



Ecole Polytechnique de l'Université de Tours

Département Informatique

64 avenue Jean Portalis

37200 Tours, France

Tél. +33 (0)2 47 36 14 14

polytech.univ-tours.fr

Projet Recherche & Développement 2019-2020

Projet SkyEye

Archéologie & Deep Learning



POLYTECH[®]
TOURS

Tuteurs académiques

Thierry BROUARD

Jean-Yves RAMEL

Étudiant

Tom SUCHEL (DI5)



Liste des intervenants

Nom	Email	Qualité
Tom SUCHEL	tom.suchel@univ-tours.fr	Étudiant DI5
Thierry BROUARD	thierry.brouard@univ-tours.fr	Tuteur académique, Département Informatique
Jean-Yves RAMEL	jean-yves.ramel@univ-tours.fr	Tuteur académique, Département Informatique



Avertissement

Ce document a été rédigé par Tom SUCHEL surnommé l'auteur.

L'Ecole Polytechnique de l'Université de Tours est représentée par Thierry BROUARD et Jean-Yves RAMEL surnommés les tuteurs académiques.

Par l'utilisation de ce modèle de document, l'ensemble des intervenants du projet acceptent les conditions définies ci-après.

L'auteur reconnaît assumer l'entière responsabilité du contenu du document ainsi que toutes suites judiciaires qui pourraient en découler du fait du non respect des lois ou des droits d'auteur.

L'auteur atteste que les propos du document sont sincères et assume l'entière responsabilité de la véracité des propos.

L'auteur atteste ne pas s'approprier le travail d'autrui et que le document ne contient aucun plagiat.

L'auteur atteste que le document ne contient aucun propos diffamatoire ou condamnable devant la loi.

L'auteur reconnaît qu'il ne peut diffuser ce document en partie ou en intégralité sous quelque forme que ce soit sans l'accord préalable des tuteurs académiques et de l'entreprise.

L'auteur autorise l'école polytechnique de l'université François Rabelais de Tours à diffuser tout ou partie de ce document, sous quelque forme que ce soit, y compris après transformation en citant la source. Cette diffusion devra se faire gracieusement et être accompagnée du présent avertissement.



Pour citer ce document

Tom SUCHEL, *Projet SkyEye: Archéologie & Deep Learning*, Projet Recherche & Développement, Ecole Polytechnique de l'Université de Tours, Tours, France, 2019-2020.

```
@mastersthesis{
  author={SUCHEL, Tom},
  title={Projet SkyEye: Archéologie & Deep Learning},
  type={Projet Recherche \& Développement},
  school={Ecole Polytechnique de l'Université de Tours},
  address={Tours, France},
  year={2019-2020}
}
```


Table des matières

Liste des intervenants	a
Avertissement	b
Pour citer ce document	c
Table des matières	i
Table des figures	iv
1 Introduction	1
1 Contexte.....	1
2 Objectifs.....	2
3 Hypothèses	3
2 Description générale	4
1 Environnement du projet	4
2 Caractéristiques des utilisateurs.....	4
3 Fonctionnalités du système	4
4 Structure générale du système.....	5
3 État de l'art	6
1 Réseaux de neurones.....	6
1.1 Un neurone.....	6
1.2 Réseau de neurones.....	7
1.3 Entraînement.....	8
2 Segmentation et classification.....	9
3 Réseau de neurones profond.....	10

3.1	CNN.....	10
3.1.1	Convolutions et Pooling.....	10
3.1.2	Paramètres	12
3.2	Transfer Learning	12
4	Problèmes similaires	13
4.1	Utilisation du Deep Learning en imagerie satellite.....	14
4.1.1	Extraction de caractéristiques.....	14
4.2	Apprentissage artificiel sur un jeu de données déséquilibré	15
4.2.1	Undersampling ou Oversampling	15
4.2.2	Augmentation des données.....	16
4.2.3	Modification de la fonction de coût.....	17
4.2.4	Vignettage ou mini-batch sampling.....	17
5	Technologie et implémentations.....	17
5.1	Keras.....	17
6	Conclusion de l'état de l'art	18
4	Analyse et conception	19
1	Hypothèses utilisées	19
2	Spécifications	20
2.1	Fonction 1 : thumbnailCreation.....	20
2.2	Fonction 2 : CNNcharb.....	20
2.3	Fonction 3 : predictCharb	20
3	Modélisation proposée.....	20
5	Mise en œuvre	24
1	Outils et librairies utilisés	24
2	Déviations par rapport aux spécifications.....	24
2.1	Modèle	24
2.2	Implémentation	25
3	Éléments d'implémentation, choix techniques	26
3.1	Extraction des vignettes	26
3.2	Entraînement et évaluation	27
3.3	Prédictions	28
4	Principales IHM.....	28
4.1	Extraction des vignettes	28
4.2	Entraînement et évaluation	29
4.3	Prédictions	30
5	Analyse des résultats, évaluation, qualité	30

6	Bilan et conclusion	32
1	Bilan du semestre 9	32
2	Bilan du semestre 10.....	33
3	Bilan qualité de mise en oeuvre.....	33
4	Bilan gestion de projet.....	33
	Annexes	34
A	Planification, gestion de projet	35
1	Évolution du projet	35
B	Cahier de Spécification	37
C	Fiche de configuration	51
D	Manuel développeur	53
E	Manuel utilisateur	57
F	Cahier de test	62
1	Tests unitaires	62
G	Bibliographie	63

Table des figures

1 Introduction

1.1	Schéma du fonctionnement du LiDAR (C. Laplaige, 2012)	2
1.2	Chambord, modèle ombré du terrain révélant des planches de labours, des secteurs d'extraction de matériaux et des loges de bûcherons.....	2
1.3	Exemple de résultat de la première version de SkyEye (SVM)	3

2 Description générale

2.1	Diagramme des cas d'utilisations de SkyEye.....	5
2.2	Diagramme simplifié de la structure du projet SkyEye	5

3 État de l'art

3.1	Schéma simplifié d'un neurone biologique.....	6
3.2	Schéma du fonctionnement d'un neurone artificiel	7
3.3	Courbe de la fonction sigmoïde	7
3.4	Organisation des couches d'un réseau de neurones.....	8
3.5	Underfitting et overfitting	9
3.6	Différences entre la classification et la segmentation	9
3.7	Objectif de SkyEye : segmentation d'images LiDAR.....	10
3.8	Fonctionnement d'une convolution	11
3.9	Fonctionnement du max pooling	11
3.10	Architecture d'un réseau CNN	11
3.11	Images LiDAR accompagnées des masques de segmentation des talus.....	13
3.12	Architecture du réseau de classification DeepSat V2.....	14
3.13	Jeu de données équilibré ou déséquilibré	15

3.14	Undersampling et oversampling.....	16
3.15	Vignettes $32 \times 32px$ extraites à partir d'une image de $400 \times 400px$	17
4	Analyse et conception	
4.1	Architecture actuelle du CNN de segmentation de charbonnières.....	21
4.2	Exemples de résultats de l'évaluation de VGG16 (30 époques)	21
4.3	Exemples de résultats de l'évaluation du nouveau modèle (30 époques)	21
4.4	Organisation des modules Python au sein du projet	23
5	Mise en œuvre	
5.1	Prédiction réalisée par le modèle élaboré au S9	25
5.2	UML final du projet (simplifié)	26
5.3	Schéma explicatif du paramètre "mode"	27
5.4	Interface d'extraction de vignettes	29
5.5	Interface d'entraînement et d'évaluation.....	29
5.6	Interface de prédictions.....	30
A	Planification, gestion de projet	
A.1	Diagramme de Gantt prévisionnel du S9	35
A.2	Diagramme de Gantt effectif du S9	35
A.3	Diagramme de Gantt prévisionnel du S10	36
A.4	Diagramme de Gantt prévisionnel du S10	36

1

Introduction

Ce document est le rapport du projet Recherche et Développement RFAI16 : “SkyEye : Archéologie & Deep Learning”. Ce projet s’inscrit dans la continuité du programme SOLiDAR, programme qui consiste à utiliser la technologie LiDAR afin de cartographier des zones forestières qui sont difficilement analysables depuis des photographies aériennes classiques, notamment à cause de la végétation.

La maîtrise d’ouvrage est ici représentée par Thierry BROUARD et Jean-Yves RAMEL (enseignants chercheurs à l’Ecole Polytechnique de l’Université de Tours, département informatique), ainsi que par Clément LAPLAIGE, Xavier RODIER et Nathanaël LE VOGUER (chercheurs en archéologie UMR 7324 CITERES - LAT).

La maîtrise d’œuvre est ici représentée par Tom SUCHEL (étudiant en 5ème année à l’École Polytechnique de l’Université de Tours, département informatique), auteur du présent document.

1 Contexte

Le LiDAR (pour “Light Detection And Ranging”) est une technique de mesure de distance basée sur l’analyse des propriétés d’un faisceau de lumière envoyé vers un objet puis renvoyé vers son émetteur. À la différence du radar ou du sonar, le LiDAR utilise de la lumière issue d’un laser pour réaliser ses mesures (voir 1.1).

Cette technologie est utilisée dans une grande variété de domaines : topographie, météorologie, sciences de l’environnement, défense, mais aussi dans le domaine de l’archéologie, où elle présente de nombreux avantages [2]. En effet, elle permet de voir à travers la couverture végétale et offre donc une étude microtopographique du sol, en affichant des structures à la fois invisibles sur le terrain et sur des images satellites classiques (voir 1.2).

L’objectif de SOLiDAR est d’utiliser cette technologie pour améliorer la connaissance et la compréhension des dynamiques d’occupation du sol, et ainsi ouvrir la porte à de nouvelles analyses archéologiques de la région de Chambord, Boulogne, Russy et Blois [8]. Le projet comporte un nombre très important de données à segmenter, travail actuellement réalisé manuellement par les archéologues du CITERES. L’outil de segmentation automatique SkyEye a alors été développé pour aider à la décision lors de l’analyse, afin d’alléger le travail fourni par les experts.

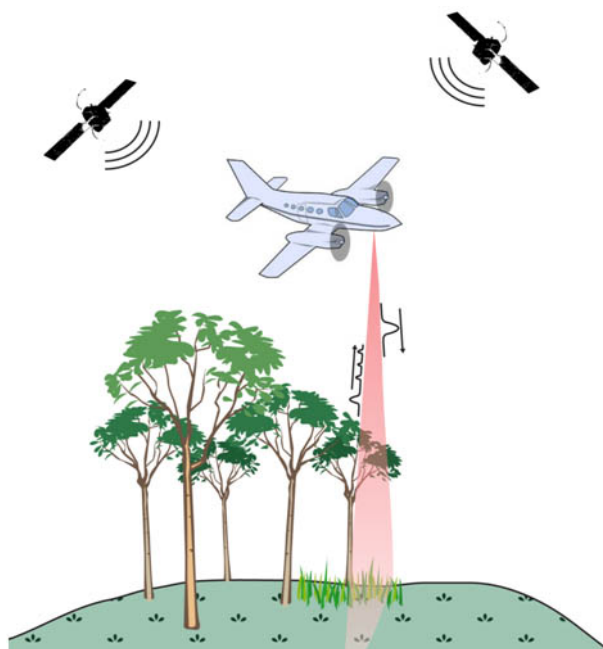


Figure 1.1 – Schéma du fonctionnement du LiDAR (C. Laplaige, 2012)

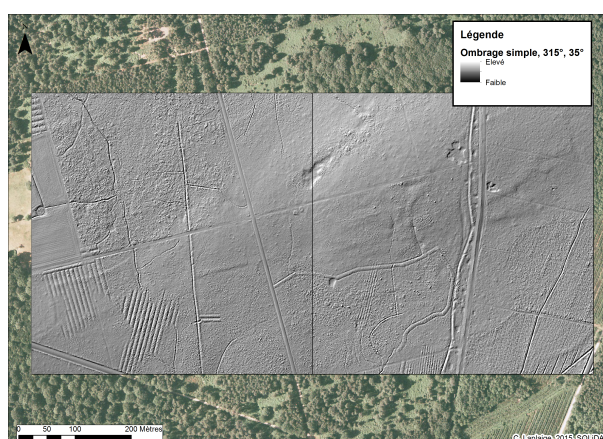


Figure 1.2 – Chambord, modèle ombré du terrain révélant des planches de labours, des secteurs d'extraction de matériaux et des loges de bûcherons

2 Objectifs

L'objectif de ce projet Recherche & Développement est l'amélioration de l'outil SkyEye, et plus précisément des méthodes de segmentation automatiques intégrées à celui-ci. Cet outil a débuté avec une classification linéaire, en utilisant un SVM (Séparateur à Vaste Marge) mais les résultats étaient peu satisfaisants, ceux-ci dépassant rarement les 70% de classification correcte.

Le projet a été ensuite repris en 2019 lors d'un PRD réalisé par Valentin MAURICE, ancien étudiant de l'École Polytechnique de l'Université de Tours. L'objectif était alors d'améliorer les performances de la segmentation à l'aide d'une approche orientée réseau de neurones profond. Valentin a fait énormément avancer le projet, en proposant une refactorisation presque complète du code et en implémentant les outils nécessaires à l'utilisation de Deep Learning (augmentation des données, entraînement d'un modèle, évaluation et prédictions). Cependant, au terme du projet, les résultats de la prédiction étaient encore trop faibles pour que le logiciel puisse servir de base solide d'aide à la décision. Néanmoins, ces résultats restent très encourageants quant à l'utilisation de réseaux de neurones profonds pour segmenter des images LiDAR.

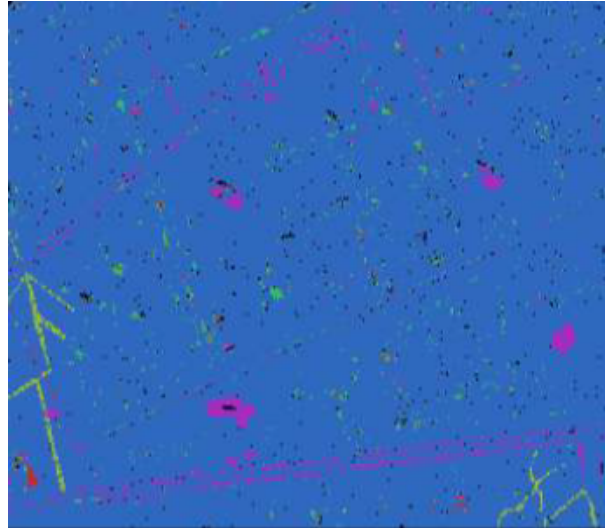


Figure 1.3 – Exemple de résultat de la première version de SkyEye (SVM)

La segmentation pixel à pixel est peu lisible et présente beaucoup de faux-positifs (zones classifiées comme appartenant à une classe qui n'est pas la bonne)

L'objectif de ce projet sera donc d'améliorer les modèles de prédiction actuels ou d'en développer de nouveaux, afin d'améliorer les performances du logiciel.

3 Hypothèses

La réalisation de ce projet dépendra de l'architecture choisie lors de la phase de recherche, donc les entrées et sorties pourront éventuellement varier par rapport à l'implémentation actuelle, même si celles-ci seront probablement les mêmes que les CNN déjà présents. L'objectif est d'implémenter le nouveau modèle de segmentation sans modifier la structure actuelle du projet. Ainsi, le logiciel prendra toujours en entrée un ensemble d'images LiDAR, et fournira en sortie des masques de segmentation et des superpositions affichant les différentes classes prédites par le CNN.

2

Description générale

1 Environnement du projet

Comme indiqué dans la partie précédente, l'objectif est d'implémenter de nouveaux modèles de segmentation, sans modifier le fonctionnement général du logiciel. Ces modèles seront d'abord développés à part, puis intégrés au programme existant.

On utilisera donc le même environnement que le projet original : le développement Python se fera avec l'IDE *PyCharm* sur Linux Ubuntu, en utilisant un certain nombre de bibliothèques nécessaires au fonctionnement de l'outil : *appdirs*, *cycler*, *matplotlib*, *numpy*, *olefile*, *opencv-python*, *packaging*, *pillow*, *pyparsing*, *pyqt5*, *python-datautil*, *pytz*, *scikit-learn*, *scipy*, *sip*, *six*.

La partie interface du projet sera gérée par *Qt*, tandis que le développement des CNN sera réalisé en utilisant la librairie *Keras*, ainsi que *Sklearn* pour les tests et la validation.

2 Caractéristiques des utilisateurs

Les utilisateurs de l'application sont des chercheurs en archéologie. On suppose qu'ils sont plutôt familiers avec l'usage de l'outil informatique, mais qu'ils possèdent seulement quelques notions, voire pas du tout, en apprentissage automatique/profond.

On veut donc une expérience utilisateur simple et intuitive, mais qui soit quand même complète pour laisser plus de choix aux utilisateurs possédant des connaissances plus poussées. On reprendra donc l'interface actuelle, qui respecte déjà tous ces points, en y intégrant les nouvelles fonctionnalités et les paramètres qui y sont liés.

3 Fonctionnalités du système

Le logiciel SkyEye servira à répondre aux cas d'utilisations affichés dans le diagramme 2.1. Toutes ces fonctionnalités sont déjà implémentées dans la version actuelle du programme, le travail consistera essentiellement en la modification de celles-ci (notamment les fonctionnalités d'entraînement, d'évaluation et de prédiction) pour qu'elles s'adaptent aux nouveaux réseaux de neurones.

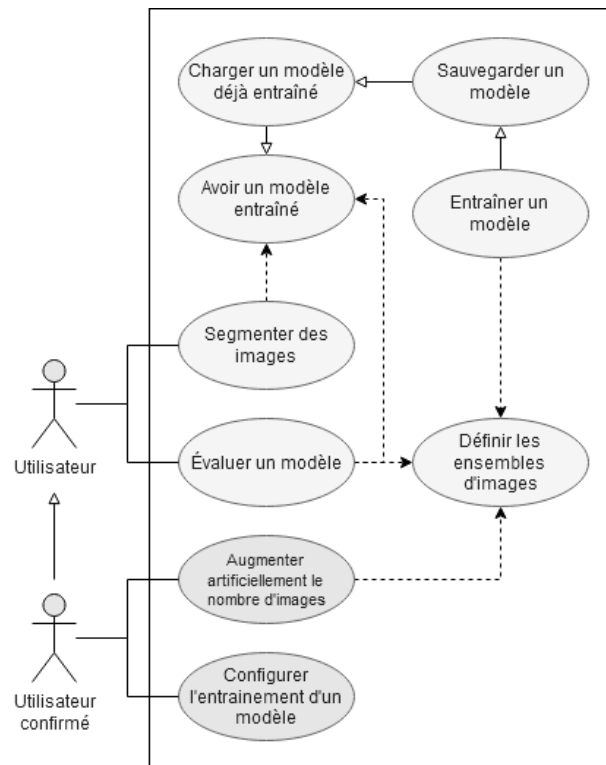


Figure 2.1 – Diagramme des cas d'utilisations de SkyEye

4 Structure générale du système

Le projet actuel est déjà structuré de manière à ce qu'il soit facilement modifiable. Ainsi, les seules modifications qui vont être faites sur la structure du projet vont être l'ajout de nouveaux modèles à la liste des modèles déjà présents dans le module *keras_segmentation* et l'ajout de nouvelles fonctions spécifiques aux modèles dans le module *skyeye*. Des modifications d'interfaces pourront éventuellement être réalisées si on veut laisser à l'utilisateur plus de personnalisation sur l'entraînement des CNN.

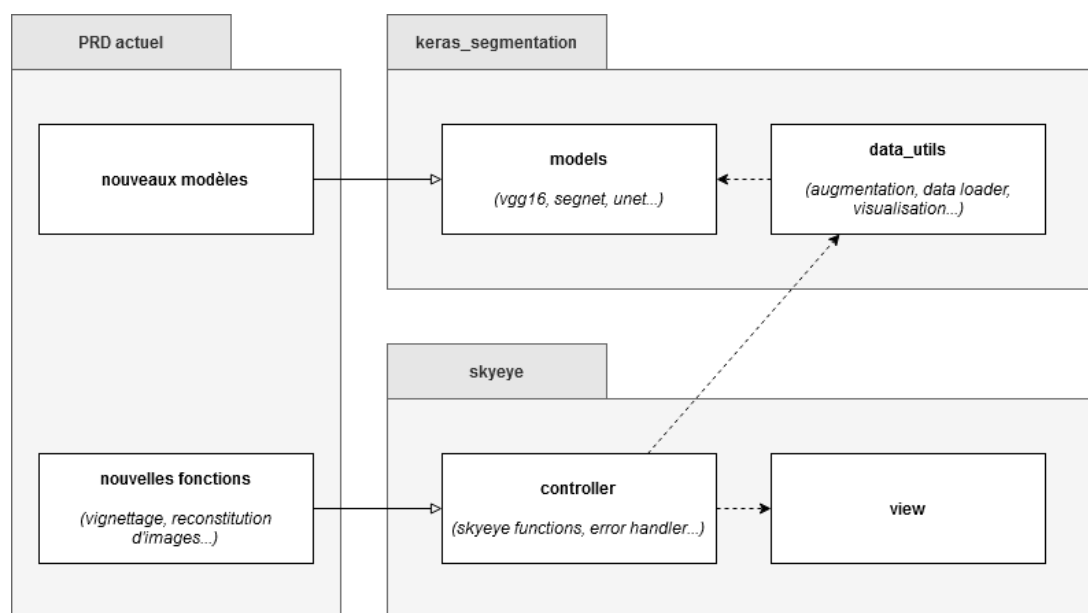


Figure 2.2 – Diagramme simplifié de la structure du projet SkyEye

3

État de l'art

1 Réseaux de neurones

Depuis la naissance du domaine de l'intelligence artificielle il a plus de 50 ans, puis plus tardivement de l'apprentissage automatique, des chercheurs ont tenté de développer des méthodes ayant pour but de donner à des ordinateurs la capacité "d'apprendre" à réaliser des décisions par eux-mêmes. Les réseaux de neurones sont des systèmes informatiques qui tentent de répondre à cette problématique en tentant d'imiter la structure du cerveau humain. Aujourd'hui, ils sont utilisés dans une multitude de domaines et offrent des résultats parfois impressionnants, bien que cette technologie soit encore en pleine évolution.

