

Rapport - Projets de Programmation

Détection automatique de pigments par Deep Learning sur des photos de blocs antiques

Victor Barrey (vbarrey@etu.u-bordeaux.fr)

Léo Dupouey (leo.dupouey@etu.u-bordeaux.fr)

Jonathan Moze (jonathan.moze@etu.u-bordeaux.fr)

Avril 2023

Lien Github : <https://github.com/JonathanMoze/PigmentDetector>



Table des matières

1	Introduction au domaine	4
1.1	Vocabulaire	4
1.2	Contexte général	5
1.3	Problématique	5
1.4	Objectifs	6
1.5	Périmètre	6
1.6	Modèle d'apprentissage profond : Analyse du problème et de l'existant	6
1.7	Vérité terrain	7
1.8	Risques	7
1.9	Tests	8
2	Analyse des besoins	9
2.1	Besoins liés à l'obtention de la vérité terrain	9
2.1.1	Besoins côté serveur	9
2.1.2	Besoins coté client	10
2.2	Besoins liés au formatage du jeu de données	12
2.3	Besoin liés à l'apprentissage automatique	13
2.4	Autres Besoins	14
3	Conception	15
3.1	Application générale	15
3.2	Étiqueteur	16
3.2.1	Entées et sorties de notre application	16
3.2.2	Choix de la technologie utilisée	16
3.2.3	Architecture générale d'un serveur Django	17
3.2.4	Maquettes de l'Étiqueteur	19
3.3	Générateur du jeu de données	20
3.4	Modèle d'apprentissage profond	21
3.4.1	Approche par segmentation sémantique (U-net)	21
3.4.2	Approche par segmentation d'instances (Mask R-CNN) .	22
3.4.3	Pré-traitements et post-traitements	22
3.4.4	Déséquilibre de classe et choix de la fonction objectif .	23
3.4.5	Évaluation des modèles	23
3.4.6	Choix de la technologie	24

4 Réalisation	25
4.1 Architecture générale de l'application	25
4.1.1 Afficher l'aide	25
4.1.2 Lancer le serveur d'étiquetage	25
4.1.3 Lancer le générateur de dataset	25
4.1.4 Lancer l'entraînement du réseau	26
4.1.5 Lancer une évaluation avec le réseau	26
4.2 Réalisation de l'étiqueteur	27
4.2.1 Implémentation de l'architecture Django	27
4.2.2 Fonctionnalités	29
4.2.3 Difficultés rencontrées	33
4.3 Réalisation du générateur de jeu de données	35
4.3.1 Description	35
4.3.2 Fonctionnalités	35
4.3.3 Difficultés rencontrées	36
4.4 Réalisation du modèle d'apprentissage profond	36
4.4.1 Unet	36
4.4.2 Mask R-CNN	37
4.4.3 Rapport d'entraînement	38
4.4.4 Prédiction sur de nouvelles images	38
5 Résultats	40
5.1 Entraînement	40
5.2 Évaluation du modèle	41
5.3 Exemples	44
6 Améliorations et extensions	46
6.1 Étiqueteur	46
6.2 Modèle d'apprentissage profond	46
7 Annexes	47
7.1 Schéma des modèles d'apprentissage	47
7.2 Captures d'écran	48
7.2.1 Rapport d'entraînement	48
7.3 Manuel utilisateur	50

1 Introduction au domaine

1.1 Vocabulaire

Clients

Les termes "clients", "archéologues" et "experts" font référence aux chercheurs du laboratoire Archéosciences de Bordeaux qui ont motivé la réalisation de ce projet. Ils sont représenté par M. Bruno Dutailly et nous ont guidé tout au long du projet afin que notre production corresponde bien à leurs besoins.

Patch

Le terme "patch" (parfois traduit par "morceau") est un terme anglophone communément employé dans le contexte de traitement d'image pour désigner une sous-partie d'une image, généralement obtenue après une opération de rognage.

Epoch

Le terme "epoch" (parfois traduit par "époque") est un terme anglophone communément employé dans le contexte de l'apprentissage automatique pour désigner un passage complet des données d'entraînement dans l'algorithme en charge de l'apprentissage.

Batch size

Le terme "batch size" (parfois traduit par "taille de lot") est un terme anglophone communément employé dans le contexte de l'apprentissage automatique pour désigner le nombre d'échantillons passés simultanément à l'algorithme d'apprentissage.

Photogrammétrie

La photogrammétrie est une technique de mesure et de cartographie qui utilise des photographies pour obtenir des informations précises sur la forme, la taille et la position d'objets ou de surfaces. En d'autres termes, c'est une méthode qui permet de créer des modèles 3D à partir de photographies.

1.2 Contexte général

Lors d'une campagne de macro-photogrammétrie au temple d'Apollon à Delphes, les chercheurs du laboratoire Archéosciences de Bordeaux ont remarqué la présence de micro-pigments bleus invisibles à l'oeil nu. Ces pigments sont des restes de peinture datant de l'époque de la Grèce antique. Cette peinture recouvrait à cette époque les blocs placés au niveau du plafond du monument, mais a depuis été altérée par le temps et les activités humaines.

