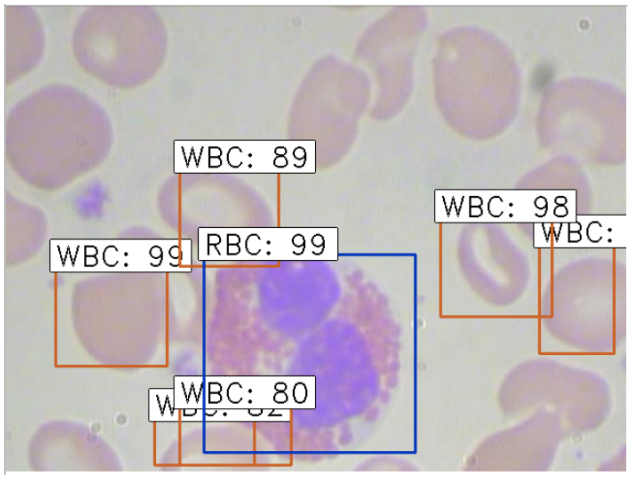

Figure 18 : Exemple de résultat après le test du modèle Faster R-CNN sur les cellules dans le sang

[7] https://github.com/keras-team/keras/blob/master/keras/applications

[8] https://www.analyticsvidhya.com/blog/2018/11/implementation-faster-r-cnn-python-object-detection/

II.1.2.6 Les performances du GPU

Afin de constater l'efficacité d'un GPU par rapport à l'utilisation d'un CPU, nous avons effectué des tests. Pour cela, nous avons entraîné le modèle en paramétrant le nombre d'époque à 1 et la longueur d'une époque à 10. Nous avons d'abord entraîné ce modèle sur le CPU et avons obtenu un temps de calcul égal à 19 minutes. Puis, nous avons effectué le même calcul sur le serveur GPU, et cette fois-ci nous avons un temps de calcul de 35 secondes, soit un facteur de 32,5.

Grâce à ce test nous avons confirmé que l'utilisation d'un GPU permet de réduire drastiquement le temps de calculs, ce qui permettra par la suite d'effectuer plus de tests. Le temps de calculs peut encore être réduit en copiant directement les données sur le serveur au lieu de les importer depuis un serveur distant et donc d'éviter des temps de latence liés au réseau informatique.

II.2 Création d'un jeu de données pour la détection et la reconnaissance des chiffres manuscrits dans des cartes anciennes

Maintenant que nous avons analysé le fonctionnement du Faster R-CNN et que nous l'avons appliqué à un exemple connu, nous devons transposer cette application à notre problème de détection et de reconnaissance de chiffres manuscrits.

Dans le but de créer un modèle *deep learning* capable de reconnaitre des caractères numériques manuscrits, il faut l'entraîner sur un jeu contenant des données semblables aux exemples qu'il rencontrera sur les planches cadastrales. Cependant, le seul jeu de données existant dans la littérature pouvant répondre à ce critère est le *dataset* DIDA. Mais il n'a pas été possible d'effectuer des tests avant la fin de ce travail car le jeu de données a été disponible trop tardivement. De plus, pour pouvoir réutiliser l'architecture Faster R-CNN précédente, il faut que les annotations soient renseignées dans un tableur. Ainsi, nous avons décidé de mettre en place notre propre jeu de données. Pour cela, nous avons écrit un programme python permettant à la fois de créer les images dont nous avons besoin et de générer le tableur qui répertorie les informations des boîtes englobantes.

L'idée principale de ce programme python est de coller des images de chiffres manuscrits provenant de *datasets* déjà existants sur une image de couleur unie.

L'objectif général de la création de ce jeu de données est de générer des images ressemblant autant que possible aux planches cadastrales qui seront par la suite testées, en conservant l'unique présence de chiffres manuscrits.

Une seconde possibilité pourrait être de créer un jeu directement à partir d'extraits de planches cadastrales. En effet, les images auraient exactement les mêmes caractéristiques que les planches. Elles contiendraient aussi bien les numéros de parcelle que les contours et les écritures de texte. Le problème de cette solution est la mise en place de ce jeu, car cela nous obligerait à étiqueter des centaines voire des milliers de numéros manuellement, chose que nous voulons éviter. C'est pourquoi la première méthode pour créer le jeu de données est celle qui a été choisie.

Dans la suite de ce mémoire, le terme « imagette » désignera les images des chiffres qui sont collés sur une plus grande image, et le terme « image » désignera une image de notre jeu de données nouvellement créé.

II.2.1 Les imagettes utilisées

Comme l'avait soulevé [Kusetogullari et al, 2020], les chiffres du *dataset* ARDIS ressemblent à ceux que nous pouvons retrouver dans les documents anciens (voir Figure 19). C'est donc ce jeu de données que nous avons utilisé.

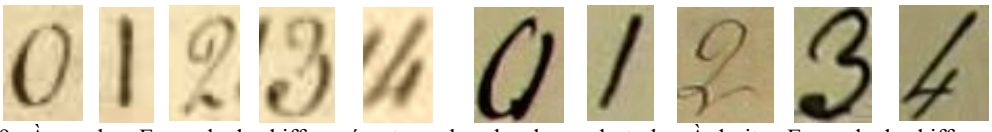

Figure 19 : À gauche : Exemple de chiffre présents sur les planches cadastrales. À droite : Exemple de chiffres du jeu de données ARDIS.

II.2.2 Le fond de l'image

Tout d'abord, nous avons dû choisir les paramètres des images qui seront créées pour notre jeu de données. Ces paramètres sont les dimensions de l'image ainsi que sa couleur de fond. Pour cela, nous avons observé les images des planches cadastrales et avons conclu qu'un découpage de ces dernières à une taille de 256x256 pixels était un choix judicieux. Nous avons fait ce choix car, lorsque nous testerons notre modèle sur les images de planches, nous devrons les découper de telle sorte que les numéros soient lisibles et que les images ne soient pas trop petites.

Ensuite, pour déterminer la couleur de fond, nous avons sélectionné plusieurs zones des planches cadastrales et avons observé la moyenne et l'écart-type. Nous en avons donc déduit que la couleur RGB du fond de l'image serait (188,181,143) et un écart-type de 0,17. En revanche, le fait de créer artificiellement une image de couleur unie n'est pas réaliste. En effet, cela produit une texture parfaitement lisse, or ce n'est pas le cas d'un fond de carte.

Nous y avons donc rajouté un bruit gaussien avec l'écart-type observé sur cette image pour qu'elle soit plus réaliste.