1.1 Un neurone

Un réseau de neurones est composé d'un ensemble de neurones artificiels agencés en couche, et qui interagissent les uns avec les autres. Les neurones artificiels sont inspirés des neurones biologiques, et sont entre autre une abstraction de leur fonctionnement. Pour simplifier, les neurones sont des cellules qui ont plusieurs récepteurs (les dendrites) et un unique émetteur (l'axone).

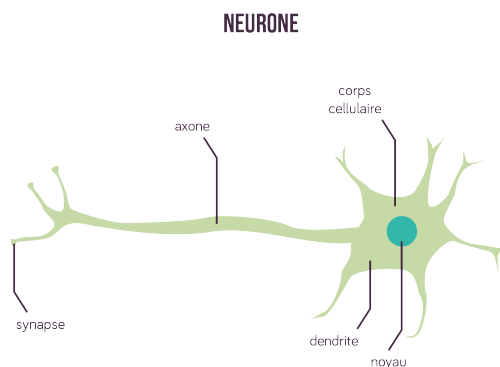


Figure 3.1 – Schéma simplifié d'un neurone biologique

Source : <https://www.schoolmouv.fr/definitions/neurones/definition>

Les neurones sont connectés entre eux via l'axone qui est relié aux dendrites de plusieurs autres neurones. Ceux-ci communiquent entre eux via des signaux qui sont envoyés lorsque le neurone "s'active". L'activation d'un neurone dépend des signaux d'entrée qu'elle reçoit, et de depuis quelle dendrite elle les reçoit. En suivant cette logique, on peut alors schématiser un neurone artificiel comme ceci :

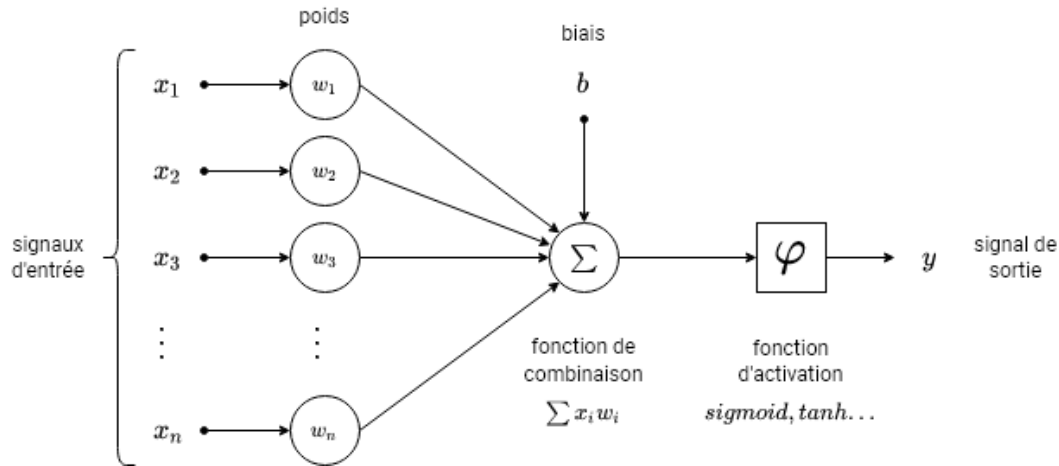


Figure 3.2 – Schéma du fonctionnement d'un neurone artificiel

Le neurone possède plusieurs **entrées** (x_1, x_2, x_3, \dots) auxquelles sont associés des **poids** (w_1, w_2, w_3, \dots) qui permettent d'ajuster l'importance de chaque signal d'entrée. Le neurone agrège ensuite tous les signaux d'entrée pour calculer ce qu'on appelle le **potentiel**. Si ce potentiel est supérieur au **seuil d'activation** du neurone, alors celui-ci s'active et propage la valeur de son potentiel aux neurones suivants, via sa **sortie** (y). On peut aussi remplacer le seuil d'activation par une **fonction d'activation**, qui gère le mécanisme de transmission du signal. Ces fonctions peuvent être la tangente hyperbolique, la sigmoïde (voir 3.3) ou encore le softmax, et transforment généralement la valeur d'entrée en un nombre compris entre 0 et 1. Enfin, un neurone peut avoir un **biais** (b), une sorte de constante prise en compte lors de l'agrégation des signaux d'entrée pour donner plus ou moins d'importance à un tel neurone.

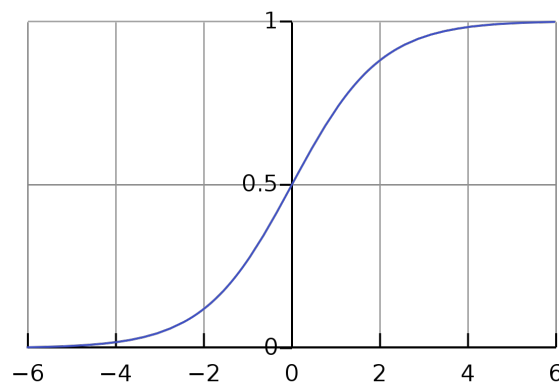


Figure 3.3 – Courbe de la fonction sigmoïde

1.2 Réseau de neurones

Un réseau de neurones est un ensemble de neurones organisé en couches, où chaque neurone d'une couche est connecté à tous les neurones de la couche suivante. On peut distinguer globalement trois types de couches :

- la couche d'**entrée**, les neurones qui vont transmettre les données d'entrée au réseau
- les couches **cachées**, qui représentent toutes les couches intermédiaires du réseau
- la couche de **sortie**, qui contient les neurones qui vont émettre les résultats du réseau

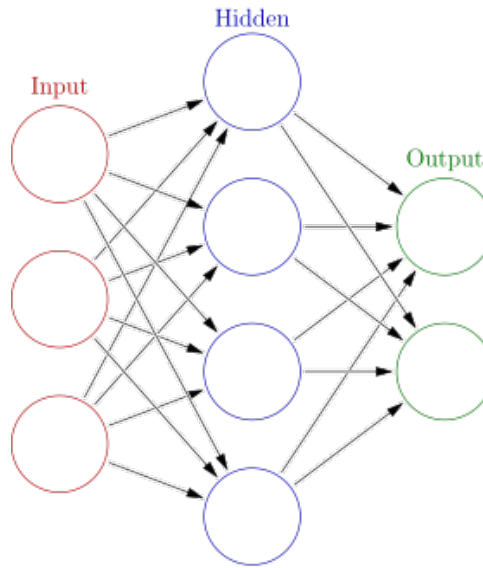


Figure 3.4 – Organisation des couches d'un réseau de neurones

Source : https://en.wikipedia.org/wiki/Artificial_neural_network

On peut donner du sens aux couches en imaginant que la couche d'entrée va représenter les données à apprendre (par exemple, chaque neurone contiendra la valeur d'un pixel d'une image). La couche de sortie pourra par exemple contenir la probabilité d'appartenance à une classe. Les couches cachées, elles, font office de boîte noire et il est parfois difficile d'interpréter leur contenu. C'est le nombre de couches cachées qui va faire augmenter la complexité du réseau et le niveau d'abstraction vers lequel il pourra se diriger.

On peut différencier les réseaux où les signaux ne se déplacent que dans un sens, à partir de la couche d'entrée vers la couche de sortie (réseaux **feedforward**) des réseaux où l'information peut boucler (RNN pour **Recurrent Neural Network**).

1.3 Entraînement

L'enjeu dans la réalisation d'un réseau de neurones consiste à trouver le bon enchaînement et paramétrage des couches cachées afin de répondre au mieux au problème posé. Une fois que l'architecture du réseau a été déterminée, les poids sont affectés à une valeur initiale (le plus souvent choisie aléatoirement), et le but va être de trouver la valeur optimale de ces poids, c'est à dire celle qui permettrait de déterminer la bonne classe de sortie à partir des données d'entrée. On peut donc voir un réseau de neurones comme une grande fonction, contenant des milliers voir millions de paramètres (les poids associés à chaque neurones) qu'on va chercher à ajuster automatiquement via une phase d'entraînement.

Pour entraîner un réseau de neurones, on a besoin d'un jeu de données labellisé, c'est-à-dire des données pour lesquelles on dispose pour chaque instance de la sortie attendue. On va alors passer ces données en entrée du réseau, qui va réaliser une prédiction. On cherchera ensuite à corriger les poids en fonction de cette prédiction.

Il existe plusieurs techniques de correction des poids, mais la plus utilisée est celle qu'on appelle la **back-propagation** (ou rétropropagation du gradient). Elle consiste à calculer la quantité

d'erreur entre la prédiction obtenue et le résultat attendu et à ajuster les poids en conséquence, de manière à converger le plus vite vers des poids optimaux.

La précision des prédictions augmente donc au fil des itérations de la phase d'entraînement (appelées époques). Cependant, il faut faire attention à certains phénomènes qui peuvent nuire au bon fonctionnement du réseau. Il se peut que le réseau fasse du “sur-apprentissage” (ou **overfitting**), c'est-à-dire qu'il apprend par coeur à reconnaître les exemples qu'on lui a fourni plutôt que d'apprendre leurs caractéristiques générales. Il existe plusieurs techniques pour éviter cela, notamment en séparant le jeu de données en deux parties : la première pour l'entraînement et la deuxième pour l'évaluation du réseau. On peut aussi faire occasionnellement “oublier” des connaissances au réseau via l'utilisation de couches de dropout. À l'inverse, il se peut que le réseau de neurones ne parvienne pas à converger vers des poids optimaux (on parle alors d'**underfitting**). Cela arrive souvent lorsque les données d'entraînement sont trop peu nombreuses ou si le réseau est mal architecturé.

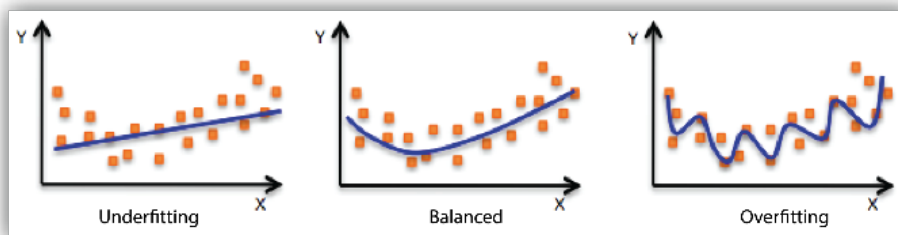


Figure 3.5 – Underfitting et overfitting

Source : <https://docs.aws.amazon.com/machine-learning/latest/dg/model-fit-underfitting-vs-overfitting.html>

2 Segmentation et classification

Le plus souvent, on utilise des méthodes d'apprentissage artificiel sur des images pour résoudre deux types de problèmes : la classification ou la segmentation. La classification a pour objectif d'identifier le contenu d'une image pour lui associer un label, tandis que la segmentation va plus loin et cherche à détecter les contours des objets contenus dans l'image. Le schéma ci-dessous résume ces deux approches.

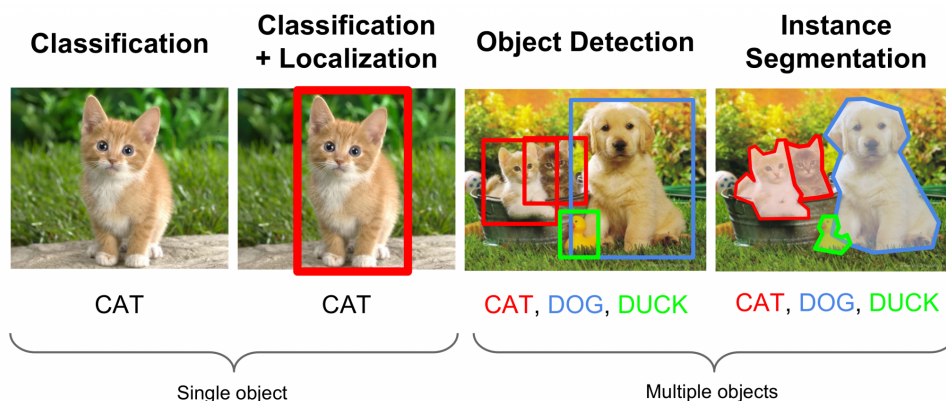


Figure 3.6 – Différences entre la classification et la segmentation

Source : J. Mathew, 2018, *Deep Learning for Image segmentation*, Medium

L'objectif du projet SkyEye est de segmenter automatiquement différentes structures archéologiques présentes dans des images de grande taille (voir 3.7), tâche actuellement réalisée à la main par les chercheurs. Cette tâche semble difficile à réaliser avec un réseau de neurones "classique" car les éléments à segmenter sont souvent de petite taille, n'ont pour la plupart pas de forme caractéristique et sont noyés dans un ensemble d'éléments qui ne nous intéressent pas (chemins, plans d'eau...). Pour tenter de résoudre ce problème, nous allons donc utiliser des réseaux de neurones à convolution, un type de réseau feedforward qui montre de très bonnes performances en reconnaissance d'image.

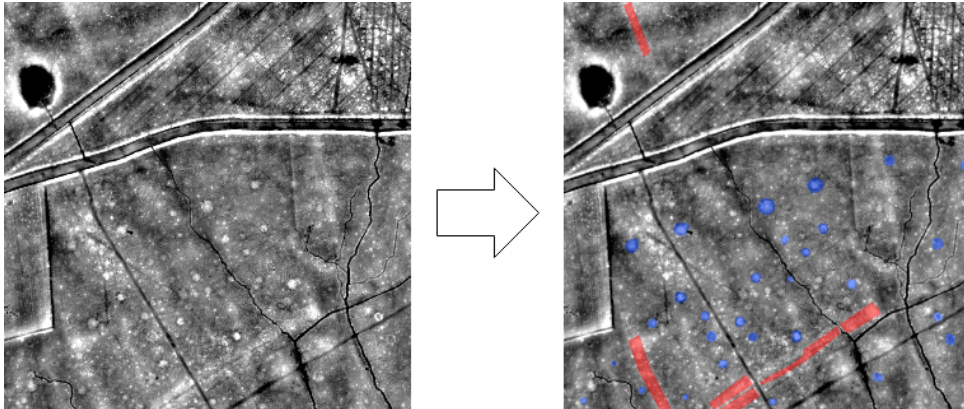


Figure 3.7 – Objectif de SkyEye : segmentation d'images LiDAR

3 Réseau de neurones profond

3.1 CNN

Un réseau de neurones à convolution (CNN) est une classe de réseaux profonds de type feedforward (voir 1.2) inspiré du fonctionnement du cortex visuel des animaux [9]. Les CNN sont très couramment utilisés en analyse d'image, mais ont aussi des applications en traitement du langage naturel ou en analyse de série temporelles. Un CNN peut être divisé en deux parties : la première est responsable de la transformation des données d'entrée et de la compression de celles-ci, tandis que la deuxième correspond à un réseau de neurones classique, chargé de l'abstraction et de la décision.

3.1.1 Convolutions et Pooling

L'objectif de la première partie d'un réseau de neurones à convolution est de transformer les données initiales (dans notre cas, les images) afin d'en extraire des caractéristiques qui ont plus de sens pour le reste du réseau. On utilise pour cela des opérations de convolution.

Une convolution consiste à appliquer une somme pondérée sur une zone de l'image à l'aide d'une matrice de poids appelée "noyau de convolution" (**kernel**). Cette technique est résumée sur l'image 3.8. En appliquant différents noyaux de convolution à l'image, on obtient alors différentes cartes de caractéristiques (**features-maps**) qui vont décrire différentes caractéristiques de l'image initiale. Plus on avance dans le réseau, plus les couches de convolution vont extraire des caractéristiques abstraites de l'image [5]. L'objectif sera alors de trouver les matrices de poids optimales qui permettent de créer une information de qualité, qui permet à la seconde partie du réseau de remplir sa fonction avec une erreur minimale. Les opérations de convolution servent donc à remplacer le raisonnement humain qui, dans le cas d'un réseau de neurones classiques, doit décider des informations à fournir à la couche d'entrée.

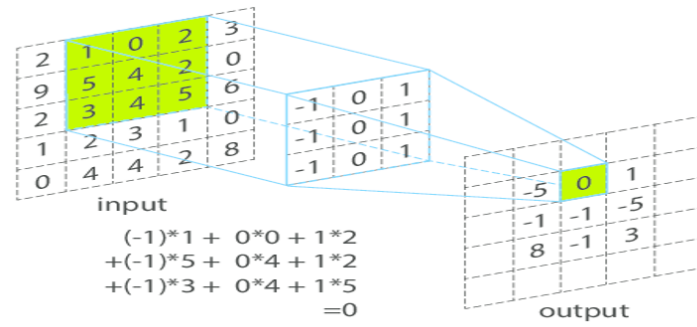


Figure 3.8 – Fonctionnement d'une convolution

Source : <https://perso.esiee.fr/~perretb/I5FM/TAI/convolution/index.html>

On peut ensuite compresser les feature-maps en utilisant des opérations de **pooling**, qui servent à diminuer la dimension des données tout en minimisant la perte d'information. On utilise le plus souvent des couches de Max Pooling, qui pour une dimension donnée, conservent la valeur maximale de chaque sous-matrice de cette dimension (voir 3.9). L'ensemble de ces couches (convolutions et pooling) forment donc la partie d'extraction de caractéristiques (**feature learning** ou feature extractor) du réseau.

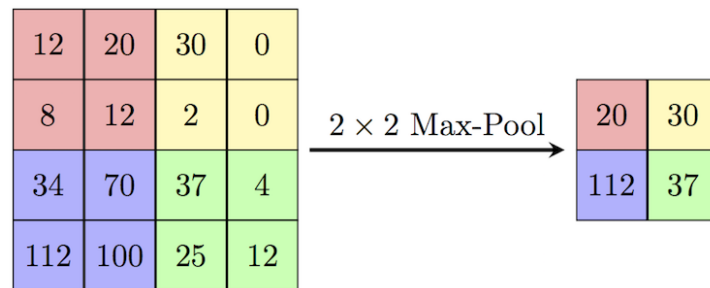


Figure 3.9 – Fonctionnement du max pooling

Source : https://computersciencewiki.org/index.php/Max-pooling/_Pooling

Pour raccorder cette partie à la seconde partie du réseau, on passe par une couche de **flattening**, qui transforme la dernière matrice de caractéristiques en un unique vecteur qu'on pourra fournir en entrée aux couches complètement connectées. Le fonctionnement général d'un CNN est résumé dans le schéma ci-dessous.

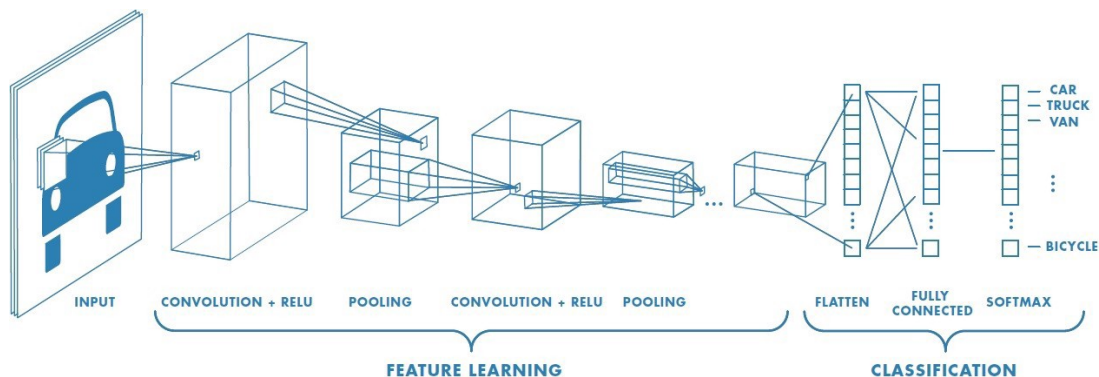


Figure 3.10 – Architecture d'un réseau CNN

Source : <https://sites.temple.edu/tudsc/2019/12/16/measuring-the-impact-of-built-environments-on-health-part-ii-use-of-transfer-learning/>

3.1.2 Paramètres

Utiliser un réseau de neurones à convolution augmente évidemment le nombre de paramètres. On peut distinguer les paramètres initiaux du réseau, fixés par le développeur (**les hyperparamètres**) des paramètres que le réseau tente de déterminer lors de la phase d'entraînement (noyaux de convolutions et poids synaptiques).

Les hyperparamètres d'un CNN sont :

Pour les couches de convolution :

- le nombre de filtres à utiliser
- la taille de ces filtres
- le pas de translation du noyau de convolution
- le padding, c'est-à-dire le nombre de pixels noirs à ajouter aux marges pour appliquer les filtres de convolution sur les bords de l'image
- une fonction d'activation, le plus souvent *ReLU*

Pour les couches de max-pooling :

- la taille des cellules
- le pas de séparation entre les cellules

Pour les couches complètement connectés :

- le nombre de neurones
- une fonction d'activation (souvent *ReLU* pour les couches intermédiaires, et *sigmoïde* ou *softmax* pour la couche de sortie)

Construire un réseau de neurones efficace consiste donc à trouver la bon bon agencement de couches et les paramètres de celles-ci afin de répondre le mieux possible au problème à résoudre.

3.2 Transfer Learning

Le transfer learning est une technique qui consiste à utiliser les connaissances apprises par un réseau de neurones pour alimenter un autre modèle. Cette technique est très utilisée avec les CNN, car elle permet de remplacer la partie extraction de caractéristiques en utilisant les connaissances acquises par un modèle entraîné sur des données similaires.

Le transfer learning permet donc de généraliser des connaissances sur un concept global (exemple : déterminer les caractéristiques les plus intéressantes d'une voiture) à un concept plus précis (des camions, des motos, etc). Ainsi, il n'est plus nécessaire de concevoir la première partie du CNN et on gagne en temps de calcul et en précision, à condition que les données sur lesquelles a été entraîné le modèle initial soient similaires aux données à reconnaître. Cette technique permet aussi de réaliser un réseau de neurones efficace qui pourra fonctionner malgré une base d'images d'entraînement de faible volume.

Il existe des centaines de modèles pré-entraînés sur internet, ou directement dans les bibliothèques comme Keras. Ces modèles sont généralement entraînés sur des bases contenant des milliers d'images, telles que ImageNet. Pour appliquer le transfer learning à notre problème de segmentation, on pourrait se tourner vers des modèles pré-entraînés directement sur des images satellite¹.

1. Voir <https://github.com/robmarkcole/satellite-image-deep-learning#land-classification>

4 Problèmes similaires

Plusieurs facteurs font que le problème à résoudre est difficile, même pour un CNN :

- On possède très peu d'exemples d'apprentissage (seulement 66 images labellisées). Lors du précédent PRD, Valentin a tenté de remédier à ce problème en développant des fonctions d'augmentation des données, qui créent artificiellement des images en réalisant des opérations sur les images initiales (rotations, zoom, décalages...). Malheureusement, l'augmentation des données n'a pas permis d'améliorer les résultats, probablement car les opérations dégradent les caractéristiques des formes à reconnaître.
- Les structures à détecter (charbonnières, tertres et talus) sont très peu représentées. Dans son rapport de stage portant sur l'utilisation de méthodes de vignettage pour améliorer les résultats de SkyEye, Florian Herguez a calculé que les images étaient composées en moyenne à 98% de fond. Ainsi, seulement 2% du nombre total de pixels correspondent à des classes à reconnaître.
- À l'exception des charbonnières, les structures à reconnaître n'ont pas forme propre. Par exemple, les talus ressemblent à des zones plus ou moins éclairées, de largeur variable, sans longueur ni orientation fixe (voir 3.11). De la même manière qu'un humain non-initié à l'archéologie, un CNN aura des difficultés à reconnaître les propriétés qui caractérisent un talus.

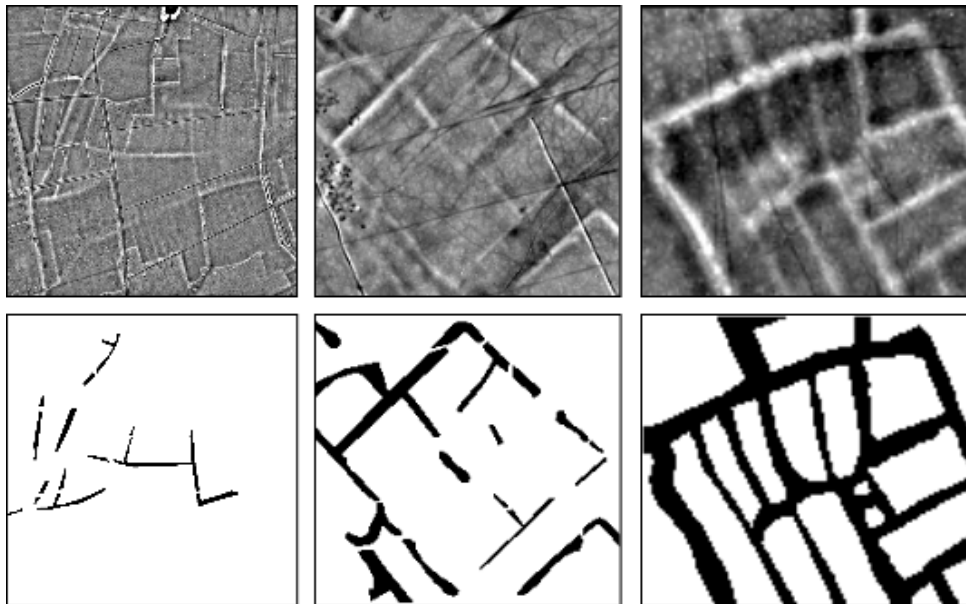


Figure 3.11 – Images LiDAR accompagnées des masques de segmentation des talus

À travers les parties suivantes, nous allons étudier les pistes proposées par la littérature scientifique pour résoudre des problèmes similaires au notre, afin de tenter de déterminer la bonne stratégie à appliquer pour atteindre l'objectif de SkyEye.

4.1 Utilisation du Deep Learning en imagerie satellite

L'imagerie satellite a toujours été importante pour de nombreuses applications : météorologie, aménagement urbain, surveillance de l'environnement, agriculture, sécurité maritime, etc. Ces applications nécessitent souvent une identification manuelle des objets contenus dans les images. Étant donné la grande taille des zones géographiques à surveiller, l'automatisation de ce procédé a vite été indispensable. Cependant, malgré une évolution constante des techniques de segmentation, le degré de précision optimal n'a toujours pas été atteint. Les méthodes d'apprentissage profond ont donc été très rapidement utilisées à cause de leur capacité à extraire automatiquement des caractéristiques et à labelliser des images avec une précision très élevée.

4.1.1 Extraction de caractéristiques

Avant l'apparition des CNN, c'était à l'utilisateur de décider des caractéristiques à fournir en entrée des réseaux de neurones. Bien que dans certains cas simples, fournir uniquement une image en noir et blanc suffise à classer un objet, il peut être très intéressant d'extraire manuellement d'autres caractéristiques à ajouter aux données d'entrées. L'utilisateur effectue alors le travail de la partie "feature-extractor" du CNN en créant ses cartes de caractéristiques lui-même.

En imagerie satellite, plusieurs caractéristiques peuvent être extraites afin d'apporter des informations supplémentaires au réseau : moyennes, variances, écart-type, contraste, entropie, homogénéité... Ces mesures ont prouvé leur efficacité en classification d'images satellites [6].

Avec l'apparition des CNN, les couches de convolution ont permis de remplacer partiellement cette étape d'extraction manuelle des caractéristiques. Cependant, il a été démontré qu'il reste intéressant de conserver ces informations et de les fusionner avec feature-maps générées par les couches de convolution, comme le propose l'architecture du réseau DeepSat V2 (voir 3.12), un des réseaux de classification d'images satellites les plus performants actuellement [Liu_2019].

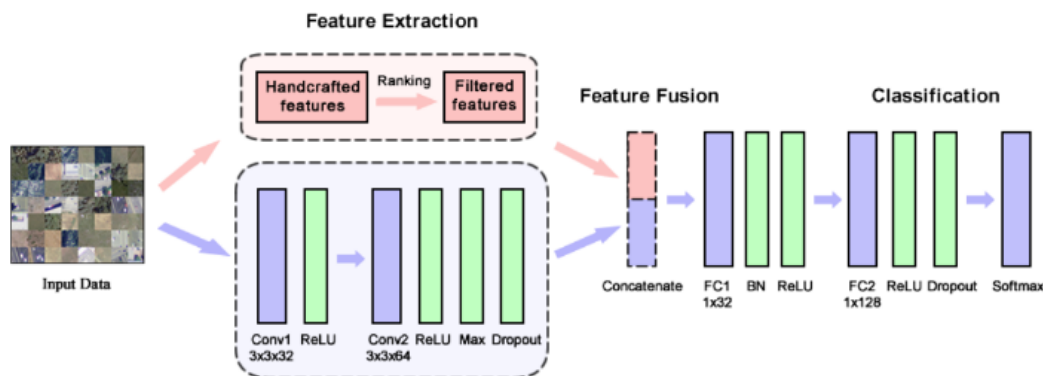


Figure 3.12 – Architecture du réseau de classification DeepSat V2

Source : Q. Liu et al, 2019. *DeepSat V2 : Feature Augmented Convolutional Neural Nets for Satellite Image Classification*

Il pourrait donc être intéressant de reproduire le même procédé dans les modèles de SkyEye, en utilisant éventuellement d'autres caractéristiques d'images mentionnés dans la littérature [10] :

- Les descripteurs de textures permettent d'extraire des caractéristiques relatives à la texture d'un objet (granularité, compacité, régularité, contraste...).
- Les descripteurs SIFT (Scale-Invariant Feature Transform) servent à décrire des sous-régions d'une scène.
- Les descripteurs HOG (Histogram of Oriented Gradients) décrivent le gradient des objets.

4.2 Apprentissage artificiel sur un jeu de données déséquilibré

Les conditions idéales pour entraîner un modèle de classification/segmentation sont d'avoir un jeu de données avec beaucoup d'images, parfaitement réparties entre les différentes classes à reconnaître. Dans les faits, ces conditions sont rarement réunies et certaines classes sont souvent plus représentées que d'autres (voir 3.13), souvent pour des raisons pratiques : constituer une base d'images labellisées est un travail long et fastidieux, et il est courant qu'une classe apparaisse très rarement dans la nature, ou bien simplement plus rarement qu'une autre. Ce problème est donc présent dans une multitude de domaines :

- Prédiction de fraudes (le nombre de fraudes est forcément bien plus faible que le nombre de transactions authentiques)
- Prédiction des catastrophes naturelles (celles-ci sont extrêmement rares et on possède donc peu d'exemples d'apprentissage)
- Détection de cellules cancéreuses (on possède beaucoup moins d'images scanner avec tumeurs que d'images sans tumeurs)

Nous allons parcourir différentes solutions évoquées dans la littérature scientifique pour tenter de pallier à ce problème.

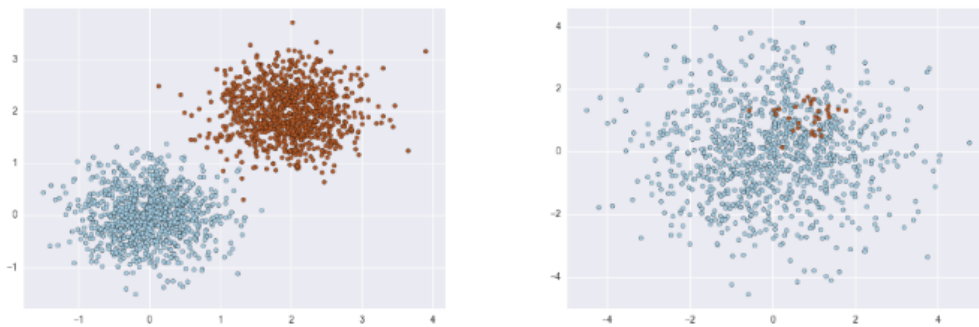


Figure 3.13 – Jeu de données équilibré ou déséquilibré

Source : <https://www.svds.com/learning-imbalanced-classes/>

4.2.1 Undersampling ou Oversampling

Lorsque l'on doit réaliser de l'apprentissage artificiel sur un jeu de données déséquilibré, deux approches très basiques peuvent être étudiées :

- L'**undersampling**, qui consiste tout simplement à supprimer aléatoirement des instances de la classe trop représentée afin d'équilibrer le ratio entre les autres classes. En réalisant cette approche, on a tout de même beaucoup de chances de diminuer le score de prédiction en supprimant des données utiles à la reconnaissance d'une classe.
- L'**oversampling**, qui consiste à augmenter les classes sous représentées en dupliquant aléatoirement des images. Cette méthode, aussi simple que la première, ne doit pas être trop utilisée car elle a le défaut de favoriser le sur-apprentissage. On utilisera alors des techniques de régularisation afin de limiter ce phénomène. Il existe plusieurs moyens d'appliquer de la régularisation à des modèles de Deep Learning : régularisation L1 ou L2 (pénalisation des paramètres via l'application de normes), couches de *Dropout* (certains poids sont remis à 0 aléatoirement lors de l'apprentissage), *Early Stopping* (arrêt du modèle dès lors qu'il ne s'améliore plus pour favoriser les modèles simples)...

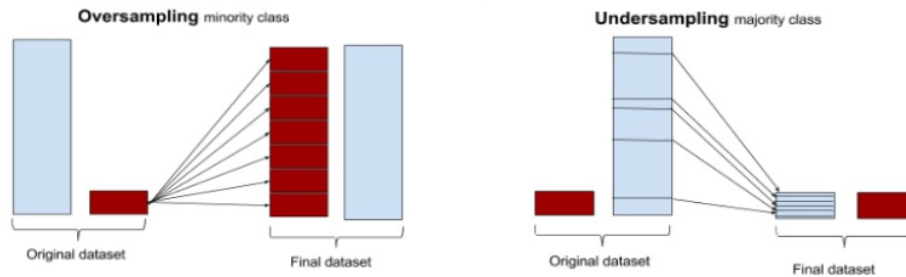


Figure 3.14 – Undersampling et oversampling

Source : <https://www.svds.com/learning-imbalanced-classes/>

4.2.2 Augmentation des données

L'augmentation artificielle des données (ou SMOTE pour Synthetic Minority Oversampling TEchnique) consiste à générer des exemples synthétiques de la ou les classes peu représentées pour réduire le déséquilibre entre les différentes classes. Cette stratégie, utilisée de paire avec la méthode d'undersampling, a montré à plusieurs reprises qu'elle permettait d'améliorer les performances d'un modèle en cas de déséquilibre du set d'apprentissage [11] [0].

L'idée derrière l'approche SMOTE est de créer de nouveaux exemples d'apprentissage en interpolant les exemples existants : pour chaque point de la classe peu représentée, on choisit ses k plus proches voisins et on crée de nouvelles instances en moyennant le point et ses voisins. En pratique, il existe des méthodes d'augmentation d'images développées au sein des bibliothèques de machine learning. Ces méthodes permettent, à partir d'un ensemble d'images de départ, de créer de nouvelles images en appliquant différentes opérations² :

- décalage horizontal et/ou vertical
- retournement horizontal et/ou vertical
- rotations
- changement de luminosité
- zoom

Ces opérations doivent être choisies avec précaution en fonction des caractéristiques des images à augmenter. Par exemple, l'orientation d'une image peut avoir du sens dans la reconnaissance de celle-ci, et effectuer des retournements/rotations pourra dégrader cette caractéristique. Dans le cas d'images satellites, l'orientation n'a aucune importance car les images sont prises en vue aérienne (à plat), on peut donc effectuer ces opérations pour multiplier le nombre d'images. À l'inverse, appliquer du zoom ou des changements de luminosité pourra significativement changer le sens de l'image.

En plus de ces différentes opérations, il est nécessaire de choisir le mode de remplissage des zones "vides" des images créées. En effet, après un décalage ou une rotation, il faut remplir certaines parties de l'image qui ont disparu après la modification. Il existe pour cela plusieurs modes :

- constant : remplissage avec une valeur fixe (kkkkkkkk|abcd|kkkkkkkk)
- nearest : remplissage avec les valeurs voisines (aaaaaaaa|abcd|dddddddd)
- reflect : remplissage avec un miroir de l'image (abcd dcba|abcd|dcba abcd)
- wrap : remplissage en copiant l'image (abcdabcd|abcd|abcdabcd)

Comme pour les opérations, il est nécessaire de choisir le mode de remplissage avec précaution car certains choix pourront affecter lourdement la reconnaissance.

2. Description des différentes opérations : <https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/>

4.2.3 Modification de la fonction de coût

Dans la plupart des frameworks de machine learning, il est possible d'affecter une matrice de poids aux différentes classes afin d'influencer l'importance donnée à une classe sous représentée. Ces poids seront pris en compte lors du calcul de l'erreur, de sorte à ce qu'un faux positif sur la classe sous représentée affecte beaucoup plus le score d'erreur. On pourra alors ré-équilibrer artificiellement l'ensemble de données sans pour autant avoir à ajouter ou supprimer des images.

4.2.4 Vignettage ou mini-batch sampling

Une autre approche souvent évoquée dans la littérature est de sectionner les images d'entraînement en **vignettes** (ou patches) de petite taille (32x32 ou 64x64). De cette manière, on augmente significativement le nombre d'images (jusqu'à une vignette par pixel de l'image source). Un autre avantage de cette méthode est que les vignettes, de plus petite taille, mettent plus en évidence le voisinage proche de l'objet à reconnaître (voir vignettes de charbonnières 3.15). On peut aussi facilement réguler l'équilibre du set d'entraînement en supprimant des vignettes composées uniquement de la classe trop représentée.

Ce type d'échantillonnage en mini-batch - prenant des vignettes à partir d'un ensemble diversifié d'images d'entraînement - s'est avéré efficace dans des problèmes de classification de pixels tels que la détection de bord et la segmentation d'image [1]. De plus, cette méthode a déjà été expérimentée par Florian Herguez lors de son stage sur le projet SkyEye. Bien que celle-ci n'ait pas montré de résultats probants (probablement car l'entraînement a été réalisé sur des modèles non-adaptés aux vignettes), cela reste une piste à explorer pour améliorer les prédictions.

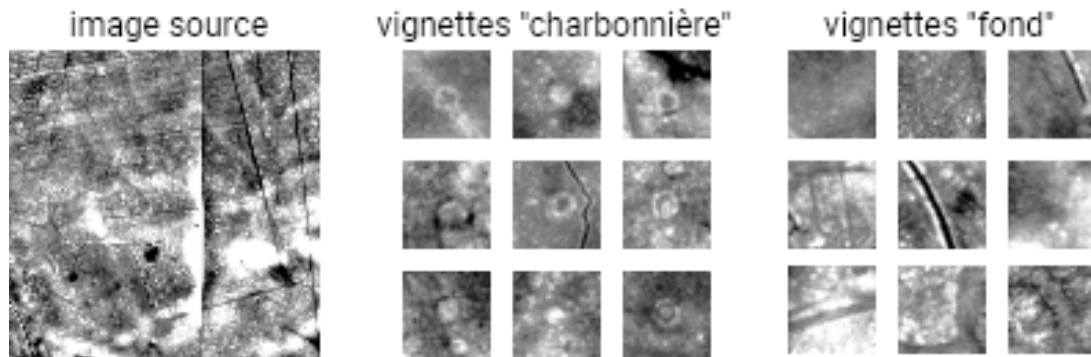


Figure 3.15 – Vignettes $32 \times 32px$ extraites à partir d'une image de $400 \times 400px$

5 Technologie et implémentations

5.1 Keras

Keras est une librairie haut niveau écrite en Python, qui permet de mettre en place rapidement tous types de réseaux de neurones. Elle permet d'interagir facilement avec des algorithmes d'apprentissage artificiel, notamment ceux implémentés dans l'outil TensorFlow. En plus de cela, elle comporte plusieurs fonctions facilitant la création de modèles d'apprentissage : lecture et augmentation des données, transfer-learning, évaluation et prédictions...

6 Conclusion de l'état de l'art

Les articles étudiés dans l'état de l'art confirment que l'utilisation de Deep Learning est généralement une bonne solution au problème de segmentation d'images satellites, comme le montrent les différents réseaux de neurones présentés dans la partie 4.1.

Cependant, il ne faut pas ignorer que la source principale de notre problème est le manque de données et la répartition déséquilibrée des classes au sein de celles-ci. On tiendras alors compte des solutions évoquées dans la partie 4.2 de l'état de l'art, notamment l'utilisation de vignettage qui semble très prometteuse. En effet, cette technique permettrait de multiplier les exemples d'apprentissage tout en régulant la proportion des différentes classes.

L'objectif serait alors d'utiliser Python et Keras pour développer un nouveau réseau de neurones convolutif adapté à des vignettes de petites taille, en s'inspirant des modèles précédemment utilisés en segmentation d'image satellite.

4

Analyse et conception

1 Hypothèses utilisées

L'analyse du problème s'est faite en deux parties :

Dans un premier temps, j'ai tenté de comprendre le stade actuel du projet, en vérifiant les systèmes mis en place par Valentin lors de son PRD et par Florian lors de son stage. J'ai réalisé plusieurs dizaines de tests sur les modèles implémentés sur SkyEye, sur des images classiques ou en utilisant des vignettes extraites par un script Python. Cette étape m'a permis à la fois de prendre en main le logiciel et la technologie, mais aussi de mieux appréhender le problème de segmentation et d'en comprendre les subtilités.

Lors d'une réunion avec mon encadrant, T. Brouard, et deux chercheurs du CITERES (C. Laplaige et N. Le Voguer), nous avons convenu que la segmentation des talus était un problème très complexe de par leur géométrie variable, et nous avons donc décidé de nous concentrer d'abord sur la détection des charbonnières. Pour cela, nous allons utiliser des techniques de classification appliquées à des vignettes, dans le but final de faire de la segmentation : en effet, on peut entraîner le modèle à donner une classe à une vignette, en assumant que cette classe représente celle du pixel central de la vignette. Il suffit ensuite de prédire la classe de chaque vignette de l'image source pour reconstituer un masque de segmentation. Cette approche du problème de classification est déjà utilisée dans d'autres réseaux de neurones tels que les R-CNN (pour Region-CNN) [4]. Ces modèles fonctionnent en extrayant des "régions d'intérêt" à partir des images (similaires à nos vignettes) et en les envoyant dans un CNN classique qui se charge de la classification de la région. Cela permet ainsi de classifier (voir segmenter) plusieurs objets différents au sein d'une même image.

À ce stade du projet, j'ai commencé à lister les fonctionnalités qui devront être réalisées au terme du PRD. Actuellement au nombre de trois, celles-ci pourront être amenées à évoluer au S10 en fonction du temps restant. L'objectif principal est actuellement de réussir à segmenter les charbonnières avec une bonne précision. Si cet objectif est atteint, on pourra alors tenter généraliser les modèles aux autres structures archéologiques.