II.2.3 Prétraitement des imagettes avant leur insertion dans l'image

Tout d'abord, coller les imagettes directement sur l'image n'est pas satisfaisant car la couleur du fond de l'imagette ne correspond pas toujours à celle de l'image (voir Figure 20). Il apparaît donc plus pertinent de prétraiter les imagettes de manière à trouver une correspondance entre les deux couleurs de fond, et ainsi que les chiffres soient intégrés de façon naturelle dans l'image. De plus, la taille des numéros est bien supérieure à la taille de ceux que l'on retrouve sur les cartes scannées, c'est pourquoi nous avons aussi redimensionné les imagettes.

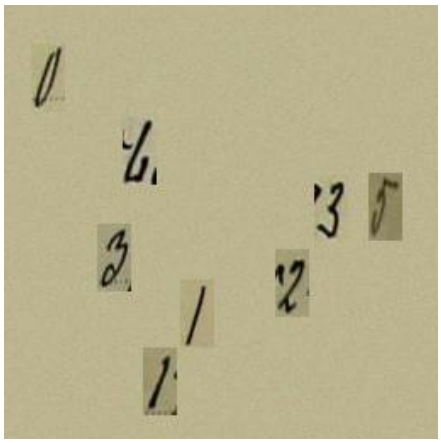

Figure 20 : Exemple d'image créé pour le jeu de données sans prétraitement des imagettes

Nous avons procédé comme suit :

- Redimensionnement des imagettes pour qu'elles aient une taille de 20x40 pixels. Ces dimensions ont été choisies en observant la taille des numéros sur les planches cadastrales.
- Classification non supervisée pour identifier les pixels qui appartiennent au chiffre et ceux qui appartiennent au fond de l'imagette.
- Modification du niveau moyen R, G et B pour les pixels qui appartiennent au fond de l'imagette, afin qu'ils correspondent à ceux du fond de l'image.

Ainsi, nous obtenons des imagettes prêtes à être insérées dans l'image.

II.2.4 Insertion des imagettes dans l'image

Désormais, nous avons à disposition des imagettes prétraitées, prêtes à être insérées sur une image. Afin, que le jeu de données ne soit pas répétitif, le nombre d'imagettes insérées est choisie aléatoirement entre 1 et 8. De plus, les imagettes sont aussi choisies de façon aléatoire dans le jeu de données ARDIS 2. Nous avons fait en sorte qu'une imagette déjà insérée sur une image ne soit pas réutilisée par la suite. Ensuite, le choix du point d'insertion est aussi aléatoire mais en respectant tout de même deux conditions : les coordonnées du point d'insertion ne doivent pas être sur une autre imagette pour éviter qu'elles ne se superposent et l'imagette ne doit pas être à cheval sur le bord de l'image.

Une fois que l'imagette est introduite, la frontière entre l'imagette et l'image doit être lissée. Pour cela, un filtre passe bas gaussien de taille 3x3 est appliqué sur tout le contour extérieur de l'imagette. Il faut donc déterminer les 9 valeurs de ce filtre. La fonction Python cv2.GetGaussianKernel fournit un noyau gaussien 1D (soit un vecteur colonne de taille 3x1). Ainsi, nous avons calculé les coefficients du noyau gaussien 1D, puis nous l'avons transformé en noyau 2D par produit scalaire. Cela est possible car les noyaux gaussiens sont des filtres séparables.

II.2.5 Enregistrement des étiquettes dans un tableur

Pour chaque imagette insérée, les données nécessaires à l'apprentissage du modèle sont enregistrées dans le fichier nommé annotation.csv.

Pour l'utilisation de ce fichier pendant la phase d'entraînement, il est nécessaire de le convertir en fichier texte (extension .txt), avec comme séparateur la virgule. De plus, si le dossier contenant les images ou le code train_frcnn.py sont déplacés , il faudra vérifier que les chemins indiqués dans la première colonne soient toujours corrects.

Une fois que le bon nombre d'imagettes est inséré dans l'image, cette dernière est enregistrée et le programme peut passer à la création des images suivantes. A l'issue de ces traitements, nous obtenons un jeu de donnée d'entraînement composé d'un total de 1663 images contenant 7596 chiffres manuscrits. La Figure 21 montre quelques exemples d'images obtenues après la création du jeu de données.

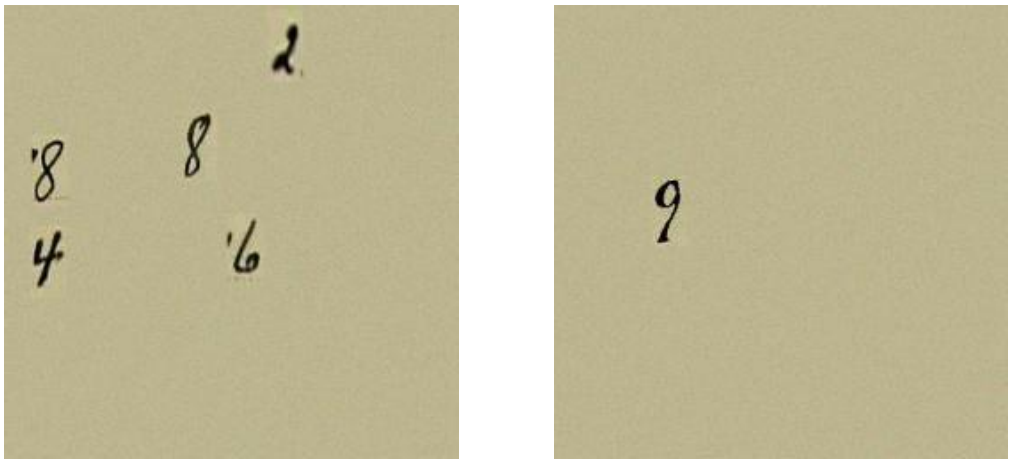

Figure 21 : Exemples d'image du jeu de données créé

Pour la phase de test, nous avons créé un jeu de données de 10 images de la même façon que précédemment pour avoir des images différentes de celles du jeu d'entraînement (même si elles s'appuient aussi sur le dataset ARDIS ce sont des chiffres différents).

II.3 Adaptation de l'architecture Faster R-CNN à la reconnaissance des chiffres manuscrits dans les cartes anciennes

Pour pouvoir utiliser le réseau Faster R-CNN sur notre jeu de données, nous devons adapter ses paramètres. Dans la partie II.1.2.3, nous avons vu les principaux paramètres à modifier pour obtenir des résultats cohérents. Nous allons donc tous les passer en revue et expliquer les valeurs choisies pour l'entraînement du modèle.

II.3.1 Les paramètres du fichier config.py

Tout d'abord, en ce qui concerne le choix du réseau convolutif pré-entrainé qui intervient dans la première partie du Faster R-CNN, nous n'avons pas d'autre choix que d'utiliser le ResNet50. En effet, n'ayant pas réussi à faire fonctionner le VGG16, nous n'avons pas de résultats concernant cette méthode, et ce paramètre ne changera pas au cours des différents tests.