2 Spécifications

La plupart des fonctions utiles à la gestion d'un réseau de neurones à convolution (entraînement, évaluation, sauvegarde et chargement de modèle...) sont déjà présentes dans le logiciel de base. Les fonctions développées lors de ce projet servent donc à adapter le nouveau modèle aux fonctions existantes. Les parties ci-dessous présentent un résumé des fonctionnalités attendues au terme de ce PRD. Pour consulter la description complète des fonctionnalités, se référer au cahier des spécifications en annexes [B](#).

2.1 Fonction 1 : thumbnailCreation

Cette fonction permet de créer un ensemble de vignettes de petite taille à partir d'un ensemble d'images. La fonction est divisée en deux sous-fonctions : la première, utilisée avant l'entraînement d'un modèle, crée les vignettes et les organise dans une structure de dossiers en fonction de leur classe (en l'occurrence : fond et charbonnière). La deuxième, utilisée avant une prédiction, crée simplement les vignettes et les classe par image source.

2.2 Fonction 2 : CNNcharb

Cette fonction représente le réseau de neurones à convolution développé pour classifier les différentes vignettes. Le modèle est décrit en détail dans la section [3](#).

2.3 Fonction 3 : predictCharb

Cette fonction permet de construire des masques de segmentation à partir des vignettes créées avec la fonction 1. Pour chaque dossier contenant des vignettes (donc pour chaque image), elle prédit la classe des vignettes et crée un masque de segmentation de la même taille que l'image originale. Ces masques de segmentation sont ensuite superposés aux images pour un rendu plus lisible.

3 Modélisation proposée

Lors de la seconde moitié du S9, j'ai travaillé sur l'élaboration d'un réseau de neurones à convolution spécialisé dans la classification de charbonnières. Pour cela, je me suis inspiré de l'architecture du VGG16, un modèle réputé pour ses performances en classification d'images, que j'ai simplifié pour l'adapter à la petite taille de mes images d'entrées (un VGG16 travaille habituellement avec des images d'entrées de 224 x 224 pixels). Cette simplification du modèle avait aussi pour objectif d'améliorer les performances du réseau : en effet, un réseau doit être conçu en considérant la quantité de données disponible. Un modèle tel que VGG16 possède plus de 30 millions de paramètres à apprendre, et a donc besoin d'un très grand volume de données pour optimiser tous ses poids. Ce problème peut avoir pour conséquence deux phénomènes :

- Soit le réseau va utiliser ses paramètres pour apprendre tous les exemples d'apprentissage plutôt que leurs caractéristiques, ce qui va donc résulter en du **sur-apprentissage**.
- Soit le réseau ne va pas parvenir à converger vers une solution optimale à cause du trop grand nombre de poids par rapport au faible volume de données. On parle alors du problème de **vanishing gradient** : au bout d'un certain nombre d'époques, la valeur du gradient, c'est-à-dire de la correction à appliquer sur les poids, devient extrêmement faible, ce qui empêche le réseau de s'améliorer.

Le modèle créé est résumé dans le schéma ci-dessous (couches de flattening et de dropout ignorées). À la différence de VGG16, il ne contient que 5 couches de convolution (contre 16), et 270.000 paramètres entraînables (contre 33 millions). Le modèle a été ensuite entraîné sur un ensemble d'entraînement contenant 4000 vignettes de fond et de charbonnières, équitablement réparties, afin de déterminer les meilleurs hyperparamètres.

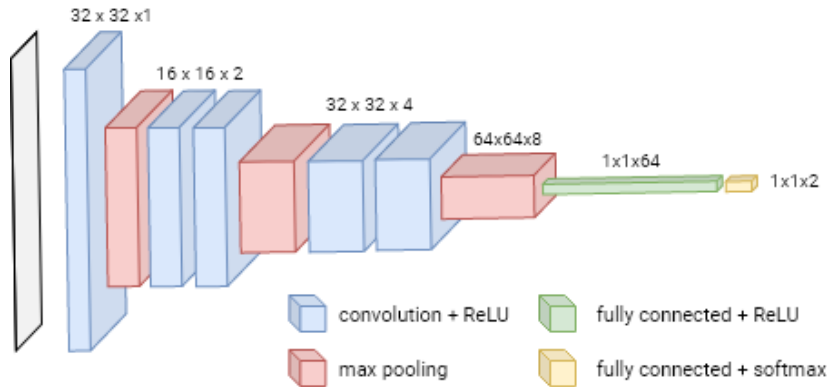


Figure 4.1 – Architecture actuelle du CNN de segmentation de charbonnières

Après plusieurs phases d'entraînement et de corrections, le réseau permet d'atteindre jusqu'à **96%** de précision sur le fond et les charbonnières, bien mieux que les précédentes implémentations. Pour tester les performances du modèle, on réalise des prédictions sur un jeu de test que le modèle n'a jamais vu auparavant, et on utilise plusieurs indicateurs statistiques : précision, rappel, f1-score, matrices de confusions, etc. Ces indicateurs permettent de vérifier la qualité des résultats obtenus, en montrant par exemple que le réseau est aussi efficace pour reconnaître du fond que des charbonnières. Les deux captures d'écran ci-dessous montrent la différence de résultats entre le nouveau modèle et le VGG16, lorsque ceux-ci sont entraînés dans les mêmes conditions (mêmes jeu de données, 30 époques). On peut aussi noter que l'entraînement du VGG16 dure plus longtemps que celui du nouveau modèle (44 minutes contre 18 minutes).

	precision	recall	f1-score	support
background	0.00	0.00	0.00	400
charbonnière	0.52	1.00	0.69	441
accuracy			0.52	841
macro avg	0.26	0.50	0.34	841
weighted avg	0.27	0.52	0.36	841

Figure 4.2 – Exemples de résultats de l'évaluation de VGG16 (30 époques)

	precision	recall	f1-score	support
background	0.90	0.85	0.88	400
charbonnière	0.87	0.92	0.90	441
accuracy			0.89	841
macro avg	0.89	0.89	0.89	841
weighted avg	0.89	0.89	0.89	841

Figure 4.3 – Exemples de résultats de l'évaluation du nouveau modèle (30 époques)

Maintenant que le *proof of concept* a été réalisé, il va falloir mettre en place tout le système d'apprentissage et de prédiction, c'est-à-dire les différentes fonctions qui vont assurer le lien entre l'utilisateur et le réseau de neurones. Ce système peut être résumé selon les cas suivants :

Entraînement :

- L'utilisateur renseigne le chemin vers les images d'entraînement et leurs segmentations.
- À partir de ces images, le logiciel extrait des vignettes et les trie par classe (**Fonction 1**).
- Le modèle est entraîné selon les paramètres renseignés par l'utilisateur : nombre d'époques, nombre d'étapes par époque, taille du batch... (**Fonction 2**).
- Une fois l'entraînement terminé, le modèle est sauvegardé pour pouvoir être réutilisé.

Évaluation :

- L'utilisateur renseigne un dossier contenant des images d'évaluation (et leurs segmentations).
- À partir de ces images, le logiciel extrait des vignettes et les trie par classe (**Fonction 1**).
- Le modèle est évalué sur cet ensemble de vignettes.
- Une fois l'évaluation terminée, on affiche différents indices de qualité dans les logs.

Prédictions :

- L'utilisateur renseigne un dossier contenant des images à segmenter.
- À partir de ces images, le logiciel extrait des vignettes et les trie en fonction de leur image source (**Fonction 1**).
- On utilise le modèle précédemment entraîné pour prédire la classe de chaque vignette et construire les masques de segmentation (**Fonction 3**).
- À partir des masques de segmentation, on crée des superpositions et on enregistre le tout.

Ainsi, comme on peut l'observer dans la figure 4.4, le seul fichier à créer sera l'implémentation du modèle de segmentation de charbonnières. Le reste du développement consistera en des modifications et des ajouts de fonctions sur des fichiers existants :

- *skyeye_func.py* : modifications des classes TrainWorker, EvalWorker et PredictWorker pour ajouter le cas d'utilisation du nouveau modèle et les exceptions qui y sont liées.
- *main_window.ui* : modifications de l'interface directement depuis QtDesigner pour ajouter la fenêtre de paramétrage du vignettage des images. Le fichier *view.main_window.py* est ensuite généré automatiquement avec la commande `pyuic5 main_window.ui > main_window.py`.
- *controller.main_window.py* : modifications à prévoir pour relier les nouveaux signaux générés par l'interface.
- *all_models.py* : ajout du nouveau CNN.
- *predict.py* et *train.py* : adaptation des fonctions d'entraînement, évaluation et prédiction dans le cas d'utilisation du nouveau CNN.
- *data_loader.py* : adaptation des fonctions de chargement d'images dans le cas d'utilisation de vignettes.

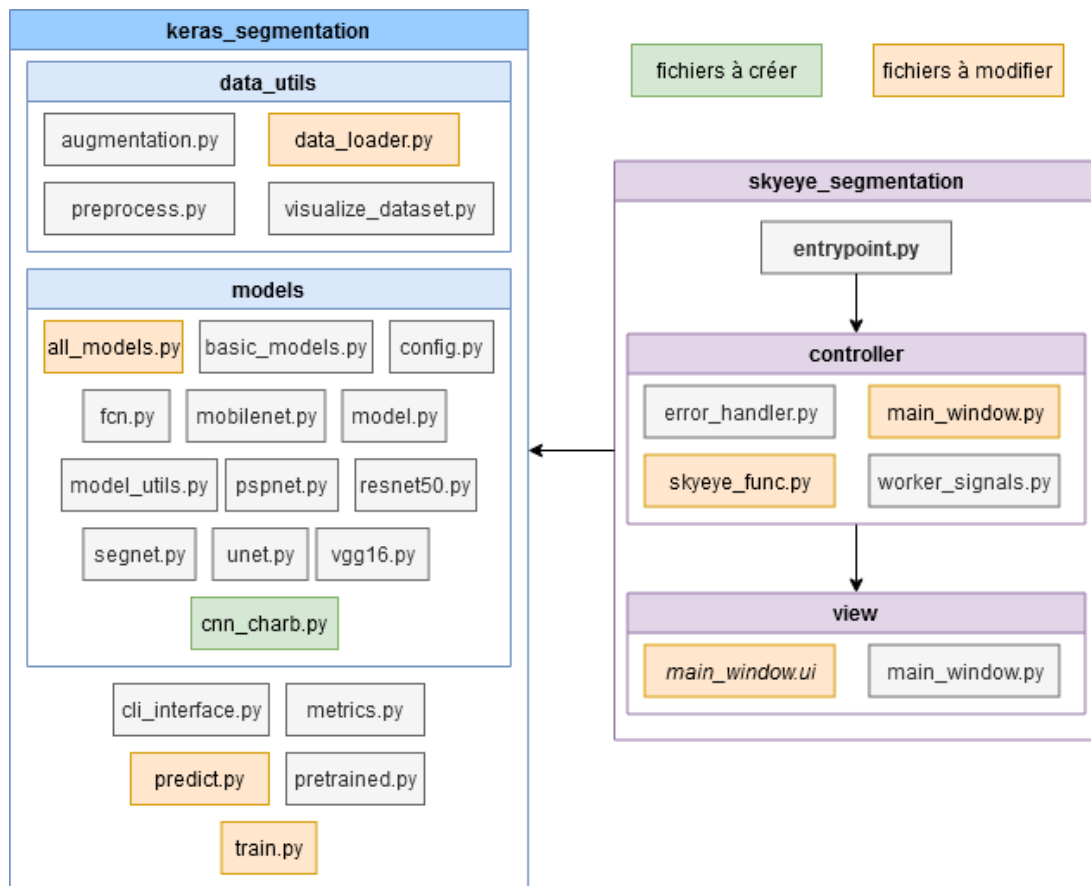


Figure 4.4 – Organisation des modules Python au sein du projet

5

Mise en œuvre

1 Outils et librairies utilisés

Tout au long de ce projet, j'ai utilisé les outils suivants :

- *PyCharm* pour le développement Python
- *Qt Designer* pour le design des interfaces
- *Atom* pour manipuler divers fichiers de configuration
- *Pytest*, *Pylint* et *Pydoc* pour gérer les tests, la qualité du code et générer la documentation
- *GitHub* pour gérer le versionning, la distribution et les manuels utilisateurs et développeurs
- *Trello* pour la gestion de projet
- *Draw.io* pour les diagrammes et les mockups

Les différentes librairies utilisées au sein du projet sont listées en annexe, dans la fiche de configuration du logiciel (D).

2 Déviations par rapport aux spécifications

2.1 Modèle

Au début du semestre 10, j'ai continué à tester et à évaluer le modèle élaboré à la fin du semestre 9, cette fois ci en reconstruisant les masques de segmentation des images LiDAR. Je me suis alors rendu compte que le modèle n'était pas aussi performant que ce que j'avais cru jusqu'à présent.

En effet, en reconstruisant des masques de segmentation à partir des résultats obtenus en prédisant la classe de chaque pixel d'une image, on obtient des scores de précision et de rappel bien plus bas que ceux obtenus à la fin du S9. En moyenne, le f1-score mesuré directement sur les segmentations dépasse rarement les 40%. On peut observer ce phénomène sur une "bonne" segmentation prédite avec le modèle (voir 5.1) : plusieurs charbonnières ne sont pas du tout reconnues (en bleu sur l'image) et on constate quelques fausses prédictions (en rouge).

Cet "incident" m'a poussé à modifier mon planning du S10 et de consacrer plus de temps au développement du modèle, afin de tester plusieurs pistes évoquées lors de réunions avec mon tuteur :

- Utilisation d'un plus grand ensemble d'apprentissage
- Changement de la taille des vignettes
- Utilisation de modèles plus profonds (VGG16, FasterCNN...)
- Post-processing pour éliminer les prédictions peu cohérentes (pixels rouges sur 5.1)
- Détermination des meilleurs hyperparamètres

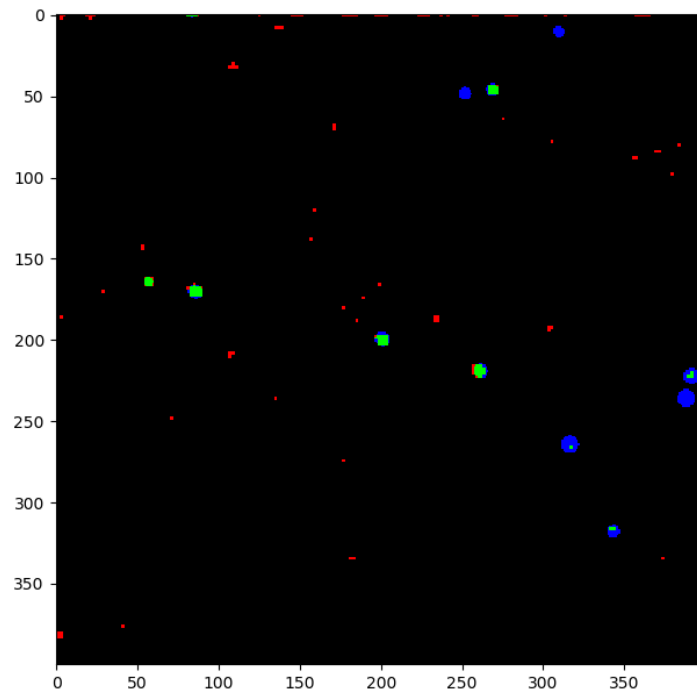


Figure 5.1 – Prédiction réalisée par le modèle élaboré au S9

Vert : charbonnière détectée comme charbonnière (TP)
Rouge : fond détecté comme charbonnière (FP)
Bleu : charbonnière détectée comme fond (FN)
Noir : fond détecté comme fond (TN)

2.2 Implémentation

La partie concernant l'implémentation du modèle dans le logiciel existant a très peu évolué entre le S9 et le S10. Sur les trois fonctions évoquées dans le cahier des spécifications, seule la fonction de prédiction a vu son comportement changer : initialement, cette fonction commençait par extraire les vignettes pour les enregistrer sur le disque, puis prédisait leur classe afin de reconstruire le masque de segmentation de l'image à prédire. Désormais, tout ce procédé se fait de manière linéaire, sans jamais enregistrer les vignettes dans la mémoire. Pour chaque image à prédire, la fonction se charge d'extraire des "batches" de vignettes, qu'elle prédit, puis elle reconstruit simultanément la segmentation à partir des résultats. Le procédé est donc plus rapide, et moins gourmand en mémoire vive et en espace disque.

Concernant le diagramme UML affiché plus haut (voir 4.4(Chapitre 4)), peu de modifications ont été réalisées. L'objectif a été d'implanter les nouvelles fonctions sans trop modifier le code existant.

- Dans /view : *main_window.ui* et *main_window.py* : Mise à jour de l'interface avec Qt Designer (ajout d'un nouvel onglet "Charbonnières")
- Dans /controller : *main_windows.py* : Ajout de nouveaux slots Qt gérant les actions des différents boutons rajoutés à l'interface.
- *skyeeye_func.py* : Ajout de quatre nouvelles classes "Worker" gérant l'extraction des vignettes, l'entraînement et l'évaluation du modèle, et la prédiction d'images.
- *charb_models/models.py* : Nouveau fichier, contenant la déclaration des modèles spécialisés en segmentation de charbonnières.

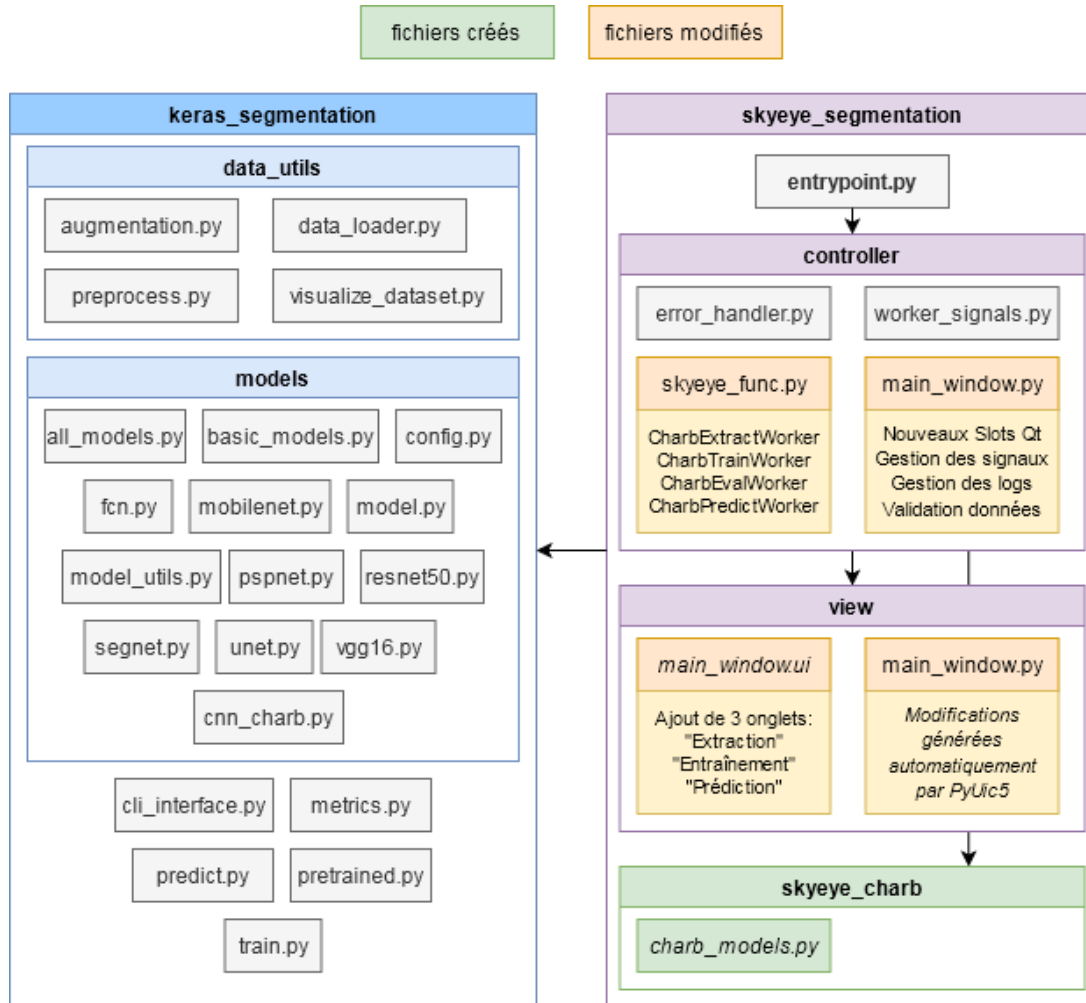


Figure 5.2 – UML final du projet (simplifié)

3 Éléments d'implémentation, choix techniques

3.1 Extraction des vignettes

Le module d'extraction de vignettes est géré dans la classe *CharbExtractionWorker* et fonctionne de la manière suivante :

- Une fonction *extract_thumbnails* prend en paramètre une image source ainsi que sa segmentation.
- La fonction commence par ajouter des bords noirs à l'image afin de pouvoir extraire des vignettes sur les contours.

- Ensuite, l'image est parcourue et des vignettes sont extraites tous les X pixels (modifiable via le paramètre "intervalle").
- Pour déterminer la classe d'une vignette, on regarde la classe du pixel central de sa segmentation (ou des 4 pixels centraux, voir schéma ci-dessous).
- Les vignettes sont ensuite enregistrées dans les dossiers *back* ou *charb*, en fonction de leur classe.