Nous ne ferons pas non plus de *data augmentation*, du moins dans un premier temps, afin d'évaluer les performances du jeu de données que nous avons mis en place.

Ensuite, il faut choisir les paramètres des ancres comme vu à la section I.4.4.1. Ces éléments sont très importants car ce sont eux qui vont définir la taille des boîtes englobantes que va rechercher le modèle. Ainsi, pour que les dimensions des boîtes correspondent à celles renseignées dans le fichier annotation.txt, nous avons choisi une échelle d'ancre de 28

et un ratio de 1/2. Cela engendre une taille de proposition de boîte de 28x56 pixels ce qui permet une marge pour la détection des chiffres. Nous avons effectué plusieurs tests notamment avec des échelles d'ancres égales à 20, 28 et 40 et des rapports largeur/hauteur égaux à 1/1, 1/2 et 2/1.

Nous avons modifié le nombre de RoI directement en ligne de commande. La valeur retenue après plusieurs tests est la valeur 18. Celle-ci a été déterminée grâce à [Erdem, 2020]. Étant donné que nos images sont au format 256x256x3 (largeur, hauteur et nombre de canaux ici RGB) et que l'architecture du réseau réorganise ses données au format 14x14x1024, nous pouvons trouver le nombre de RoI en faisant 256/14 ce qui donne 18 RoI.

Le reste des paramètres du fichier config.py n'est pas modifié car il s'agit des paramètres utilisés couramment pour obtenir des résultats corrects. Nous les changerons dans le cas où le modèle ne serait pas suffisamment performant.

II.3.2 Les paramètres de la ligne de commande

L'élément qui ne change pas en ligne de commande est le format des données d'annotations, puisque nous avons gardé le formalisme du fichier texte. Bien évidemment, les chemins de sauvegarde des données de sorties changeront en fonction des tests effectués. Le paramètre que nous avons fait varier est le nombre d'époques. En effet, lors de l'exemple des cellules de sang, la documentation conseillait d'entraîner le modèle pendant 500 époques. Nous avons donc testé cette valeur, cependant nous avons constaté que l'entraînement durait entre 22h et une journée. Afin de réduire ce temps de calcul, nous avons diminué ce nombre à 250 puis à 120 époques pour effectuer des tests plus rapides. Une fois que les autres paramètres seront convenables, nous pourrons ré-augmenter cette valeur.

III Analyse des résultats

III.1 Optimisation des paramètres pour l'entraînement

Lors des nombreux tests effectués pour trouver les paramètres adéquates, nous avons appliqué le protocole suivant :

- Étape 1 : Entraînement du modèle en modifiant un paramètre à la fois pour bien comprendre l'influence qu'il a sur le réseau et ses résultats. Les paramètres qui ont été modifiés au cours de ces tests sont : le nombre d'époques, le rapport et l'échelle des ancres et le nombre de RoI.
- Étape 2 : Analyse des courbes de perte et de performance (précision et *mean overlapping bboxes*) du modèle.
- Étape 3 : Test du modèle sur 10 images similaires au jeu de données. Ce sont des images qui ont été créées avec le même protocole que celui mis en place pour le jeu de données d'entraînement.
- Étape 4 : Analyse des images traitées lors de la phase de test et conclusion sur l'efficacité des paramètres choisis.
- Étape 5 : Deuxième test effectué sur des extraits de planches cadastrales. En effet, si le réseau est jugé performant sur les images de tests précédentes, nous effectuons ce deuxième test. Puis nous analysons ces résultats.

Test	Nombre d'époques	Échelle des ancres	Rapport largeur/hauteur des ancres	Nombre de RoI	Temps d'entraînement	Test du modèle (seuil de probabilité à 80%)	Test du modèle (seuil de probabilité à 50%)
1	250	28, 40	1/2	32	11h50	8%	18%
2	120	28, 40	1/2	18	4h40	54%	78%
3	120	28, 40	1/2	4	4h20	30%	Non testé

Tableau 5 : Quelques tests d'entrainement de modèles avec différents paramètres

Le tableau ci-dessus montre des exemples de tests effectués. Les pourcentages indiqués dans les deux dernières colonnes correspondent aux pourcentages de caractères correctement détectés et reconnus avec une probabilité d'appartenance à la classe supérieure au seuil indiqué. De nombreux autres tests ont été fait durant ce TFE, cependant ils ne sont pas comparables avec ceux du tableau car ils n'ont pas été entraînés sur le même jeu de données c'est pourquoi ils ne sont pas présentés.

Suite à plusieurs tests, le modèle le plus efficace trouvé à ce jour est celui dont les paramètres sont les suivants : un nombre d'époque égal 120, une largeur d'époque de 1 000, deux échelles d'ancres valant 28 et 40, un rapport d'ancre largeur/hauteur égal à 1/2 et 18 RoI.

Le modèle a été entrainé sur le jeu de données créé et nous avons obtenu les valeurs de perte, de précision et de *mean overlapping bboxes* visibles sur la Figure 22, Figure 23 et Figure 24. Les graphes illustrent les variables calculées à chaque fin d'époques, c'est pourquoi il y a 120 valeurs.

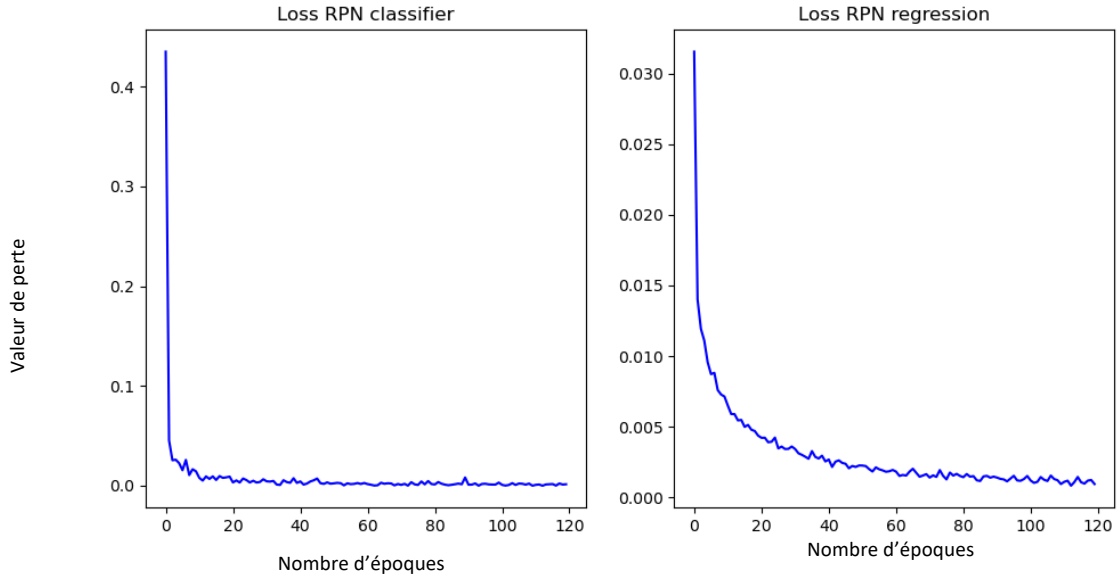

Figure 22 : Graphiques des pertes du RPN en fonction du nombre d'époques

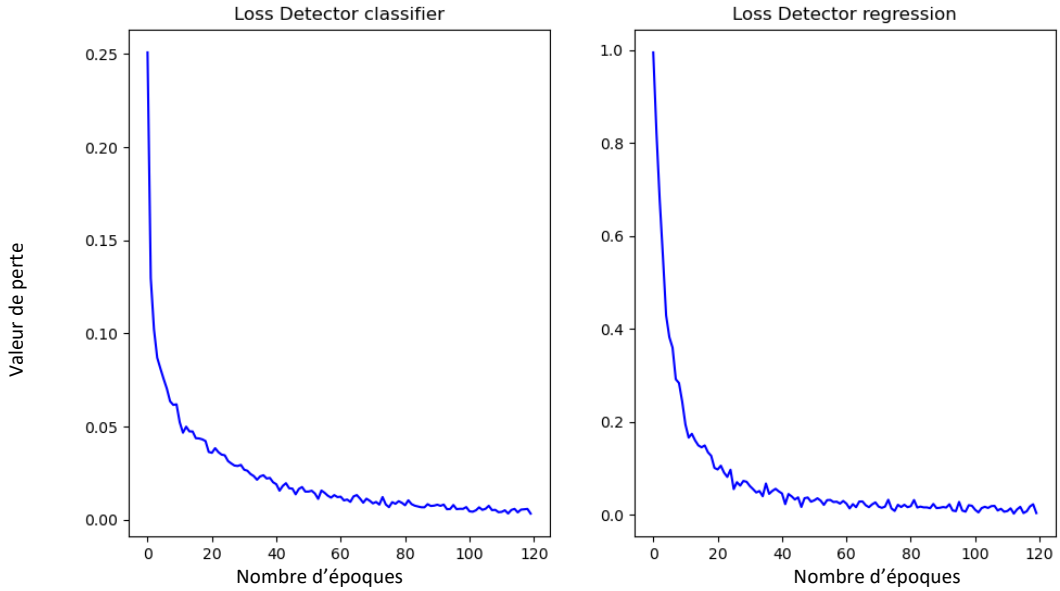

Figure 23 : Graphiques des pertes de la classification

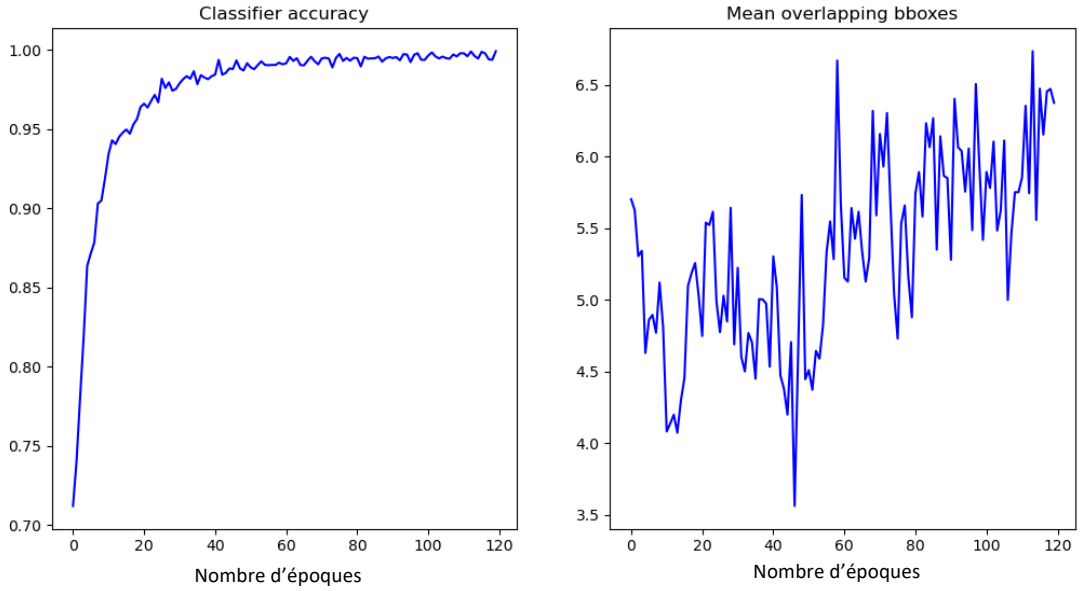

Figure 24 : À gauche : Graphique de la précision de classification des boîtes. À droite : Graphique du *mean overlapping bboxes*

Sur ces figures, nous remarquons dans un premier temps que les quatre valeurs de perte diminuent au cours de l'entraînement, ce qui signifie que l'entraînement du modèle progresse correctement. Ensuite, nous constatons que la valeur de précision du classificateur vaut 0,9991. Cela traduit le fait que les boîtes proposées par le RPN sont correctement classées à 99,91% par le Fast R-CNN, ce qui est encourageant pour la phase de test à venir. De plus, la valeur finale du *mean overlapping bboxes* est proche de 6,5 à la fin de l'entraînement. Cette valeur est censée se rapprocher du nombre moyen d'imagettes par image dans le jeu de données, soit 4,56, puisqu'elle traduit le nombre de boîte par image jugées correctes par le modèle. La valeur 6,5 est supérieure à celle nous devrions trouver. Cela peut se justifier par le fait que plusieurs propositions de boîtes englobantes peuvent concerner un même chiffre, et plusieurs d'entre elles peuvent remplir les critères de correspondance. Ainsi, plusieurs boîtes peuvent être retenues pour un même caractère numérique.

Puis nous avons effectué l'étape 3 de notre protocole d'évaluation du modèle. Ainsi, ce modèle a été testé sur les 10 images de test similaires au jeu de données d'entraînement.

Dans un premier temps, nous avons testé le modèle sur les 10 images en fixant le seuil de probabilité d'appartenance à une classe d'une proposition de boîte à 80%. Donc, si une boîte a été détectée mais que l'algorithme n'est pas sûr à 80% qu'elle soit bien classée, alors elle ne sera pas retenue pour le résultat final. Pour ce test, le modèle a identifié 29

numéros sur les 50 présents dans les 10 images, soit 58%. Et parmi ces 29 boîtes 27 ont été correctement classées, soit 93% des boîtes détectées. Nous pouvons en déduire que le modèle n'est pas optimisé pour la partie détection des numéros mais qu'il est performant concernant la partie reconnaissance des numéros. Nous pouvons observer ces résultats à la Figure 25.