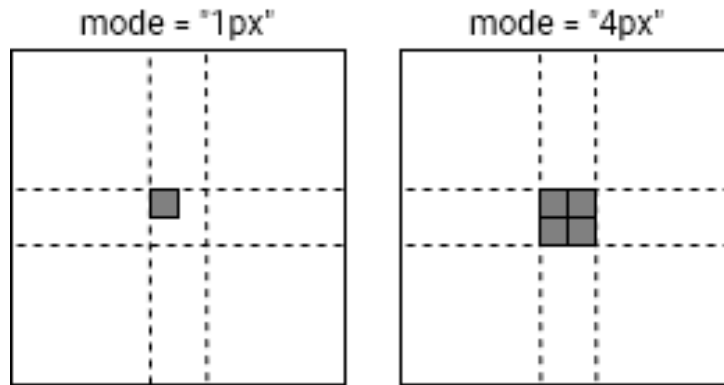


Figure 5.3 – Schéma explicatif du paramètre "mode"

Ce procédé est ensuite répété pour chaque image du dossier source, en prenant soin de répartir les images dans les ensembles d'apprentissage et d'évaluation, selon les proportions définies par l'utilisateur.

De façon évidente, on disposera moins de vignettes de charbonnières que de vignettes de fond (en l'occurrence, on peut extraire au maximum 43 279 vignettes de charbonnières à partir des 78 images LiDAR disponibles). Un système permet donc de limiter la création de vignettes de fond, afin de ne pas surcharger l'espace disque (il serait en théorie possible d'extraire jusqu'à 11 millions de vignettes de fond, soit 8Go de données).

3.2 Entraînement et évaluation

Le système d'entraînement du réseau de neurones est géré par la classe *CharbTrainWorker*.

Dans un premier temps, le modèle demandé par l'utilisateur est généré puis compilé. Les images du set d'entraînement sont chargées en mémoire au fur et à mesure en utilisant un *ImageDataGenerator*, qui forme des batches et les fournit au réseau lorsque celui-ci en a besoin. À la fin de chaque époque de l'entraînement, on évalue le modèle sur la base d'évaluation et si la précision a augmenté par rapports aux dernières itérations, alors les poids du modèle sont sauvegardés.

La classe *CharbEvalWorker* permet de tester un modèle déjà entraîné sur un jeu d'évaluation. Après un court instant où les vignettes sont prédites par le modèle, les résultats sont comparés à la vérité terrain et plusieurs métriques sont affichés à l'utilisateur : matrice de confusion, précision, rappel, f-score... (voir exemple ci-dessous).

1	PREDICTION	GROUND TRUTH			
2		background	charbonnière		
3	background	387	54		
4	charbonnière	19	352		
5					
6		precision	recall	f1-score	support
7	background	0.88	0.95	0.91	406
8	charbonnière	0.95	0.87	0.91	406

3.3 Prédictions

La difficulté dans le système de prédictions a été de traiter les images par batchs afin d'accélérer le traitement. Ainsi, la classe *CharbPredictWorker* embarque une fonction *predict* qui prend en entrée un modèle déjà entraîné, une image à prédire ainsi qu'un ensemble de paramètres (taille de batch, taille des vignettes, intervalle...). Cette fonction est similaire à la fonction d'extraction de vignette : d'abord, celle-ci ajoute des bords noirs à l'image afin de pouvoir extraire les vignettes dans les contours, puis l'image est parcourue et les vignettes sont extraites en batchs, qui sont prédits par le modèle. Les résultats de ces prédictions, associés à la position des vignettes, permettent de reconstruire la segmentation. Enfin, à partir de cette segmentation, on pourra représenter le résultats sous différentes formes : masque binaire (0=fond, 1=charb), superposition avec l'image LiDAR, carte de température, etc.

4 Principales IHM

Trois interfaces ont été rajoutées au logiciel développé par Valentin Maurice lors de son PRD sur le même sujet [7]. Ces trois fenêtres sont disponibles sous un nouvel onglet "Charbonnières", ce qui permet de séparer les nouvelles fonctions des fonctionnalités déjà implémentées dans le projet, car celles-ci ont un but différent et n'oeuvrent pas de la même manière. J'ai néanmoins essayé de conserver le design et la logique imaginés par Valentin afin de ne pas perturber les utilisateurs déjà habitués à l'outil.

4.1 Extraction des vignettes

L'interface d'extraction de vignette est très simple : il suffit de spécifier le chemin des images sources ainsi que de leurs segmentation, en précisant l'ID qui représente les charbonnières (cet ID est choisi lors de la création des masques, dans l'onglet "Préparation des données" du logiciel). On peut ensuite spécifier la taille des vignettes (qui doit être un multiple de 32), l'intervalle (en pixels) entre chaque vignette, la proportion d'images entre le jeu d'entraînement et le jeu de validation, et le "mode". Ce dernier paramètre a peu d'influence sur la qualité des prédictions, mais change par un facteur 4 la vitesse de prédictions ainsi que la résolution des segmentations obtenues (voir 3.1).

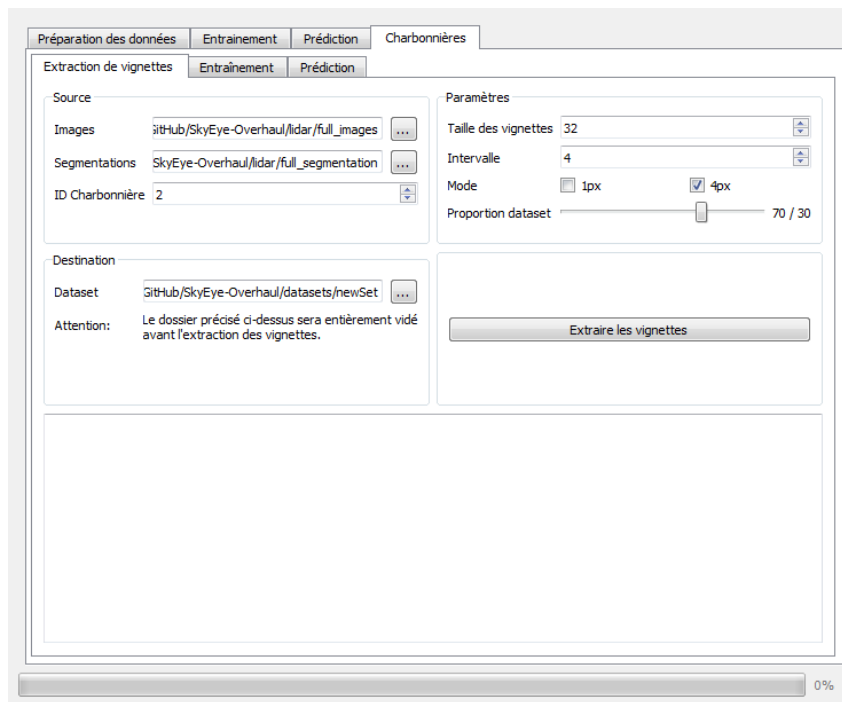


Figure 5.4 – Interface d'extraction de vignettes

4.2 Entraînement et évaluation

Cette interface est très similaire à celle du logiciel existant : des champs permettent de préciser le dossier contenant l'ensemble d'apprentissage et d'évaluation, ainsi que le chemin où enregistrer le modèle. On peut ensuite préciser plusieurs hyperparamètres : taille du batch, nombre d'étapes par époque, nombre d'époques... Enfin, deux boutons permettent d'entraîner un nouveau modèle, ou d'évaluer un modèle existant.

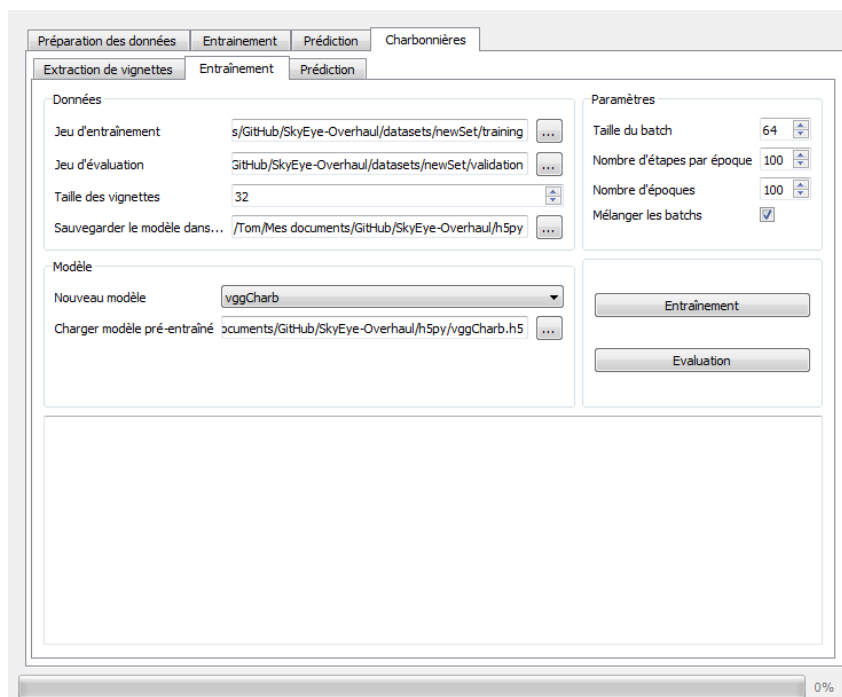


Figure 5.5 – Interface d'entraînement et d'évaluation

4.3 Prédictions

Comme les précédentes, l'interface de prédictions ressemble à celle déjà implémentée dans SkyEye : 4 champs sont présents pour spécifier le modèle à utiliser, les images à prédire ainsi que les dossiers où sauvegarder les superpositions et les segmentations. En plus de cela, 4 autres champs permettent de préciser la taille du batch, la taille des vignettes, l'intervalle et le mode. Afin d'obtenir de bons résultats, ces paramètres doivent être les mêmes que ceux précisés lors de l'extraction des vignettes.

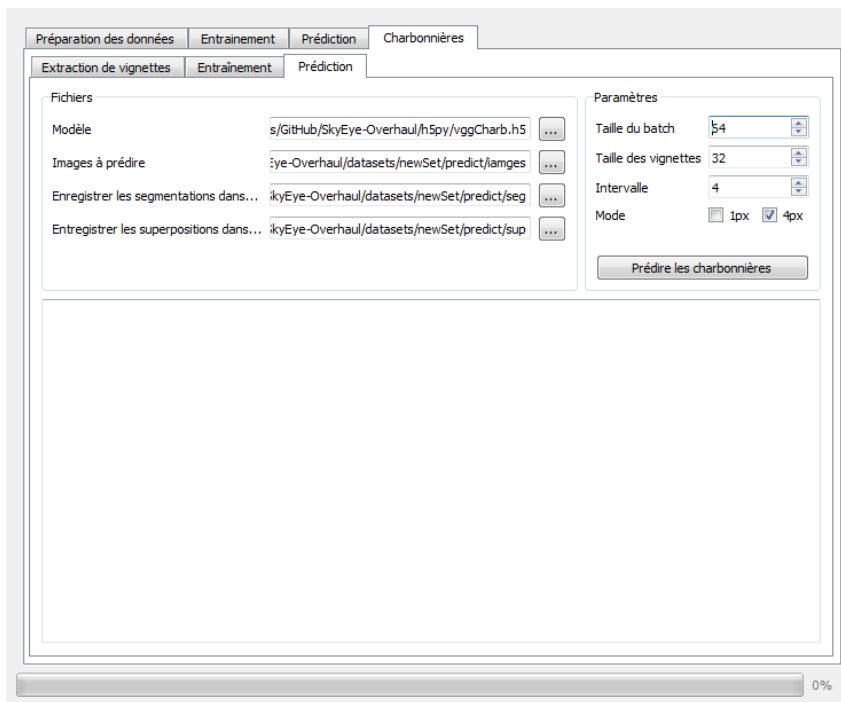


Figure 5.6 – Interface de prédictions

5 Analyse des résultats, évaluation, qualité

Plusieurs tests ont déjà été réalisés par Valentin Maurice lors de son PRD : la grande majorité des modules de *keras-segmentation* ont été testés par son créateur ou par Valentin, tout comme les fonctionnalités initiales de l'outil SkyEye. J'ai donc utilisé l'outil *PyTest* pour implémenter de nouveaux tests unitaires sur les fonctions développées au cours de ce PRD. Ces tests ne permettent malheureusement pas de vérifier parfaitement le déroulement des fonctions d'entraînement, d'évaluation ou de prédictions, mais permettent néanmoins d'assurer la cohérence des entrées/sorties et des fichiers manipulés par ces fonctions. Ainsi, je les ai utilisés en tant que tests de non-régression lors de l'ajout ou la modifications de certaines fonctionnalités. La liste complète des tests unitaires est disponible en annexe, dans la partie "Cahier de tests" **F**.

Pour évaluer la qualité du code source produit, j'ai utilisé l'outil *PyLint* qui permet d'effectuer une analyse statique du code, afin de vérifier le respect des bons usages et normes de la convention Python PEP 8.

Enfin, les performances et la qualité du modèle ont été évalués à l'aide de différentes mesures : matrice de confusion, précision, rappel et F1-Score sur les 2 classes à prédire, etc. Malheureusement, au terme du projet, les résultats retournés par les prédictions étaient toujours insuffisants par rapport à ceux attendus. Le réseaux de neurones semble parvenir à reconnaître certaines caractéristiques des charbonnières, car il arrive fréquemment à en détecter quelques une sur les images LiDAR. De plus, les rares faux-positifs (fond détecté comme charbonnière) pourraient

être retirés à l'aide de post-traitement (ex : suppression du bruit sur une image). Cependant, il reste un grand nombre de charbonnières qui ne sont tout simplement pas détectées par le réseaux, ce qui rend l'outil peu utile à des fins d'aide à la décision. Je pense néanmoins que la piste explorée à travers ce PRD est prometteuse, et qu'il serait judicieux de continuer à l'exploiter afin d'atteindre des résultats plus satisfaisants.

En terme de temps d'exécution, la phase d'entraînement et de prédictions sont relativement rapides : l'entraînement (avec 100 époques et 40 000 vignettes) dépasse rarement les 30 minutes, et les prédictions ne durent pas plus d'une minute par image (tests réalisés avec Keras compilé pour utiliser l'accélération GPU, sur une GTX 2060 6Go).

6

Bilan et conclusion

1 Bilan du semestre 9

Je suis globalement satisfait du déroulement de cette première phase du projet Recherche & Développement. Ce projet me permet de mettre en oeuvre mes connaissances théoriques en apprentissage artificiel, et ainsi développer des compétences pratiques, notamment via l'utilisation du framework Keras.

La première phase du projet a pris légèrement plus de temps que prévu à démarrer à cause de la prise en main du logiciel SkyEye, qui nécessitait une installation assez complexe (probablement à cause du passage du développement de Windows à Linux). Une fois le logiciel maîtrisé, j'ai pu étudier les travaux de Valentin Maurice et Florian Herguez, et commencer un état de l'art sur les problèmes similaires au mien. Enfin, j'ai utilisé la fin du semestre 9 pour développer un prototype de réseau de neurones, et rédiger la première version du rapport de PRD.

Le semestre 10 va être consacré à l'amélioration du modèle existant et à son implémentation dans le logiciel SkyEye. Pour cela, je vais commencer par améliorer le prototype jusqu'à avoir une architecture stable : celle-ci devra renvoyer de bons résultats, ne pas prendre trop longtemps à s'entraîner et ne doit pas planter en cours d'apprentissage. Une fois le modèle réalisé, je l'implémenterai au projet existant en l'adaptant aux fonctions développées par Valentin lors de son PRD. Je réaliserai ensuite une phase de tests pour valider ma solution, et mettrai à jour la documentation du projet ¹. Si ces objectifs sont accomplis en avance, deux pistes pourront être étudiées :

- La création d'un réseau de neurones à convolution spécialisé dans la segmentation de talus. Cette tâche, qui me semble complexe mais plus importante, pourra être réalisée en s'inspirant du modèle développé pour les charbonnières.
- L'amélioration du fonctionnement global du programme. Plusieurs points à faire avaient été relevés par Valentin à la fin de son PRD : refonte de la fonctionnalité d'augmentation des données, amélioration de la gestion d'erreurs, implémentation d'options de qualité de vie (en fonction des retours utilisateurs)...

1. Github : <https://github.com/Millasta/image-segmentation-keras/>

2 Bilan du semestre 10

Le début du semestre 10 a été marqué par une légère déception lorsque j'ai découvert que le modèle développé n'atteignait pas les résultats prédis au semestre 9. Cependant, cela m'a permis d'appréhender la difficulté de la recherche en intelligence artificielle et plus particulièrement en apprentissage profond. Les réseaux de neurones étant des technologies relativement récentes et en constante évolution, il est assez difficile d'en comprendre parfaitement le fonctionnement et d'en diagnostiquer les erreurs. J'ai néanmoins eu le sentiment que je ne parvenais pas à résoudre mes problèmes à cause d'un manque de recul et de connaissances théoriques sur le sujet.

Malgré le temps utilisé à travailler sur la conception du modèle, j'ai pu implémenter plusieurs nouvelles fonctions dans le logiciel SkyEye, ce qui, je l'espère, facilitera les prochaines expérimentations sur ce concept de modèle. Comme il est écrit précédemment, je pense que la piste explorée à travers ce PRD est prometteuse et qu'il serait judicieux de continuer à l'explorer afin de déterminer le problème qui bloque l'apprentissage et de parvenir à développer un modèle capable de prédire efficacement les charbonnières.

3 Bilan qualité de mise en oeuvre

La reprise d'un projet existant est souvent quelque chose de délicat, car si le développeur précédent ne s'est pas imposé une certaine rigueur, le code peut rapidement devenir une boîte noire difficile à déchiffrer. Cependant, je n'ai pas eu de mal à comprendre le code existant, et les mesures prises par Valentin au cours de son PRD pour rendre le projet plus accessible (documentation, tests unitaires, analyse statique du code...) m'ont énormément facilité la tâche lorsqu'il m'a fallu implémenter de nouvelles fonctionnalités au logiciel.

J'ai donc réutilisé les outils choisis par Valentin pour tenter de produire du code de la même qualité, en documentant toutes les nouvelles fonctions avec *pdoc*, en réalisant le plus possible de tests unitaires avec *pytest*, et en analysant fréquemment la qualité du code source avec *pylint*. Je pense que ces mesures permettront au projet d'être facilement repris, modifié et amélioré par la suite.

4 Bilan gestion de projet

Tout au long de ce projet, j'ai veillé à réserver régulièrement des moments destinés à faire le point sur mon avancée, sur mes difficultés et sur les prochains objectifs à accomplir. J'ai aussi pu bénéficier d'une communication très régulière avec mon encadrant, Thierry Brouard, ce qui m'a permis faire remonter rapidement mes problèmes et de décider ensemble de solutions à aborder. Enfin, l'utilisation de l'outil Trello m'a permis de visualiser rapidement mon état d'avancée du projet et de rester à jour sur ma progression.

Une de mes grandes difficultés lors de ce projet a été de devoir travailler de manière totalement autonome depuis chez moi. Ainsi, j'ai eu une légère baisse de productivité à l'annonce du second confinement, car il est pour moi assez difficile de rester concentré sur une tâche précise dans un environnement peu adapté au travail. À partir du début du second semestre, j'ai pu réserver des créneaux deux fois par semaine afin de travailler dans des salles de Polytech. Le fait de travailler dans un lieu approprié, dans un cadre beaucoup plus scolaire, m'a permis de rattraper du retard accumulé sur certains tâches et de me recentrer sur mon planning initial.