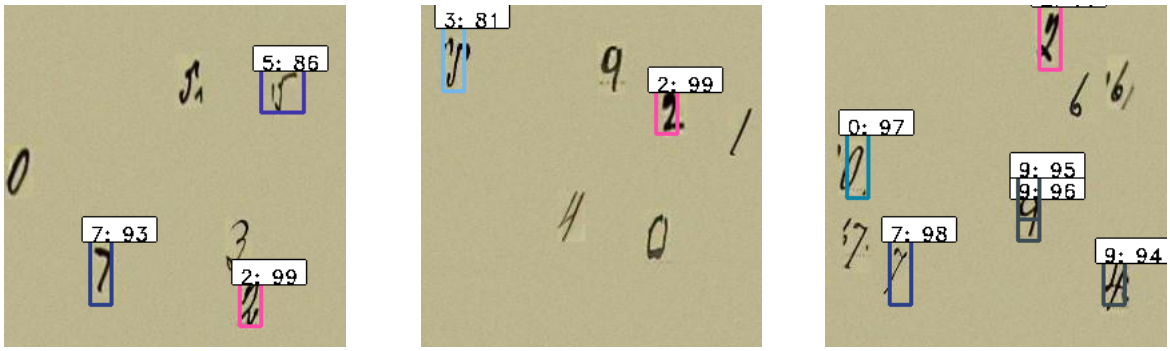

Figure 25 : Résultats du premier test avec un seuil de recouvrement à 80%

Dans un second temps, nous avons effectué le même test en diminuant le seuil de probabilité d'appartenance à une classe à 50%. Nous obtenons plus de résultats pour la partie détection puisqu'un total de 48 boîtes est trouvé sur les 50 (soit 96%) et que parmi elles, 39 appartiennent à la bonne classe, soit 81% des boîtes détectées. Ces résultats sont visibles sur les images de la Figure 26.

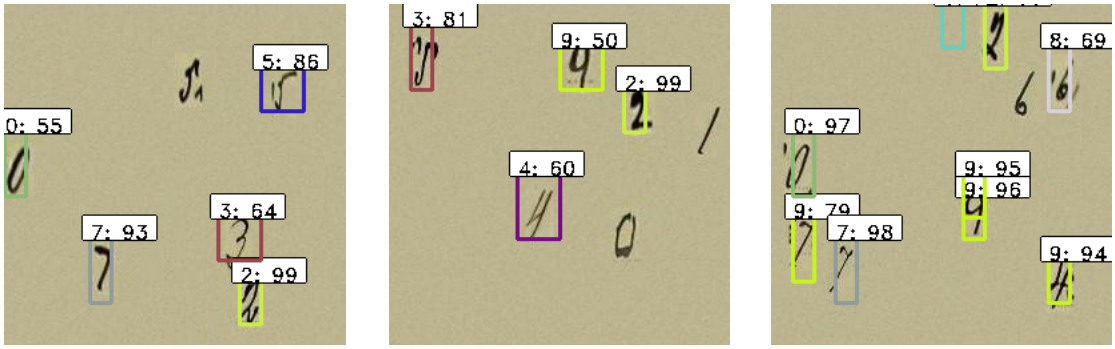

Figure 26 : Résultats du deuxième test avec un seuil de recouvrement à 50%

Nous constatons tout de même que certains numéros ne sont pas du tout détectés par le modèle, ou encore que d'autres sont recouverts par plusieurs boîtes englobantes. De plus, le contour des boîtes n'est pas optimal. Par exemple, sur la Figure 26 nous voyons que la boîte du numéro 4 (de l'image du centre) est bien plus grande que la place occupée par le numéro. Nous observons aussi que des numéros sont tronqués, comme le numéro 5 ou le 2 de la même image. Cela peut être dû à l'entraînement qui ne comporte pas assez d'époques ou aux paramètres des ancres. De plus, il faudrait davantage de données pour confronter le

modèle à une plus grande variété de caractères numériques et donc à un plus grand nombre de boîtes.

III.2 Préparation des planches cadastrales

L'objectif premier de ce projet était de détecter les numéros de parcelles sur des images de planches cadastrales scannées. Cependant, pour appliquer le réseau *deep learning* précédent aux planches cadastrales, nous devons les transformer au format attendu en entrée de ce réseau. Ainsi, nous avons conçu un script Python qui découpe les planches cadastrales en images de taille 256x256 pixels. Étant donné que des numéros pourraient être tronqués par ce découpage, nous avons ajouté un recouvrement de 20% entre chaque image découpées afin qu'aucun numéro de parcelle ne soit omis. Nous obtenons donc un grand nombre d'images pour une seule planche cadastrale, par exemple pour la planche 015.jpg de Vaas (Figure 27), qui a une dimension de 8107x5359 pixels, nous obtenons 1 039 images de taille 256x256.

Figure 27 : Planche cadastrale de la commune de Vaas de 1813

III.3 Analyse des résultats obtenus et perspectives

À la suite du découpage d'une planche cadastrale de la commune de Vaas, nous avons effectué un test du modèle sur quelques images afin d'analyser son comportement en conditions réelles. Malheureusement, les 11 images qui ont été testées avec un seuil de 80%

ne renvoient pas de résultats concluants puisque très peu de numéros sont trouvés et quasiment aucun ne sont correctement reconnus (voir Figure 28). En effet, sur environ 81 chiffres présents dans ces images test, 30 boîtes ont été dessinées (soit 37%) et 5 labels sont corrects, soit 17% des boîtes trouvées et 6% des labels à trouver. Nous remarquons que sur les 5 boîtes correctement reconnues, 4 d'entre elles correspondent au chiffre 7. Nous pouvons émettre l'hypothèse que le modèle mis au point est plus performant pour cette classe.

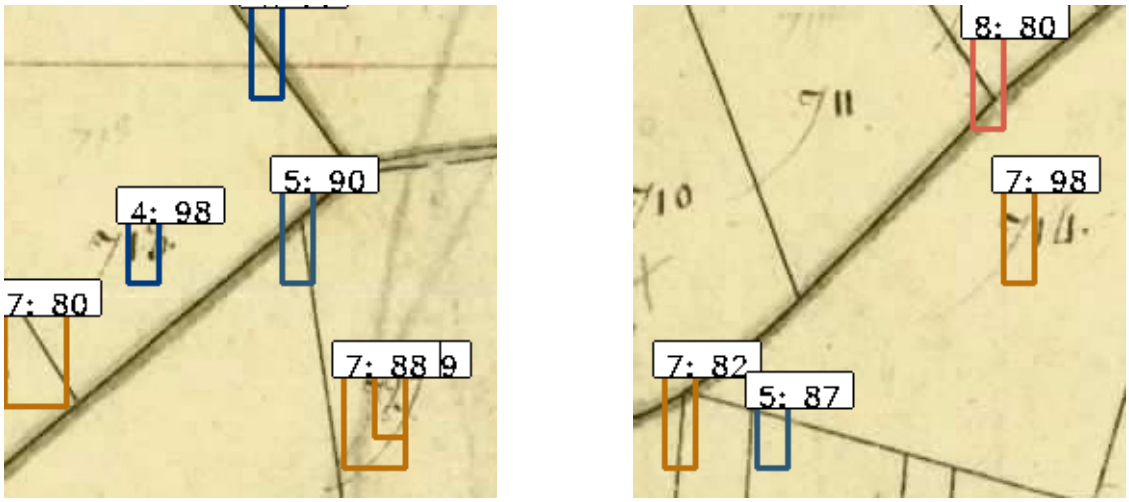

Figure 28 : Résultats obtenus sur les images tests (de dimension 256x256) de la planche cadastrale de Vaas de 1813 avec un seuil de 80%

De plus, des lettres et des intersections de limites de parcelle ont été confondues avec des chiffres comme nous pouvons le voir sur la Figure 28. Aussi, les boîtes semblent être trop grandes pour la taille des numéros.

Afin de constater si la détection et la reconnaissance des chiffres manuscrits peuvent être améliorées en baissant le seuil de probabilité d'appartenance à une classe, nous avons de nouveau testé le modèle avec un seuil à 50%. La Figure 29 montre les mêmes extraits de planches que la Figure 28, testés avec un seuil de 50%.

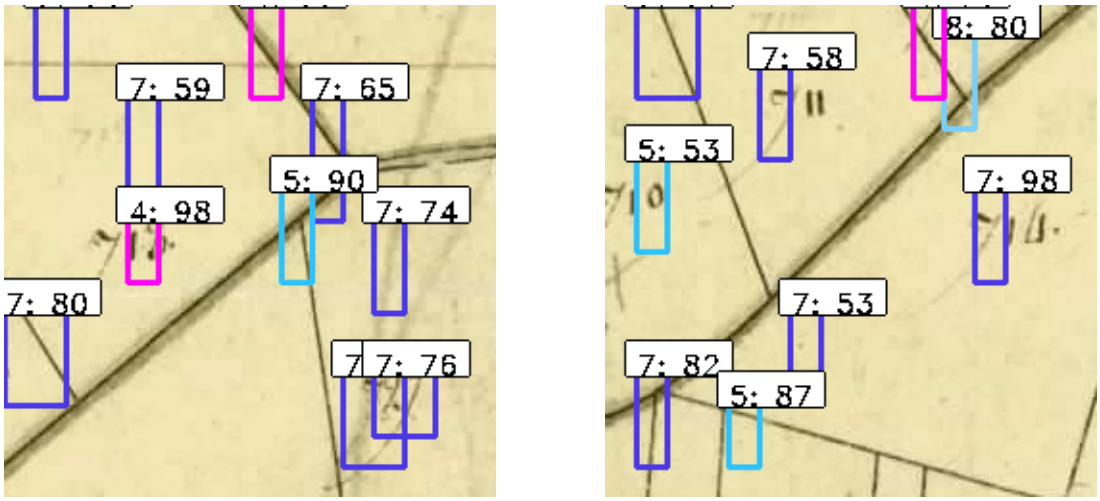

Figure 29 : Résultats obtenus sur la planche cadastrale de Vaas de 1813 avec un seuil de 50%

Comme nous pouvons le constater les résultats sur les planches cadastrales ne sont pas probants. En effet, ici aussi nous constatons que beaucoup de boîtes sont dessinées à des endroits qui ne contiennent aucun numéro. En général, les boîtes trouvées sont trop larges ou trop allongées.

Ce qui ressort de ces résultats est que le modèle n'est pas correctement entraîné pour être appliqué aux planches cadastrales. En effet, il serait judicieux de continuer à modifier les paramètres pour améliorer les résultats sur le jeu de données que nous avons créé avant de passer aux planches cadastrales. Pour cela, nous pourrions essayer d'autres tailles d'ancres et rapport d'échelles car les boîtes ne semblent toujours pas être adaptées à la taille des numéros.

Une autre solution serait de tester notre modèle avec le jeu de données DIDA. Actuellement, seuls les jeux de données comportant des chiffres uniques sont disponibles. Ce sont des *datasets* similaires au jeu de données ARDIS mais ils contiennent plus de données, à savoir 10 000 ou 70 000 chiffres manuscrits (voir partie I.5.2.1). Il faudrait donc adapter notre programme pour qu'il puisse se baser à la fois sur les données de DIDA et d'ARDIS. L'utilisation de ce nouveau jeu de données permettrait une plus grande représentation des chiffres manuscrits et donc un meilleur entraînement pour le réseau *deep learning*.

Conclusion

L'automatisation de la détection des numéros de parcelle via un réseau d'apprentissage profond s'inscrit dans une volonté d'amélioration de la chaîne de traitement, élaborée dans le cadre du projet GefVectMoCad. À terme, cette automatisation devrait permettre de recouper les informations de la vectorisation des contours des parcelles avec leurs numéros.

La mise en place d'un réseau *deep learning* nécessite des jeux de données d'entraînement et de test composés des mêmes caractéristiques que les éléments qui seront, par la suite, analysés par ce réseau. C'est pourquoi, la mise en place d'un jeu de données adéquate est primordiale. Dans le cadre de ce projet, nous avons créé notre propre jeu de données en faisant des choix concernant les chiffres manuscrits adoptés ou la méthode d'élaboration. L'expérimentation d'autres jeux de données, et notamment le DIDA *dataset*, est une piste d'amélioration à envisager.

Une autre solution concernant le jeu de données d'entraînement serait d'ajouter sur les images des éléments comme des limites de parcelles ou du texte. Ainsi, cela permettrait d'apprendre au modèle à différencier ces éléments des numéros qu'il doit détecter.

De plus, le choix de l'architecture Faster R-CNN a été fait car il s'agit d'un réseau assez performant pour la détection et l'identification d'objets. En outre, le problème de la détection de caractères numériques manuscrits n'avait jusqu'alors pas été réellement traité à l'aide de cette architecture. Néanmoins, d'autres architectures comme YOLOv3 ou celle mise au point par [Kusetogullari et al, 2020] ont été expérimentées et offrent des résultats très performants d'après la littérature. Ainsi, les dernières versions d'architectures YOLO pourront être étudiées afin d'obtenir une meilleure détection des chiffres.