Annexes



Planification, gestion de projet

1 Évolution du projet

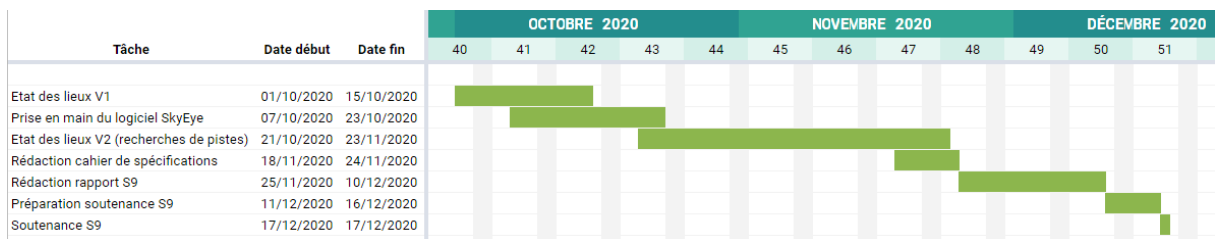


Figure A.1 – Diagramme de Gantt prévisionnel du S9

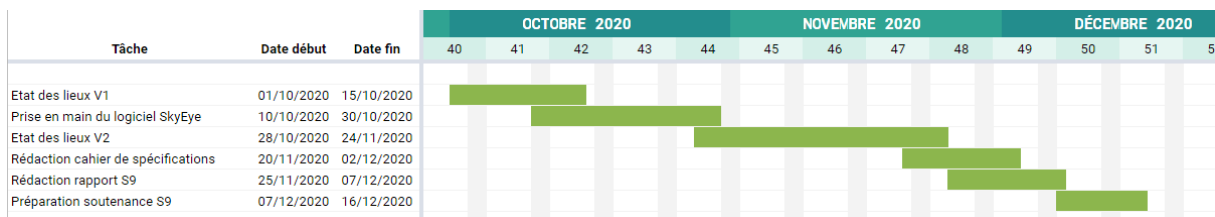


Figure A.2 – Diagramme de Gantt effectif du S9

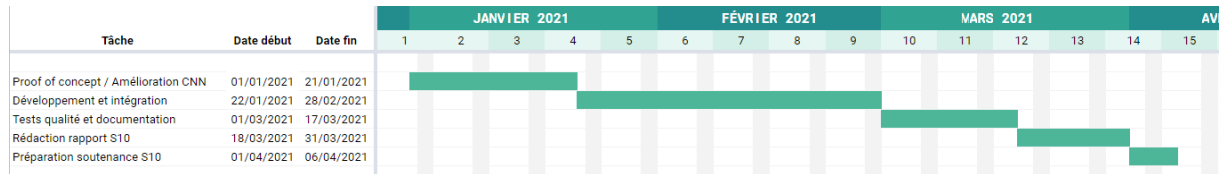


Figure A.3 – Diagramme de Gantt prévisionnel du S10

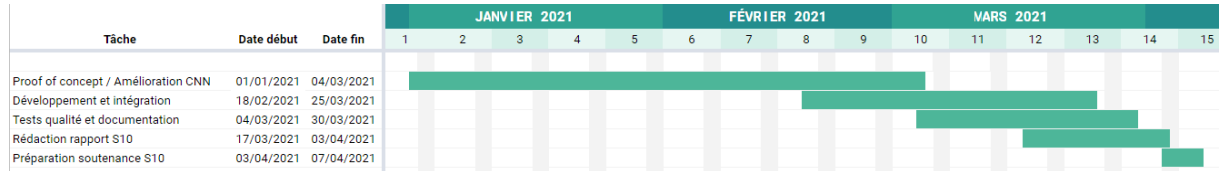


Figure A.4 – Diagramme de Gantt prévisionnel du S10

B

Cahier de Spécification

École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Département informatique

Cahier de spécification système			
Projet :	SkyEye : Archéologie & Deep Learning		
Émetteur :	Tom SUCHEL	MOA :	T. BROUARD, J-Y. RAMEL, C. LAPLAIGE, X. RODIER, N. LE VOGUER
Date d’émission :	11/2020		
Validation			
Nom	Date	Valide (O/N)	Commentaires
Thierry Brouard	09/12/2020	O	

Historique des modifications		
Version	Date	Description de la modification
00	11/2020	Version initiale du document
01	12/2020	Version corrigée selon retours MOA (T. Brouard)

Tables des matières

1 Introduction	3
2 Contexte de la réalisation	3
2.1 Contexte	3
2.2 Objectifs	4
3 Description générale	5
3.1 Environnement du projet	5
3.2 Caractéristiques des utilisateurs	5
3.3 Fonctionnalités du système	5
3.4 Structure générale du système	6
4 Description des interfaces externes du logiciel	6
4.1 Interfaces matériel/logiciel	6
4.2 Interfaces logiciel/logiciel	7
4.3 Interfaces homme/machine	7
5 Spécification fonctionnelles	8
5.1 Définition de la fonction 1 : thumbnailCreation	8
5.2 Définition de la fonction 2 : CNNcharb	9
5.3 Définition de la fonction 3 : predictCharb	10
5.4 Intégration dans le logiciel existant	10
6 Spécifications non fonctionnelles	11
6.1 Contraintes de développement et conception	11
6.1.1 Qualité du code	11
6.1.2 Documentation	11
6.1.3 Tests et évaluation	12
6.2 Contraintes de fonctionnement et d'exploitation	12
6.2.1 Performances	12
6.2.2 Capacités	12
6.2.3 Contrôlabilité	12
6.2.4 Sécurité	12
Glossaire	13

1 Introduction

Ce document est le cahier des spécifications du projet Recherche et Développement RFAI16 : “SkyEye : Archéologie & Deep Learning”. Ce projet s’inscrit dans la continuité du programme SOLiDAR, programme qui consiste à utiliser la technologie LiDAR afin de cartographier des zones forestières qui sont difficilement analysables depuis des photographies aériennes classiques, notamment à cause de la végétation. La maîtrise d’ouvrage est ici représentée par Thierry BROUARD et Jean-Yves RAMEL (enseignants chercheurs à l’Ecole Polytechnique de l’Université de Tours, département informatique), ainsi que par Clément LAPLAIGE, Xavier RODIER et Nathanaël LE VOGUER (chercheurs en archéologie UMR 7324 CITERES - LAT). La maîtrise d’œuvre est ici représentée par Tom SUCHEL (étudiant en 5ème année à l’École Polytechnique de l’Université de Tours, département informatique), auteur du présent document.

2 Contexte de la réalisation

2.1 Contexte

Le LiDAR (pour “Light Detection And Ranging”) est une technique de mesure de distance basée sur l’analyse des propriétés d’un faisceau de lumière envoyé vers un objet puis renvoyé vers son émetteur. À la différence du radar ou du sonar, le lidar utilise de la lumière issue d’un laser pour réaliser ses mesures. Cette technologie est utilisée dans une grande variété de domaines : topographie, météorologie, sciences de l’environnement, défense, mais aussi dans le domaine de l’archéologie, où elle présente de nombreux avantages. En effet, elle permet de passer outre la couverture végétale et permet donc une étude microtopographique du sol, en affichant des structures à la fois invisibles sur le terrain et sur des images satellites classiques.

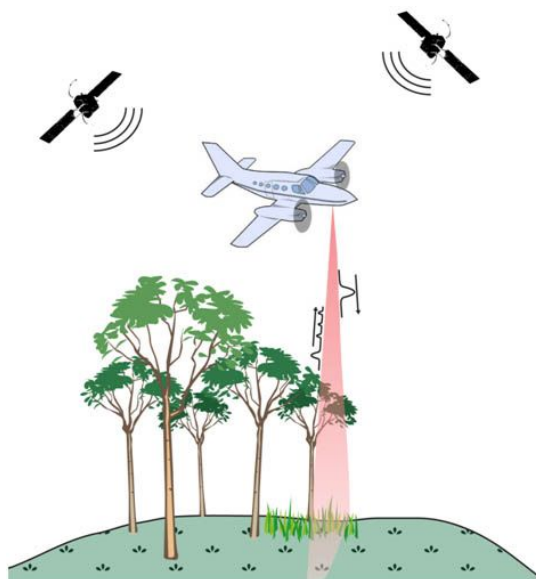
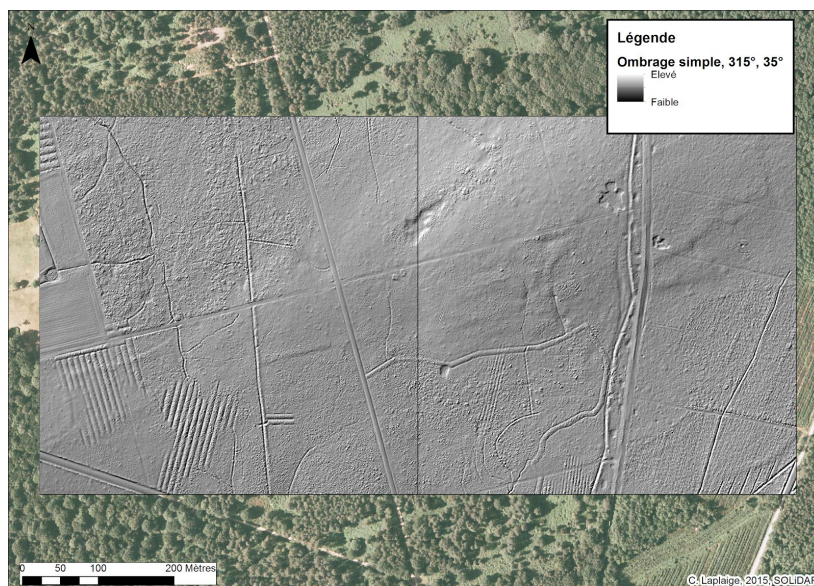


Schéma du fonctionnement du LiDAR (C. Laplaige, 2012)

(source: <http://citeres.univ-tours.fr/spip.php?article2133>)



Chambord, modèle ombré du terrain révélant des planches de labours, des secteurs d'extraction de matériaux et des loges de bûcherons
(source: <http://citeres.univ-tours.fr/spip.php?article2133>)

L'objectif de SOLiDAR est d'utiliser cette technologie pour améliorer la connaissance et la compréhension des dynamiques d'occupation du sol dans la durée, et ainsi ouvrir la porte à de nouvelles analyses archéologiques de la région de Chambord, Boulogne, Russy et Blois.

Le projet comporte un nombre très important de données à segmenter, travail actuellement réalisé à la main par les archéologues du CITERES. L'outil de segmentation automatique SkyEye a alors été développé pour aider à la décision lors de l'analyse, afin d'alléger le travail fourni par les experts.

2.2 Objectifs

L'objectif de ce projet Recherche & Développement est l'amélioration de l'outil SkyEye, et plus précisément des méthodes de segmentation automatiques intégrées à celui-ci. Cet outil a débuté avec une classification linéaire, en utilisant un SVM (Séparateur à Vaste Marge) mais les résultats étaient peu satisfaisants, ceux-ci dépassant rarement les 70% de classification correcte.

Le projet a été ensuite repris en 2019 lors d'un PRD réalisé par Valentin MAURICE, ancien étudiant de l'École Polytechnique de l'Université de Tours. L'objectif était alors d'améliorer les performances de la segmentation à l'aide d'une approche orientée réseaux de neurones profonds. Valentin a fait énormément avancer le projet, en proposant une refactorisation presque complète du code et en implémentant les outils nécessaires à l'utilisation de Deep Learning (augmentation des données, entraînement d'un nouveau modèle, sauvegarde et chargement d'un modèle existant, évaluation et prédictions) ainsi qu'une dizaine de modèles. Cependant, au terme du projet, les résultats de la prédiction étaient encore trop faibles pour que le logiciel puisse servir de base solide d'aide à la décision. Néanmoins, ces résultats restent très encourageants quant à l'utilisation de réseaux de neurones profonds pour segmenter des images LiDAR.

L'objectif de ce projet sera donc d'améliorer les modèles de prédiction actuels ou d'en développer de nouveaux, afin d'augmenter les performances du logiciel. On commencera d'abord par réaliser un premier modèle utilisable seulement pour différencier les charbonnières du fond, puis si le temps le permet, on généralisera ce modèle sur des structures archéologiques plus complexes : talus et tertres.

3 Description générale

3.1 Environnement du projet

Comme indiqué dans la partie précédente, l'objectif est d'implémenter de nouveaux modèles de segmentation, sans modifier le fonctionnement général du logiciel. Ces modèles seront d'abord développés à part, puis intégrés au programme existant.

On utilisera donc le même environnement que le projet original : le développement Python se fera avec l'IDE *PyCharm*, en utilisant un certain nombres de bibliothèques nécessaires au fonctionnement de l'outil : *appdirs*, *cycler*, *matplotlib*, *numpy*, *olefile*, *opencv-python*, *packaging*, *pillow*, *pyparsing*, *pyqt5*, *python-datautil*, *pytz*, *scikit-learn*, *scipy*, *sip*, *six*.

La partie interface du projet sera gérée par Qt, tandis que le développement des CNN sera réalisé en utilisant la librairie *Keras*, ainsi que *Sklearn* pour les tests et la validation.

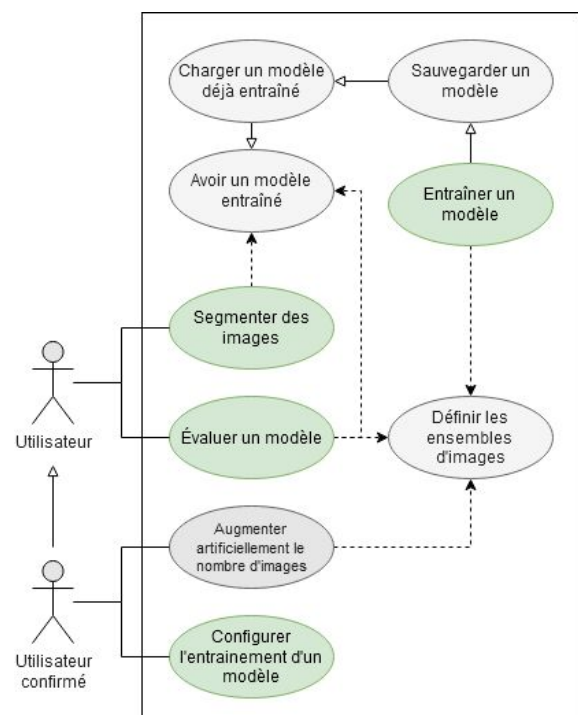
3.2 Caractéristiques des utilisateurs

Les utilisateurs de l'application sont des chercheurs en archéologie, on suppose qu'ils sont plutôt familiers avec l'usage de l'outil informatique, mais qu'ils possèdent seulement quelques notions, voire pas du tout, en apprentissage automatique/profond.

On veut donc une expérience utilisateur simple et intuitive, mais qui soit quand même complète pour laisser plus de choix aux utilisateurs possédant des connaissances plus poussées. On reprendra donc l'interface actuelle, qui respecte déjà tous ces points, en y intégrant les nouvelles fonctionnalités et les paramètres qui y sont liés.

3.3 Fonctionnalités du système

Le logiciel SkyEye sert à répondre aux cas d'utilisations affichés dans le diagramme ci-contre. Toutes ces fonctionnalités sont déjà implémentées dans la version actuelle du programme, le travail consistera essentiellement en la modification de ces fonctionnalités affichées en vert (fonctions d'entraînement, d'évaluation et de prédiction) pour qu'elles s'adaptent aux nouveaux réseaux de neurones.



3.4 Structure générale du système

Le projet actuel est déjà structuré de manière à ce qu'il soit facilement modifiable. Ainsi, les seules modifications qui vont être faites sur la structure du projet vont être l'ajout de nouveaux CNN à la liste des modèles déjà présents dans le module *keras_segmentation* et l'ajout de nouvelles fonctions spécifiques aux modèles créés dans le module *skyeeye*. Des modifications d'interfaces pourront aussi être réalisées si on veut laisser à l'utilisateur plus de personnalisation sur l'entraînement des CNN, ou la personnalisation des fonctions de vignettage.

Les CNN qui vont être implémentés lors de ce projet ne fonctionneront pas de la même façon que les modèles déjà implémentés dans le module *keras_segmentation* : actuellement, les modèles implémentés se chargent de segmenter des images, c'est à dire de prédire la classe de chacun des pixels de celle-ci. Les nouveaux modèles vont réaliser uniquement de la classification d'images, et c'est à partir des résultats fournis par ceux-ci que l'on reconstruit les masques de segmentation. Le procédé est donc réalisé en deux étapes : la classification, qui est gérée par les nouveaux modèles, et la segmentation, qui est gérée par les nouvelles fonctions ajoutées au contrôleur.

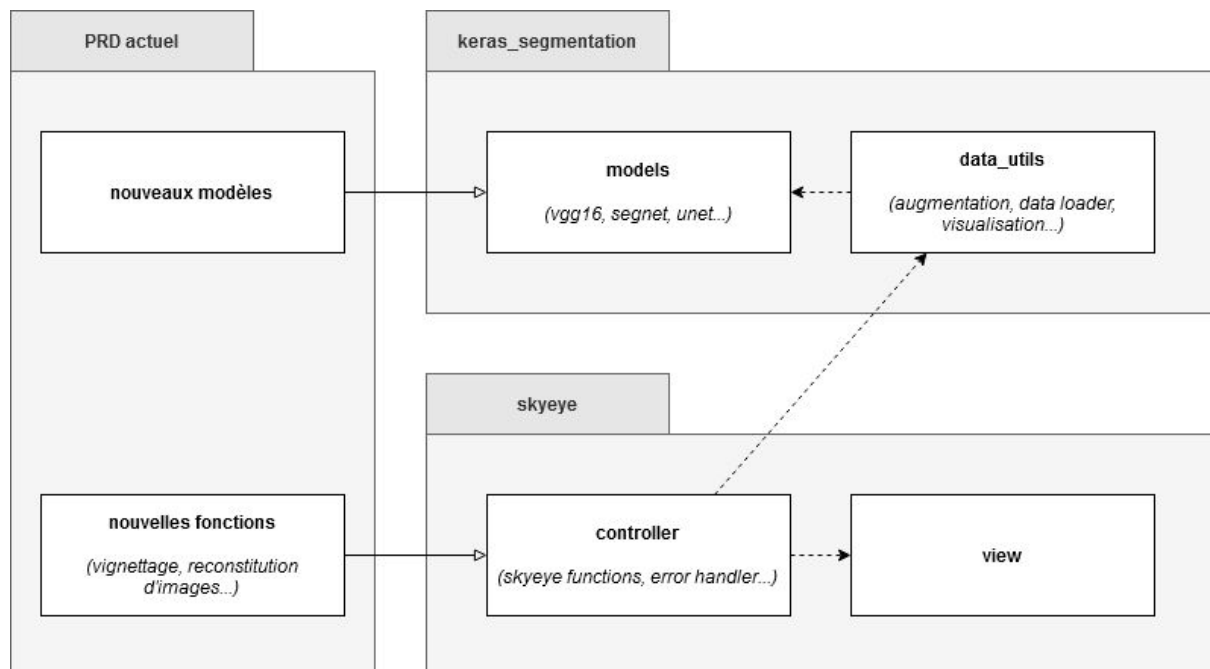


Diagramme simplifié de la structure du projet SkyEye

4 Description des interfaces externes du logiciel

4.1 Interfaces matériel/logiciel

Le logiciel est standalone, il ne nécessite pas d'interfaces matérielles supplémentaires. Néanmoins, il est nécessaire d'avoir un ordinateur assez puissant pour diminuer le temps de calcul lié à l'entraînement des modèles. De ce fait, l'utilisation d'une carte graphique Nvidia est recommandée pour profiter de l'accélération matérielle GPU, mais pas nécessaire.

4.2 Interfaces logiciel/logiciel

Le programme agit de manière indépendante et ne nécessite pas d'interface logicielle.

4.3 Interfaces homme/machine

L'interface homme/machine sera exactement la même que celle du logiciel existant, à laquelle on aura ajouté les nouveaux modèles ainsi que certaines entrées utilisateur pour saisir des paramètres. Les nouveaux modèles seront tout simplement ajoutés à la liste déroulante encadrée en rouge.

The screenshot shows a software interface with three tabs: 'Préparation des données', 'Entraînement', and 'Prédiction'. The 'Prédiction' tab is active. Under the 'Paramètres' section, there are several input fields and buttons. The 'Nouveau modèle' dropdown menu is highlighted with a red box and contains the text 'fcn_8'. Other parameters include 'Jeu d'entraînement', 'Jeu d'évaluation', 'Entraînement', and 'Logs'. The 'Entraînement' section has a 'Taille du batch' of 32, 'Nb d'étapes par époque' of 10, and 'Nb d'époques' of 10. The 'Logs' section is empty. At the bottom right, there is a progress bar showing 0%.

Si un modèle nécessitant l'utilisation de vignettage est sélectionné, une fenêtre similaire au mockup ci-contre s'affichera pour proposer des paramètres relatifs à la création des vignettes.

L'utilisateur peut y renseigner la taille des vignettes à créer via une liste déroulante proposant les différentes tailles possibles (celles-ci doivent être des multiples de 32 pour fonctionner dans un réseau de neurones de type VGG16)

L'incrément représente l'écart en pixels entre chaque vignette. On peut calculer le nombre d'images qui vont

The 'Vignettage' window is a modal dialog for configuring tiling parameters. It contains a 'Taille des vignettes' dropdown menu with the value 32, an 'Incrément' dropdown menu with the value 4, and a checkbox labeled 'Équilibrer les classes' which is checked. A 'Générer les vignettes' button is located at the bottom of the window.

être créées avec la formule ci-dessous (en assumant que les images source sont carrées):

$$\text{nombre de vignettes} = (\text{taille de l'image} / \text{incrément})^2$$

Ainsi, un petit incrément (minimum 1) générera un grand nombre d'images, ce qui améliorera probablement l'entraînement mais augmentera le temps de calcul.

Enfin, une case à cocher est présente pour que l'utilisateur puisse spécifier s'il veut équilibrer les classes ou pas. En effet, les images sont constituées en moyenne à 98% de fond (F. HERGUEZ, 2020, *Rapport de stage sur le projet SkyEye*), et il est peu intéressant de générer des milliers, voir des millions de vignettes pour une seule classe (sur une image de 400 * 400 pixels, on peut créer jusqu'à 160.000 vignettes de fond). Cette fonction limite alors le nombre de vignettes de fond créées en fonction du nombre de vignettes de charbonnières, sans impacter les performances de l'apprentissage (les premiers tests ont démontré qu'avoir environ 4000 vignettes pour chaque classe était suffisant pour atteindre jusqu'à 96% de précision).

5 Spécification fonctionnelles

La plupart des fonctions utiles à la gestion d'un réseau de neurones à convolution (entraînement, évaluation, sauvegarde et chargement de modèle...) sont déjà présentes dans le logiciel de base. Les fonctions développées lors de ce projet servent donc à adapter le nouveau modèle aux fonctions existantes.

5.1 Définition de la fonction 1 : *thumbnailCreation*

Identification de la fonction 1

Cette fonction permet de créer un ensemble de vignettes de petite taille à partir d'un ensemble d'images. La fonction est divisée en deux sous-fonctions : la première, utilisée avant l'entraînement d'un modèle, crée les vignettes et les organise dans une structure de dossiers en fonction de leur classe. La deuxième, utilisée avant une prédiction, crée simplement les vignettes et les classe par image source.

Description de la fonction 1

Les deux versions de cette fonction prennent en entrée un chemin vers un dossier contenant un ensemble d'images, et différents paramètres : taille des vignettes, incrément en pixel entre chaque vignette, booléen représentant l'équilibrage ou non des classes (seulement pour la version "entraînement"). En sortie, la première fonction crée l'arborescence de dossiers ci-dessous et y place les vignettes en fonction de leur classe. La deuxième fonction répartit les vignettes dans des dossiers du même nom que l'image utilisée pour les créer. Les vignettes sont nommées en fonction de leur position dans l'image.