Toujours concernant les architectures de réseaux, il pourrait aussi être possible d'expérimenter d'autres réseaux pré-entraînés en entrée de notre modèle Faster R-CNN. En effet, dans le cadre de ce projet, seul le ResNet50, pré-entraîné sur le *dataset* ImageNet, a été testé. Or l'architecture VGG16, qui a été évoquée, pourrait apporter des résultats différents.

Enfin, d'autres entraînements de modèles pourraient être effectués notamment en modifiant les paramètres des ancres et la largeur d'une époque. Avec un jeu de données plus important, de l'ordre de 100 000 boîtes englobantes, la largeur d'une époque pourrait être réduite afin que les valeurs aberrantes aient moins d'influence sur l'entraînement.

Bibliographie

Ouvrages imprimés

[Chollet, 2020] : CHOLLET F. L'apprentissage profond avec python. France : Machinelearning.fr, 2020, 487 p.

[Patterson et al., 2018] : PATTERSON J. et GIBSON A. Deep learning en action : la référence du praticien. Paris : O'Reilly, 2018, 534 p.

Travaux universitaires

[Chalais, 2019] : CHALAIS A. Vectorisation du cadastre anciens : restructuration de la chaîne de traitement, implémentation d'une nouvelle méthode de détection et utilisation de la théorie des graphes. Mémoire d'ingénieur, Le Mans, France : CNAM-ESGT, 2019, 77 p.

Articles de périodiques

- Imprimés

[Girshick, 2015] : GIRSHICK R., *Fast R-CNN*, IEEE International Conference on Computer Vision, 2015, pp. 1440-1448, doi: 10.1109/ICCV.2015.169. [en ligne] Disponible sur : https://ieeexplore.ieee.org/document/7410526 (consulté le 21 juin 2021).

[Heitzler et al., 2020] : HEITZLER M. et HURNI L. *Cartographic reconstruction of building drawn symbols in old maps : A study on the Swiss Siegfried map*. 2020. Transactions in GIS. 24. 10.1111/tgis.12610.

[Kusetogullari et al., 2019] : KUSETOGULLARI H., YAVARIABDI A. et al. *ARDIS: a Swedish historical handwritten digit dataset*. 2020. Neural Comput & Applic 32, 16505–16518. Disponible sur : https://link.springer.com/article/10.1007/s00521-019-04163-3 (consulté le 22 mars 2021).

[Kusetogullari et al., 2020] : KUSETOGULLARI H., YAVARIABDI A. et al. *DIGITNET: A Deep Handwritten Digit Detection and Recognition Method Using a New Historical Handwritten Digit Dataset*. 2020. Science direct Big data research, volume 23, 100182.

[Laumer et al., 2020] : LAUMER D., GÜMGÜMCÜ H. et al. *A semi-automatic Label Digitization Workflow for the Siegfried map*. 2020, dans International workshop organized by the ICA Commission on Cartographic Heritage into the Digital à Budapest le 13 Mars 2020. p55-62.

- Électroniques

[Doshi, 2019] : DOSHI S., *Various Optimization Algorithms for Training Neural Network*. Towards data science, 2019 [en ligne]. Disponible sur : https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6 (consulté le 14 juin 2021).

[Dumoulin et al., 2018] : DUMOULIN V. et VISIN F. *A guide to convolution arithmetic for deep learning*. 2018. Cornell university [en ligne]. Disponible sur : https://arxiv.org/abs/1603.07285 (consulté le 5 juillet 2021).

[Erdem, 2020] : ERDEM K. *Understanding Region of Interest (RoI pooling)*. Towards Data Science, 2020 [en ligne]. Disponible sur : https://towardsdatascience.com/understanding-region-of-interest-part-1-roi-pooling-e4f5dd65bb44 (consulté le 15 juin 2021).

[Gandhi, 2018] : GANDHI R. *R-CNN, Fast R-CNN, Faster R-CNN, YOLO – Object Detection Algorithms*. Towards Data Science, 2018 [en ligne]. Disponible sur : https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e (consulté le 27 avril 2021).

[Girshick et al., 2014] : GIRSHICK R., DONAHUE J., et al. *Rich feature hierarchies for accurate object detection and semantic segmentation Tech report (v5).* Disponible sur : https://arxiv.org/pdf/1311.2524.pdf (consulté le 21 juin 2021).

[Khandelwal, 2019] : KENDELWAL R. COCO and Pascal VOC data format for Object detection. Towards Data Science, 2019 [en ligne]. Disponible sur : https://towardsdatascience.com/coco-data-format-for-object-detection-a4c5eaf518c5 (consulté le 9 juillet 2021).

[LeCun et al., 1989] : LECUN Y., BOSER B. et al. *Backpropagation applied to Handwritten Zip Code Recognition,* 1989. Disponible sur : https://direct.mit.edu/neco/article/1/4/541/5515/Backpropagation-Applied-to-Handwritten-Zip-Code (consulté le 5 mai 2021).

[Marcelino, 2018] : MARCELINO P., *Transfert learning from pre-trained models*. Towards Data Science, 2018 [en ligne]. Disponible sur : https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751 (consulté le 23 juin 2021).

[Mitsa, 2019] : MITSA T. How do you know you have enough training data. Towards Data Science, 2019 [en ligne] Disponible sur : https://towardsdatascience.com/how-do-you-know-you-have-enough-training-data-ad9b1fd679ee (consulté le 1er juillet 2021).
[Ren et al, 2016] : REN S., HE K., et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. Cornell University [en ligne]. Disponible sur : https://arxiv.org/pdf/1506.01497.pdf (consulté le 10 juin 2021).
[R. R. Uijlings et al, 2013] : R. R. UIJLINGS J. , SANDE K. E. A., et al. *Selective Search for Object Recognition*. 2013, Research Gate [en ligne]. Disponible sur : https://www.researchgate.net/publication/262270555_Selective_Search_for_Object_Recognition (consulté le 2 juin 2021).
[Saha, 2018] : SAHA S. A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. Toward Data Science, 2018 [en ligne]. Disponible sur : https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (consulté le 9 juin 2021).
[Sambasivarao, 2019] : SAMBASIVARAO K., *Region of interest*. Toward data science, 2019 [en ligne]. Disponible sur : https://towardsdatascience.com/region-of-interest-pooling-f7c637f409af (consulté le 8 juillet 2021).
[Simonyan K. et al., 2015] : SIMONYAN K. et ZISSERMAN A. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015, Cornell University [en ligne]. Disponible sur : https://arxiv.org/abs/1409.1556 (consulté le 7 juillet 2021).
[Yohanandan, 2020] : YOHANANDAN S. *mAP (mean Average Precision) might confuse you!* Toward Data Science, 2020 [en ligne]. Disponible sur : https://towardsdatascience.com/map-mean-average-precision-might-confuse-you-5956f1bfa9e2 (consulté le 11 juin 2021).
[Zeiler et al., 2014] : ZEILER MD. et FERGUS R. *Visualizing and Understanding convolutional networks*. 2014, Springer link [en ligne]. Disponible sur : https://link.springer.com/chapter/10.1007/978-3-319-10590-1_53 (consulté le 31 mai 2021).

Sites web

[ALPAGE]. ALPAGE. AnaLyse diachronique de l'espace urbain Parisien approche GEomatique, [en ligne]. Disponible sur : https://alpage.huma-num.fr/glossaire-geomatique/ (consulté le 3 juin 2021).

[ESRI] : ESRI. Support technique : Dictionnaire SIG, [en ligne]. Disponible sur : https://support.esri.com/fr/other-resources/gis-dictionary/term/c37dcc4d-7b91-4ed8-bca1-59ee5fb1c39e (consulté le 3 juin 2021).
[MNIST] : The MNIST Database of Handwritten Digits, [en ligne]. Disponible sur : http://yann.lecun.com/exdb/mnist/ (consulté le 22 mars 2021).
Forum de discussion concernant la longueur d'une époque, [en ligne]. Disponible sur : https://stackoverflow.com/questions/59067368/in-training-a-faster-r-cnn-model-what-does-epoch-length-mean

Conférence

[Chazalon J., 2021] : CHAZALON J. Séminaire du projet VECCAR, MSH Val de Loire. Vectorisation de plans anciens et apprentissage profond (deep learning). Tours, le 8 avril 2021 [en ligne]. Disponible sur : https://www.canal-u.tv/video/msh_val_de_loire/vectorisation_de_plans_anciens_et_apprentissage_profond_deep_learning.60461
[Dupas Y., 2021] : DUPAS Y. Séminaire du projet VECCAR, MSH Val de Loire. L'apprentissage machine pour vectoriser des cartes anciennes. Tours, le 8 avril 2021 [en ligne]. Disponible sur : https://www.canal-u.tv/video/msh_val_de_loire/l_apprentissage_machine_pour_vectoriser_des_cartes_anciennes.60441

Table des annexes

Annexe 1
Notice de démarrage du Faster R-CNN

le cnam
GeF

Notice de démarrage du Faster R-CNN

TFE Margot GLENAZ 2021

A qui s'adresse cette notice ?

Cette notice s'adresse aux personnes qui souhaitent reprendre ou réaliser des traitements de deep learning avec l'installation mise en place par le laboratoire GeF en 2021.
Cette notice reprend les étapes qui ont été mise en place dans le cadre du TFE de M. GLENAZ intitulé "Détection et reconnaissance de chiffres manuscrits à partir de plans cadastraux anciens par apprentissage profond".
Elle décrit les étapes à suivre pour créer un environnement de calcul fonctionnel sous le système d'exploitation Linux et pour installer les bibliothèques nécessaires au fonctionnement. Les commandes données sont celles à entrer directement dans le terminal une fois que vous vous êtes connectés au serveur de calcul nommé lamarr.

Sur le serveur lamarr sont fournis plusieurs README.txt qui permettent de comprendre l'organisation des dossiers.

Bonne lecture et bonne installation !

CRÉATION DE L'ENVIRONNEMENT

Les instructions sont données pour une installation via le terminal de l'ordinateur sous le système d'exploitation Linux.

Afin de faire fonctionner correctement les différents programmes qui composent le Faster R-CNN, il est nécessaire d'installer un certain nombre de bibliothèques Python. Pour cela, nous conseillons de créer un environnement virtuel (comme présenté ci-dessous) afin de ne pas altérer les autres installations qui pourrait déjà exister.

1. **Installer Anaconda3**

2. **Créer un environnement**
   conda create --name nom_environnement

3. **Activer l'environnement**
   conda activate nom_environnement

Il est possible de voir la liste des environnements déjà créés avec :
conda env list

TÉLÉCHARGEMENT DES BIBLIOTHÈQUES PYTHON

1. **Installer Python dans l'environnement :**
   conda install python=3.7.0

Attention : Bien spécifier les versions des bibliothèques en ligne de commande lorsque celles-ci sont précisées.

2. **Installer les bibliothèques nécessaires avec conda :**
   conda install numpy h5py pandas matplotlib

3. **Installer les autres bibliothèques avec pip :**
   pip install opencv-python
   pip install sklearn
   pip install keras==2.0.3
   pip install tensorflow-gpu=1.15.0

Attention : Ne pas installer tensorflow-gpu avant keras sinon des conflits seront engendrés lors de l'utilisation des bibliothèques. Dans le cas où tensorflow-gpu serait déjà installé, le désinstaller puis installer keras et enfin ré-installer tensorflow-gpu.

Liste des figures

Liste des tableaux

Détection et reconnaissance de chiffres manuscrits à partir de plans cadastraux anciens par apprentissage profond.

Mémoire d'Ingénieur C.N.A.M. E.S.G.T., Le Mans 2021

RESUME

Les planches cadastrales anciennes sont une source d'informations importantes pour l'étude du territoire au fil du temps. C'est dans l'optique de préserver ces informations que le laboratoire Géomatique et Foncier se concentre sur le géoréférencement, la vectorisation et le mosaïquage des planches cadastrales anciennes. Pour cela, le laboratoire a mis en place depuis 2016 une chaîne de traitement.

L'objectif de ce TFE est d'optimiser la détection des contours de parcelle en se servant des numéros de parcelle. Étant donné qu'un numéro de parcelle est associé à un contour, la détection de ces numéros permet d'améliorer les résultats. Cependant, les documents étant anciens, leurs numéros sont manuscrits et donc difficiles à détecter avec des algorithmes « classiques » de classification. C'est pourquoi l'apprentissage profond (ou deep learning) semble parfait pour cette tâche.

Mots clés : Apprentissage profond, Deep learning, Planches cadastrales anciennes, Chiffres manuscrits.

SUMMARY

The ancient cadastral maps are a source of important information for the study of the territory over time. It is with the aim of preserving this information that the Géomatique et Foncier Laboratory focuses on georeferencing, vectorization and mosaicing of ancient cadastral maps. Since 2016, the laboratory has implemented a processing chain.

The objective of this TFE is to optimize the detection of parcel contours using parcel numbers. Since a parcel number is associated with a contour, detection of these numbers could improve the results. However, since documents are old, their numbers are handwritten and therefore difficult to detect with «classical» classification algorithms. That's why deep learning seems perfect for this task.

Key words : Deep learning, Ancient cadastral maps, handwritten numbers.