Arborescence fonction “train”	Arborescence fonction “predict”
<pre> > train > background > x_y.png > ... > charbonniere > eval > background > charbonniere </pre>	<pre> > img1 > x_y.png > ... > img2 > img3 > ... </pre>

Alternative envisagée

Cette fonction, très efficace et simple à implémenter, possède néanmoins une faille : toutes les vignettes sont enregistrées sur l’espace mémoire du disque pour être finalement supprimées à la fin de l’exécution du programme. Cela n’est pas dérangeant tant que l’on ne crée pas un trop gros volume de vignettes : par exemple, pour la phase d’entraînement, on peut créer au maximum environ 10 millions de vignettes (car on possède seulement 66 images labellisées), soit environ 9Go de données. Dans les faits, se limiter à quelques milliers ou dizaines de milliers de vignettes sera suffisant pour entraîner efficacement un modèle, et n’occupe que quelques Mo d’espace disque. Cependant, pour la phase de prédiction, il faut générer toutes les vignettes d’une image pour la segmenter efficacement, et le nombre d’images à segmenter pourra être très élevé. Dans ce cas là, on ne peut pas se permettre d’enregistrer les vignettes sur le disque, et on utilisera donc une implémentation de type *DataLoader*, qui permet de générer des données à la volée pour les transmettre au réseau de neurones. La logique de la fonction sera la même, à l’exception que les vignettes seront générées et fournies une par une au réseau qui s’occupera de les classer.

5.2 Définition de la fonction 2 : *CNNcharb*

Identification de la fonction 2

Cette fonction représente le réseau de neurones à convolution développé pour classer les différentes vignettes.

Description de la fonction 2

Cette fonction prend en entrée les chemins vers les bases d’apprentissage et de validation, ainsi que le nombre d’époques, le nombre d’étapes par époque, la taille du batch et le chemin où sauvegarder le modèle. Cette fonction est ensuite appelée par la fonction *train*, déjà présente dans le logiciel de base, qui lance l’entraînement sur le réseau de neurones et enregistre le modèle au chemin spécifié. Le modèle utilisé est une version “simplifiée” de VGG16, comportant à ce jour 4 couches de convolution, 3 couches de max pooling et 2 couches complètement connectées.

5.3 Définition de la fonction 3 : *predictCharb*

Identification de la fonction 3

Cette fonction permet de construire des masques de segmentation à partir des vignettes créées avec la fonction 1. Pour chaque dossier contenant des vignettes (donc pour chaque image), elle prédit la classe des vignettes et crée un masque de segmentation de la même taille que l'image originale.

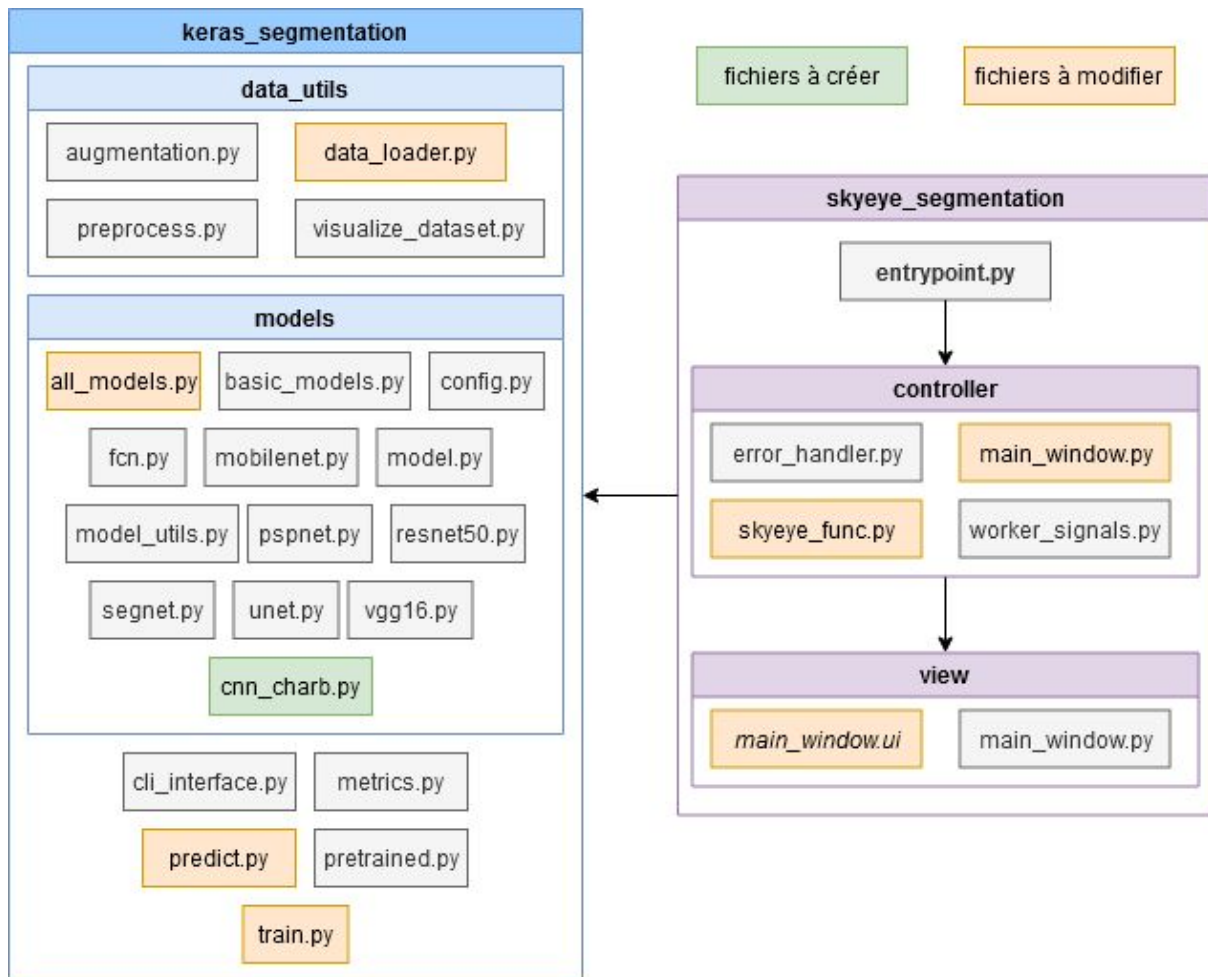
Description de la fonction 3

Cette fonction prend en entrée le chemin vers un ensemble de vignettes (créé par la version "predict" de la fonction 1), le chemin vers un modèle 'CNNcharb' entraîné sur des vignettes, et un chemin où enregistrer les masques de segmentation. La fonction prédit ensuite la classe de chaque vignette, qui correspond à la classe du pixel central, et crée des masques de segmentation qu'elle enregistre dans le chemin spécifié.

5.4 Intégration dans le logiciel existant

Ainsi, comme on peut l'observer dans le schéma ci-dessous, le seul fichier à créer sera l'implémentation du modèle de segmentation de charbonnières. Le reste du développement consistera en des modifications et des ajouts de fonctions sur des fichiers existants :

- **skyeeye_func.py**: modifications des classes TrainWorker, EvalWorker et PredictWorker pour ajouter le cas d'utilisation du nouveau modèle et les exceptions qui y sont liées.
- **main_window.ui**: modifications de l'interface directement depuis QtDesigner pour ajouter la fenêtre de paramétrage du vignettage des images. Le fichier view.main_window.py est ensuite généré automatiquement avec la commande `pyuic5 main_window.ui > main_-window.py`.
- **controller.main_window.py**: modifications à prévoir pour relier les nouveaux signaux générés par l'interface.
- **all_models.py**: ajout du nouveau CNN.
- **predict.py** et **train.py**: adaptation des fonctions d'entraînement, évaluation et prédiction dans le cas d'utilisation du nouveau CNN.
- **data_loader.py**: adaptation des fonctions de chargement et/ou de génération d'images dans le cas d'utilisation de vignettes.



6 Spécifications non fonctionnelles

6.1 Contraintes de développement et conception

Le développement sera réalisé en **Python** pour rester dans la continuité du projet existant. Pour les mêmes raisons, on utilisera les fonctionnalités de la librairie **Keras**, choisie lors du précédent PRD, pour construire les nouveaux réseaux de neurones.

6.1.1 Qualité du code

Pour veiller au bon respect des conventions de nommage et des bonnes pratiques de programmation, on utilisera l'outil **pylint** pour réaliser une analyse statique du code.

6.1.2 Documentation

La documentation du projet sera générée au format HTML en utilisant l'outil **pdoc**. On veillera donc à réaliser des descriptions et commentaires pour chaque nouvelle fonction en respectant le même formalisme que le projet existant.

6.1.3 Tests et évaluation

Le projet comportera une très grande phase de tests qui pourra être divisée en deux parties :

- On devra tester avec attention les fonctions développées : vérification des entrées/sorties, gestion des exceptions, cohérence des résultats, etc. Ces tests seront réalisés avec l'outil **pytest**, déjà utilisé sur le projet actuel.
- On devra aussi réaliser une phase d'évaluation des nouveaux CNN : en calculant différentes métriques (précision, rappel, f1-measure, matrices de confusions...), on évaluera la performance de nos modèles, afin de vérifier qu'ils respectent un certain seuil de qualité. Au terme de ce projet, on souhaite dépasser les 90% de f1-measure (moyenne harmonique entre la précision et le rappel) sur la détection de fond et de charbonnière.

6.2 Contraintes de fonctionnement et d'exploitation

6.2.1 Performances

Entraîner un réseau de neurones et l'utiliser pour réaliser des prédictions est une tâche très coûteuse en ressources, et qui peut donc durer très longtemps.

Pour notre projet, on souhaite que le temps d'entraînement et de prédiction ne dépasse pas plus de quelques heures : pour la phase d'entraînement, on souhaite se donner une limite maximum de 4h. Pour respecter cette contrainte, on veillera à ne pas surdimensionner le réseau afin de ne pas augmenter sa complexité. Pour la phase de prédiction, on souhaite limiter le temps d'exécution à 5 minutes par image à segmenter.

On souhaite aussi que le programme ne bloque pas au milieu d'un traitement. Pour cela, on veillera à faire attention à la validation des données d'entrée, à la gestion des exceptions et on sauvegardera les modèles à intervalle régulier (chaque époque) pour ne pas perdre la progression en cas de crash (il est impossible de reprendre un entraînement en cours de route, mais on peut utiliser les fichiers sauvegardés pour réaliser des prédictions).

6.2.2 Capacités

Le nombre d'images généré par la fonction de vignettage peut être très élevé (plusieurs centaines de milliers). Ces images, bien que très petites donc légères, peuvent représenter un volume important de données. Pour pallier ce problème, on utilisera les fonctionnalités des *DataLoader*, qui permettent de générer les données au fur et à mesure du déroulement des algorithmes (entraînement, évaluation, prédictions), ce qui évite d'encombrer temporairement l'espace disque.

6.2.3 Contrôlabilité

Le suivi de l'exécution des phases d'entraînement et de prédiction est déjà géré par le logiciel existant via l'utilisation d'une barre de progression et d'un fichier de logs. On veillera à adapter les nouvelles fonctionnalités à ces contraintes.

6.2.4 Sécurité

Aucune demande particulière n'a été faite de la part de la MOA en ce qui concerne la sécurité.

Glossaire

CITERES : UMR Cités, TERRitoires, Environnement et Sécurité

LiDAR : Light Detection And Ranging

SVM : Support Vector Machine, Machine à Vecteurs de Support/Séparateur à Vaste Marge

CNN : Convolutional Neural Network, Réseau de Neurones à Convolution

VGG16 : CNN réputé pour ses résultats en classification d'images (K. Simonyan, A. Zisserman, 2014, *Very Deep Convolutional Networks for Large-Scale Image Recognition*)

C

Fiche de configuration

Fiche de configuration

PRD RFAI16: Archéologie & Deep Learning

Configuration requise pour la dernière version de SkyEye:

- Python $\geq 3.6.5$
- Qt 5.15
- Librairies Python:
 - tensorflow==1.9.0
 - protobuf==3.6.0
 - keras==2.2.5
 - pyqt5
 - opencv-python
 - imgaug
 - sklearn
 - six
 - pillow
 - matplotlib
 - scikit-image
 - numpy
 - h5py
 - tqdm
- Librairies additionnelles:
 - pytest
 - pdoc3
 - pylint

Étudiant: Tom SUCHEL

Encadrants: Thierry BROUARD, Jean-Yves RAMEL

D

Manuel développeur

Outil de segmentation d'image en Deep Learning : SkyEye

Ce projet est un *fork* du projet <https://github.com/Millasta/image-segmentation-keras>, lui-même étant un fork du projet <https://github.com/divamgupta/image-segmentation-keras> : *Implémentation de Segnet, FCN, UNet, PSPNet et d'autres modèles avec Keras.*

Le projet a été réalisé par Tom SUCHEL dans le cadre du **Projet Recherche et Développement** (Projet de Fin d'Étude) de la troisième année du cycle ingénieur au département informatique de l'école Polytech Tours, lors de l'année 2020-21.

L'objectif de ce projet est de rajouter un ensemble de fonctionnalités à l'outil SkyEye, permettant la segmentation automatique des charbonnières sur des images LiDAR. Le projet initial a été développé par Valentin MAURICE lors de son PRD sur le même sujet, réalisé en 2019-20.

Prérequis

Le projet a été développé avec une distribution Python 3.6.5 (64bit) sur windows, et nécessite les dépendances suivantes (à retrouver dans requirements.txt) :

- tensorflow==1.9.0
- keras==2.2.5
- protobuf==3.6.0
- six (1.14.0)
- pillow (7.0.0)
- matplotlib (3.1.3)
- scikit-image (0.16.2)
- numpy (1.18.1)
- h5py (2.10.0)
- tqdm (4.43.0)
- PyQt5 (5.14.1)
- opencv-python (4.2.0.32)
- imgaug (0.4.0)
- sklearn (0.0)

Attention: Qt 5.15 doit être installé sur l'ordinateur afin de pouvoir exécuter le logiciel.

Lancement

Il suffit d'exécuter `skyeye_segmentation/entrypoint.py` avec Python, l'application devrait s'ouvrir.

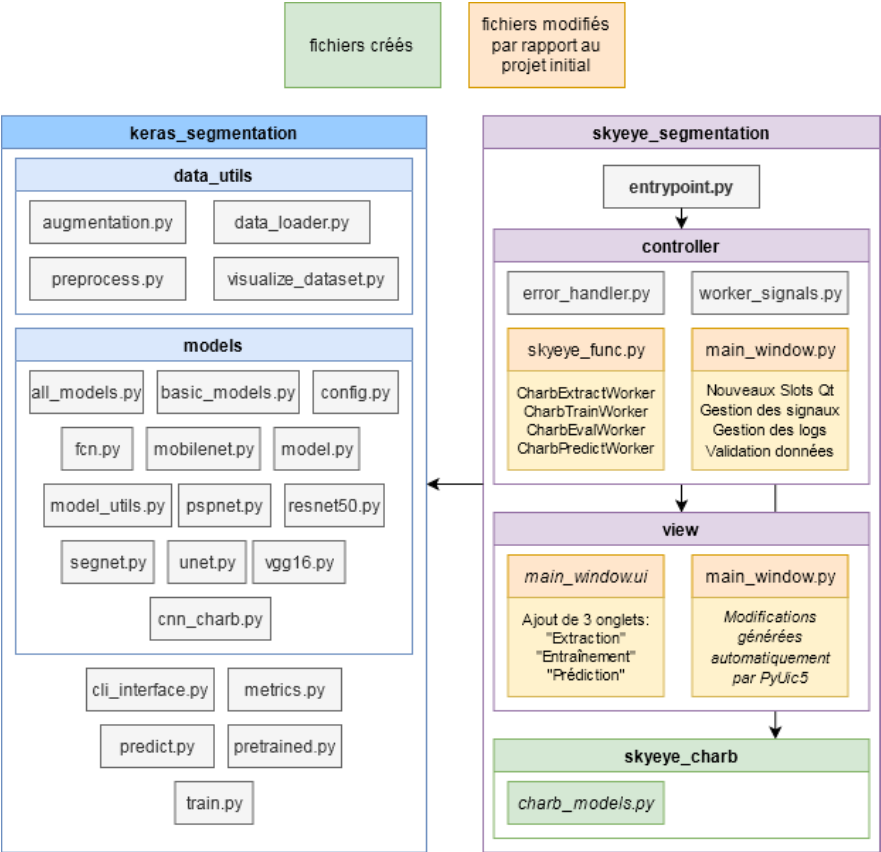
Structure du projet

Le projet est structuré de la manière suivante :

- **html/** : contient la documentation générée avec [pdoc](#)
- **keras_segmentation/** : sources du projet initial forké
- **log/** : fichiers logs
- **README/** : ressources de ce document
- **skyeye_segmentation/** : source du projet
- **test_keras_segmentation/** : tests unitaires du projet forké
- **test_syeye_segmentation/** : tests unitaires du projet

Modules

Les modules pythons s'organisent de la manière suivante. Les modules en orange sont ceux qui ont été modifiés depuis la version initiale du projet.



Le module **view** contient un fichier .ui généré avec [Qt Designer](#), qui a été automatiquement traduit en .py grâce au convertisseur pyuic5 (contenu dans PyQt5) :

```
>>> pyuic5 main_window.ui > main_window.py
```

Documentation

La documentation disponible dans *html/* a été générée avec [pdoc](#) :

```
>>> pdoc skyeeye_segmentation --html
```

Tests unitaires

Lancer les tests unitaires des nouvelles fonctions du module *skyeeye_segmentation* (se placer dans *test_charb_segmentation/*) :

```
>>> pytest
```

Qualimétrie

Pour lancer une analyse statique du code avec [PyLint](#) (fichier de configuration : *.pylintrc*) :

```
>>> pylint --rcfile=.pylintrc skyeeye_segmentation > pylint.txt
```

Le rapport Pylint est alors disponible et donne des indications sur la qualité du code analysé, ainsi qu'une note générale :

Your code has been rated at 9.41/10 (previous run: 9.39/10, +0.02)

Manuel d'utilisation

Voir le [manuel d'utilisation](#)

Téléchargement

Télécharger l'archive de la dernière version [ici](#).

E

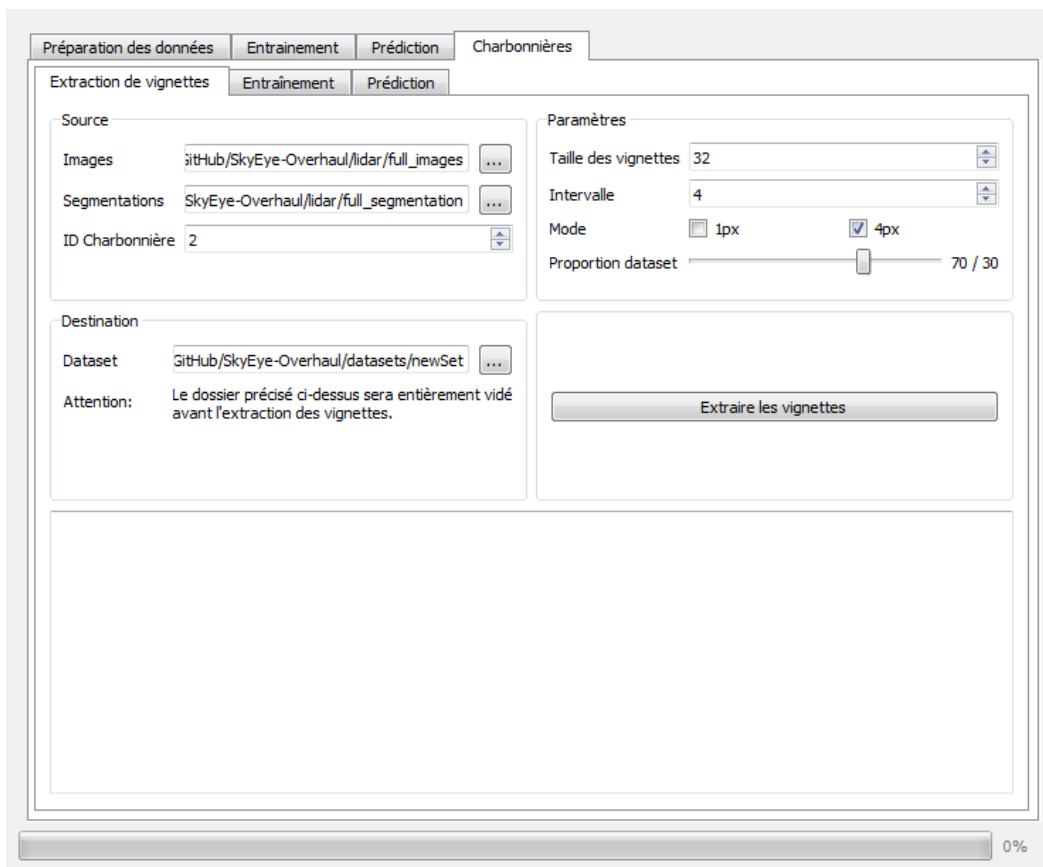
Manuel utilisateur

Manuel d'utilisation SkyEye (Outil "Charbonnières")

L'outil SkyEye permet d'entraîner un modèle de segmentation d'images, d'évaluer ses performances, effectuer des prédictions sur de nouvelles images, et préparer les données en amont de ces opérations. Le manuel d'utilisation de cet outil est disponible sur le dépôt du projet initial: <https://github.com/Millasta/image-segmentation-keras>. Ce manuel d'utilisation ne concerne que l'onglet "Charbonnières", implémenté en 2020-21.

Extraction des vignettes

La première fonctionnalité de l'onglet "Charbonnières" permet d'extraire des vignettes (des images carrées de petites dimensions) et de former des ensembles de données pour l'apprentissage et l'évaluation d'un modèle.



The screenshot shows the 'Charbonnières' tab in the SkyEye application. It features a sub-tab 'Extraction de vignettes'. The 'Source' section contains three fields: 'Images' (set to 'github/SkyEye-Overhaul/lidar/full_images'), 'Segmentations' (set to 'SkyEye-Overhaul/lidar/full_segmentation'), and 'ID Charbonnière' (set to '2'). The 'Destination' section has a 'Dataset' field (set to 'GitHub/SkyEye-Overhaul/datasets/newSet') and an 'Attention' note: 'Le dossier précisé ci-dessus sera entièrement vidé avant l'extraction des vignettes.' The 'Paramètres' section includes 'Taille des vignettes' (32), 'Intervalle' (4), 'Mode' (with '1px' and '4px' checkboxes, '4px' is selected), and a 'Proportion dataset' slider set to 70 / 30. A large 'Extraire les vignettes' button is located at the bottom right of the parameter section. A progress bar at the very bottom shows 0% completion.

Dans la partie "**Source**", il faut préciser:

1. Le dossier contenant les images LiDAR sur lesquelles vont être extraites les vignettes
2. Le dossier contenant les segmentations de ces images (Attention: les images et les segmentations doivent avoir le même nom!)
3. L'ID représentant les charbonnières (voir documentation du projet SkyEye). Sur les images contenues dans le dépôt GitHub, celui-ci est toujours 2.

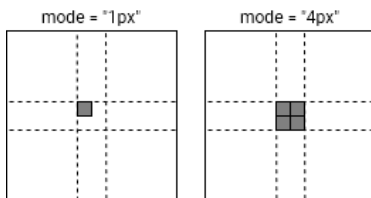
Dans la partie "**Destination**", il faut préciser le dossier dans lequel sera créé les ensembles d'apprentissage et d'évaluation. Après l'extraction, le dossier suivra la structure suivante:

- train
 - charb
 - *liste des vignettes*
 - [nom_image]_x_y.png
 - ...
 - back
- eval

- charb
- back

Enfin, la partie "**Paramètres**" permet de préciser:

1. La taille des vignettes. Celle-ci doit être un multiple de 32.
2. L'intervalle (en pixels) entre chaque vignette à extraire. Pour maximiser le nombre de vignettes, fixer l'intervalle à 1 pixel.
3. Le mode de choix de la classe. Si le mode est fixé à "1px", la classe de la vignette sera celle d'un unique pixel au centre de la vignette. Si le mode est fixé à "4px", la classe de la vignette sera "charb" si les 4 pixels centraux représentent une charbonnière, ou "back" dans le cas échéant.
4. La répartition des vignettes entre l'ensemble d'apprentissage et l'ensemble d'évaluation.



Il suffit ensuite d'appuyer sur le bouton "Extraire les vignettes" pour lancer l'extraction. Le dossier de destination sera vidé (cela peut prendre un peu de temps), et les vignettes seront créées et réparties selon l'arborescence détaillée plus haut.

Entraînement et évaluation

Le second onglet permet d'entraîner un nouveau modèle et/ou d'évaluer un modèle existant.

Pour entraîner un nouveau modèle, il faudra spécifier:

Dans la partie "**Données**":

1. Le dossier contenant le jeu d'entraînement.
2. Le dossier contenant le jeu d'évaluation. C'est sur les images contenues dans ce dossier que sera évalué le modèle entre chaque époque.
3. La taille des vignettes.

4. Le dossier dans lequel on veut enregistrer le modèle une fois entraîné. Le modèle sera nommé sous la convention suivante: [type_modèle].h5

Dans la partie "**Modèle**":

1. Le type de modèle à entraîner. Actuellement seuls deux architectures sont implémentées:
 - vgg4
 - vgg16

Dans la partie "**Paramètres**":

1. Taille du batch : combien d'images vont être montées en mémoire par itération, une plus grande valeur pourra produire un entraînement de meilleure qualité, mais sera plus couteux en ressources (RAM et CPU)
2. Nombre d'étapes par époque : nombre d'itérations par entraînement. Le nombre maximum d'étapes par époque est égal au nombre d'images dans le jeu d'entraînement divisé par la taille du batch. Pour utiliser toutes les images disponibles, le fixer à 0.
3. Nombre d'époques : nombre d'entraînements à effectuer.
4. Mélanger les batches: si on ne couvre pas la totalité du jeu d'entraînement dans une époque, il peut être intéressant de mélanger les batches afin que le réseaux de neurones s'entraîne sur plus d'images différentes.

Il suffira ensuite de cliquer sur le bouton "Entraînement" pour lancer l'apprentissage. Le modèle sera sauvegardé à chaque amélioration de sa précision, afin de n'en garder que la meilleure version.

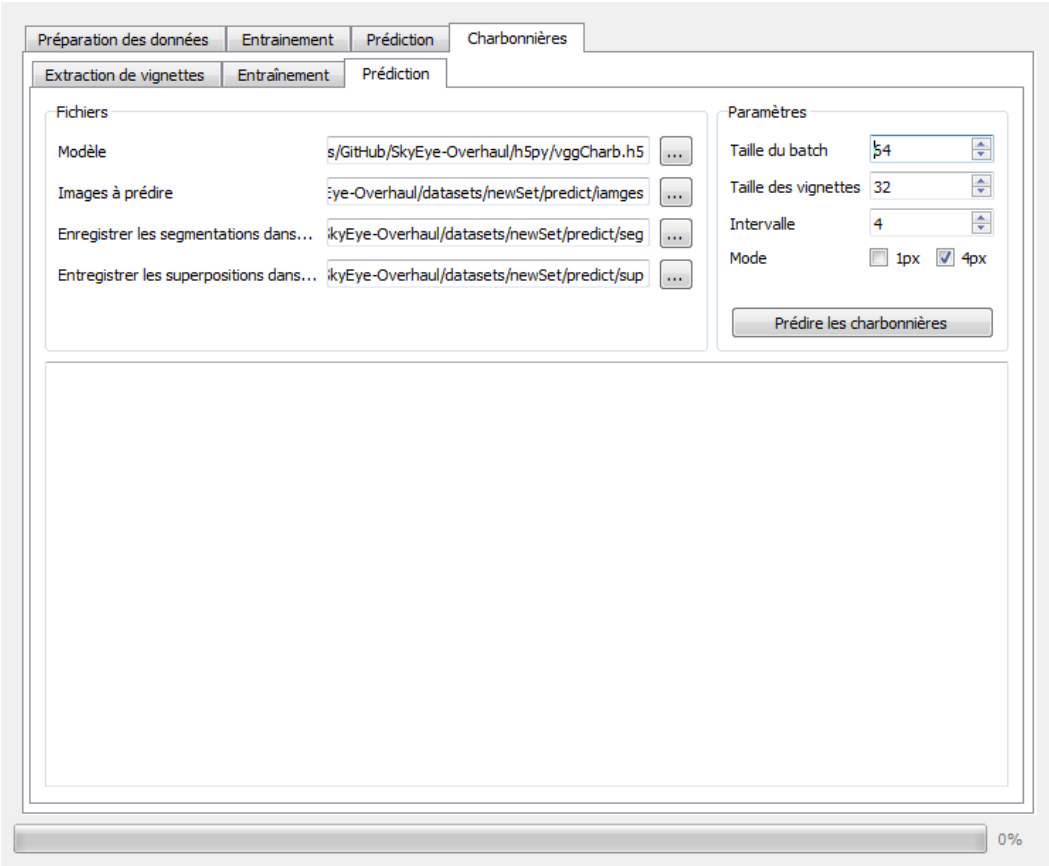
Une fois le modèle entraîné, il est nécessaire de le tester sur une base d'images séparée afin d'en évaluer ses performances. Pour cela, il faut spécifier:

1. Dans la partie "**Modèle**", le modèle qu'on vient d'entraîner. Celui-ci doit être au format h5.
2. Dans la partie "**Données**", le jeu d'évaluation sur lequel tester le modèle. Celui-ci peut être le même que le jeu d'évaluation utilisé pendant l'entraînement.

Il suffira ensuite de cliquer sur "Évaluation" pour tester le modèle. Plusieurs métriques (matrices de confusion, précision, rappel, f1-score) seront affichés dans les logs et permettent d'évaluer la qualité du modèle entraîné.

Prédictions

Le dernier onglet permet d'utiliser un modèle entraîné afin de prédire les segmentations d'une ou plusieurs images LiDAR.



Pour l'utiliser, il faudra spécifier:

Dans la partie "**Fichiers**":

1. Le modèle à utiliser. Celui-ci doit être au format *h5*.
2. Le dossier contenant les images LiDAR à prédire.
3. Le dossier où enregistrer les segmentations obtenues.
4. Le dossier où enregistrer les superpositions obtenues.

Dans la partie "**Paramètres**":

1. La taille du batch.
2. La taille des vignettes.
3. L'intervalle (en pixels) entre chaque vignette.
4. Le mode.

Attention: Afin de maximiser la précision des prédictions, la taille des vignettes et le mode doivent être les mêmes que ceux utilisés lors de l'extraction des vignettes. De plus, utiliser une intervalle plus grand que 1 (pour le mode "1px") ou 2 (pour le mode "4px") accélérera les prédictions mais diminuera la "résolution" des segmentations prédites.

F

Cahier de test

Le tableau affiché ci-dessous contient une description de tous les scénarios testés avec l'outil Pytest. Seules les fonctions nécessaires à la segmentation de charbonnières ont été testées.

1 Tests unitaires

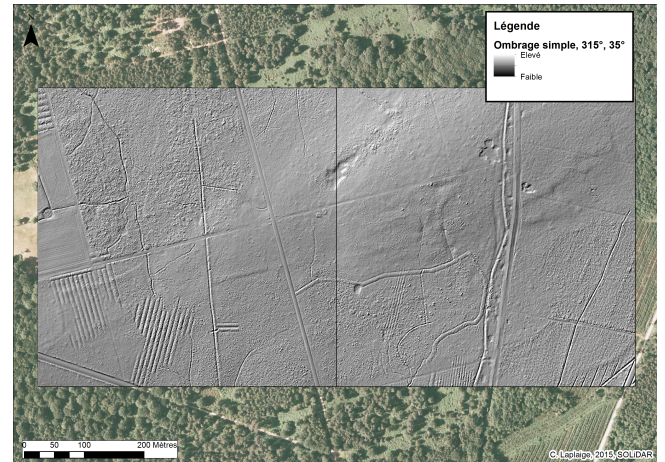
N°	Action / Objectif	Résultat attendu	Résultat	Commentaires
1	Extraction de vignettes			
1.1	Fonctionnement du module	Le module d'extraction s'exécute correctement sans planter	OK	/
1.2	Vitesse d'exécution	temps d'extraction pour 1 image (400x400px) < 5s	OK	moyenne = 2.9s/image
1.3	Création des sous-dossiers	dans le dossier "destination" sont créés les dossier suivants: - /training/charb, /training/back, /validation/charb, /validation/back	OK	/
1.4	Extraction du bon nombre de vignettes	le système récupère et sauvegarde le plus possible de vignettes de charbonnières	OK	le nombre de vignettes créé correspond au nombre calculé semi-automatiquement
1.5	Taille des vignettes valide	les vignettes sont de la taille spécifiée en paramètre	OK	testé avec vig_size = 32 et 64
1.6	Classes des vignettes correctes	la classe des vignettes est bien déterminée en fonction des pixels centraux (selon le mode)	OK	testé avec mode "1px" et mode "4px"
2	Entraînement du modèle			
2.1	Fonctionnement du module	Le module d'entraînement s'exécute correctement sans planter	OK	/
2.2	Sauvegarde du modèle	Les poids du modèle sont correctement sauvegardés au chemin précisé par l'utilisateur	OK	/
3	Évaluation du modèle			
3.1	Fonctionnement du module	Le module d'évaluation s'exécute correctement sans planter	OK	/
4	Prédictions d'images			
4.1	Fonctionnement du module	Le module de prédictions s'exécute correctement sans planter	OK	/
4.2	Création des segmentations et superpositions	Les segmentations et les superpositions sont créées et sauvegardées à l'emplacement renseigné par l'utilisateur	~	ne passe pas quand il est lancé depuis pytest, mais pass dans tous les autres cas
4.3	Taille des segmentation valide	Les segmentations sont de la même taille que les images initiales	OK	/
5	Fonction d'ajout de bords			testé avec plusieurs combinaisons de taille d'images et de vignettes (8/6, 400/32, 400/64)
5.1	Taille des images valide	Les images renvoyées sont de la bonne taille	OK	/
5.2	Bords corrects	Les "bords" sont rajoutés au bons endroits, avec la bonne valeur de remplissage	OK	/

- [1] Aayush BANSAL, Xinlei CHEN, Bryan RUSSELL, Abhinav GUPTA et Deva RAMANAN. *PixelNet : Towards a General Pixel-level Architecture*. 2016. arXiv : [1609.06694 \[cs.CV\]](#).
- [2] Erin BLAKEMORE. *Lasers are driving a revolution in archaeology*. Juil. 2019. URL : <https://www.nationalgeographic.com/culture/archaeology/lasers-lidar-driving-revolution-archaeology/>.
- [3] N. V. CHAWLA, K. W. BOWYER, L. O. HALL et W. P. KEGELMEYER. « SMOTE : Synthetic Minority Over-sampling Technique ». In : *Journal of Artificial Intelligence Research* 16 (juin 2002), p. 321-357. ISSN : 1076-9757. DOI : [10.1613/jair.953](#). URL : <http://dx.doi.org/10.1613/jair.953>.
- [4] Ross GIRSHICK, Jeff DONAHUE, Trevor DARRELL et Jitendra MALIK. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2014. arXiv : [1311.2524 \[cs.CV\]](#).
- [5] Suhyun KIM. *A Beginner's Guide to Convolutional Neural Network (CNNs)*. Fév. 2019. URL : <https://towardsdatascience.com/a-beginners-guide-to-convolutional-neural-networks-cnns-14649dbddce8>.
- [6] Qun LIU, Saikat BASU, Sangram GANGULY, Supratik MUKHOPADHYAY, Robert DiBIANO, Manohar KARKI et Ramakrishna NEMANI. « DeepSat V2 : feature augmented convolutional neural nets for satellite image classification ». In : *Remote Sensing Letters* 11.2 (nov. 2019), p. 156-165. ISSN : 2150-7058. DOI : [10.1080/2150704x.2019.1693071](#). URL : <http://dx.doi.org/10.1080/2150704x.2019.1693071>.
- [7] Valentin MAURICE. « Interactive Deep Learning : Application à la reconnaissance d'éléments archéologiques dans les images LiDAR ». Projet Recherche & Développement. Tours, France : Ecole Polytechnique de l'Université François Rabelais de Tours, 2019-2020.
- [8] *Présentation du projet SOLiDAR*. URL : <http://citeres.univ-tours.fr/spip.php?article2133>.
- [9] *Réseau neuronal convolutif*. URL : https://fr.wikipedia.org/wiki/R%C3%A9seau_neuronal_convolutif.

- [10] Mayar SHAFAY, Mohammed Abdel-Megeed Mohammed SALEM, Hala EBIED, Maryam AL-BERRY et Mohamed TOLBA. « Deep Learning for Satellite Image Classification ». In : jan. 2019, p. 383-391. ISBN : 978-3-319-99009-5. DOI : [10.1007/978-3-319-99010-1_35](https://doi.org/10.1007/978-3-319-99010-1_35).
- [11] H. SMALL et BROWN. « Handling Unbalanced Data in Deep Image Segmentation ». In : 2017.

Objectifs

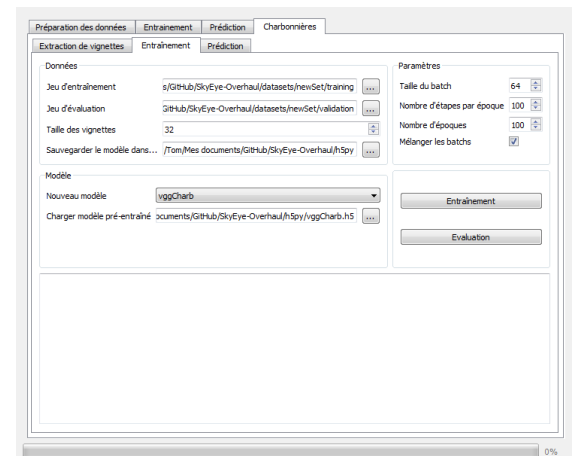
Le LiDAR est une technologie de télédétection qui permet de représenter les micro-reliefs d'un terrain tout en éliminant la couverture végétale. L'objectif de ce projet est de concevoir un système capable de segmenter des charbonnières sur ce type d'images, afin de servir d'outil d'aide à la décision pour les archéologues.



Superposition d'une prise de vue aérienne et image LiDAR d'une zone boisée.

Mise en œuvre

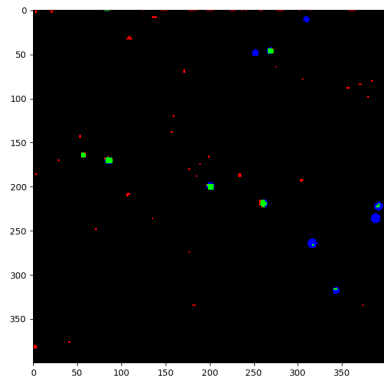
SkyEye est un logiciel développé par Valentin Maurice lors d'un PRD réalisé en 2019-20. Il embarque une dizaine de modèles de segmentation, qui se sont malheureusement révélés peu efficaces. Il a donc fallu imaginer un nouveau modèle, évaluer ses performances et l'intégrer au logiciel existant.



Interface d'entraînement de l'outil SkyEye

Résultats

Ce PRD aura permis de développer un nouveau modèle utilisant une méthode qui permet de palier au problème du faible nombre d'images disponibles pour l'apprentissage. De plus, de nouvelles fonctions d'entraînement, d'évaluation et de prédictions ont été intégrées dans le logiciel SkyEye.



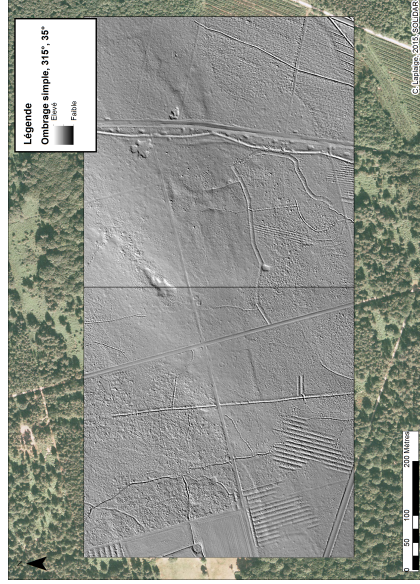
Segmentation prédite par le modèle

Objectifs

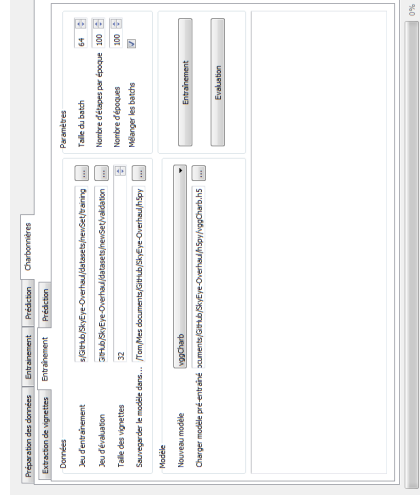
Le LiDAR est une technologie de télé- SkyEye est un logiciel développé par Va- Ce PRD aura permis de développer un détection qui permet de représenter les lentin Maurice lors d'un PRD réalisé en nouveau modèle utilisant qui méthode qui micros-reliefs d'un terrain tout en élimi- 2019-20. Il embarque une dizaine de mo- permet de palier au problème du faible nant la couverture végétale. L'objectif de dèles de segmentation, qui se sont malheu- nombre d'images disponibles pour l'ap- ce projet est de concevoir un système ca- reusement révélés peu efficaces. Il a donc prentissage. De plus, de nouvelles fonc- pable de segmenter des charbonnières sur fallu imaginer un nouveau modèle, évaluer tions d'entraînement, d'évaluation et de ce type d'images, afin de servir d'outil ses performances et l'intégrer au logiciel prédictions ont été intégrées dans le logi- d'aide à la décision pour les archéologues. existant.

Mise en œuvre

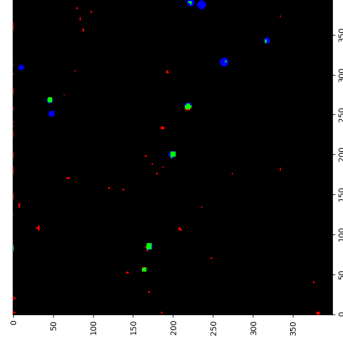
Résultats



Superposition d'une prise de vue aérienne et image LiDAR d'une zone boisée.



Interface d'entraînement de l'outil SkyEye



Segmentation prédite par le modèle

Projet SkyEye

Archéologie & Deep Learning

Résumé

Ce Projet de Recherche et Développement porte sur l'amélioration d'un système de segmentation automatique de structures archéologiques (charbonnières, tertres, talus...) fonctionnant à l'aide de Deep Learning. Ce système a été réalisé lors d'un ancien PRD, mais offre des résultats peu satisfaisants. Ce rapport couvre donc l'étude des différentes pistes d'amélioration de ce problème de segmentation, et la mise en place d'un CNN fournissant des résultats satisfaisants sur la détection de charbonnières.

Mots-clés

deep-learning, archéologie, LiDAR, segmentation, CNN

Abstract

This Research and Development Project concerns the improvement of an automatic segmentation system for archaeological structures (charcoal beds, mounds...) using Deep Learning. This system was developed in an old R&D project, but offers unsatisfactory results. This report therefore covers the study of the different ways of improving this segmentation problem, and the implementation of a CNN providing satisfactory results on the detection of charcoal beds.

Keywords

deep-learning, archeology, LiDAR, segmentation, CNN

Tuteurs académiques

Thierry BROUARD
Jean-Yves RAMEL

Étudiant

Tom SUCHEL (DI5)