1.3 Problématique

Ces pigments, invisibles a l'oeil nu, ont une forme plutôt circulaire avec un diamètre de quelques pixels dans les images de macro-photogrammétries. Cette petite taille dans les images de très grande résolution fait qu'ils sont difficile à détecter.



FIGURE 1 – Exemple de photo issue de la campagne de macro-photogrammétrie

1.4 Objectifs

Le but premier de ce projet est donc de permettre aux équipes de l'archéosciences de Bordeaux de retrouver dans des images de macro-photogrammétrie les traces de ces pigments afin de déterminer si un bloc était situé au niveau du plafond du monument et ainsi pouvoir mieux reconstituer le bâtiment en son temps.

Le second but du projet, à la demande de nos client, était d'étudier la possibilité d'utiliser les techniques d'apprentissage automatisé par réseaux de neurones afin de détecter ces pigments. En y parvenant nous pourrions fournir une preuve de concept permettant à l'avenir d'élargir la détection de petits éléments se détachant d'un fond relativement uniforme.

1.5 Périmètre

Notre projet s'inscrit dans le module **Projet de programmation** du second semestre du master Informatique de l'université de Bordeaux. Nous avons jusqu'au 23 Avril pour finir ce projet. Pour le mener à bien nous avons accès à 93 images issue de la campagne de macro-photogrammétrie avec un fichier au format csv nous indiquant les potentiels pigments présent dans ces images. Sur ces 93 images on retrouve parfois deux images d'un même bloc mais prises avec un éloignement différent. Après avoir effectué un tri, on décidera de ne garder que 70 images.

1.6 Modèle d'apprentissage profond : Analyse du problème et de l'existant

Nous souhaitons donc ici prédire la position de pigments de très petite taille dans des images de grande taille. Dans la littérature les problèmes de ce type sont souvent approchés comme des problèmes segmentation d'image, c'est à dire que l'on cherche à identifier des zones (groupe de pixels) et à les classifier. Un survey sur le sujet publié en 2020 [1] présente un bon nombre d'architectures de réseaux de neurones et d'approches pour réaliser de la segmentation d'image par apprentissage automatique.

Plus particulièrement le réseau U-net, initialement développé en 2015 pour traiter de l'imagerie médicale [2] semble pouvoir correspondre à notre besoin comme le montre cet article [3] qui propose un réseau capable de détecter les positions des bâtiments dans des prises de vue satellites.

Un autre réseaux qui a retenu notre attention est le Mask R-CNN [4] qui permet de réaliser de la segmentation d’instances sur toutes sortes images et qui obtient de très bon résultats sur le COCO Dataset [5], un jeu de données qui contient des images de plus de 40 types d’objets et entités dans des contextes variés. On peut penser qu’un tel réseau qui réalise des segmentations plus complexes pourrait nous permettre de correctement identifier nos zone de pigments.

1.7 Vérité terrain

Un des premiers problèmes à résoudre pour pouvoir envisager d’entraîner notre modèle d’apprentissage est celui de la vérité terrain. En effet, nous avons accès aux images prises sur les blocs antiques présentant des traces de pigments mais les positions de ces pigments n’ont jamais été formellement renseignés par des experts. Hors cela est nécessaire pour effectuer de l’apprentissage supervisé. De plus la détection des pigments sur les images est un processus compliqué et long même pour un expert, il nous faudra donc trouver un moyen d’obtenir cette vérité terrain avant de pouvoir mettre en place notre modèle d’apprentissage automatique.

1.8 Risques

Nous avons identifié 2 risques principaux pour la réalisation du projet :

Obtention de la vérité terrain

L’obtention de la vérité terrain est un objectif clef de la réalisation du projet car elle conditionne la possibilité d’entraîner des modèles d’apprentissage. Un potentiel retard dans l’obtention de cette vérité terrain ou de trop nombreuses erreurs d’étiquetage représentent un risque important pour notre projet. Il est donc primordial que cette vérité terrain soit créée le plus rapidement possible sans pour autant compromettre la qualité des données qu’elle contient. Pour cela nous avons fait le choix de concentrer les effort de l’équipe sur la réalisation de l’étiqueteur dès le début de la phase de développement afin de permettre aux experts d’avoir accès à l’outil le plus vite possible. Aussi nous avons jugé qu’un outil ergonomique permettrait aux experts non seulement d’être plus efficaces mais également plus précis dans la réalisation de la vérité terrain et c’est pour cela que nous avons pris soin de prendre en

compte cet aspect dès la phase de conception en réalisant notamment des maquettes pour guider le développement 3.2.4.

Modèle d'apprentissage

Comme évoqué plus tôt, le fait que le modèle d'apprentissage n'arrive pas à détecter les pigments à cause de leur faible présence dans les images est un risque important qui pourrait compromettre la capacité de notre programme à prédire les positions des pigments. Pour prévenir cela nous avons sélectionné deux réseaux de neurones qui pourrait répondre à notre problème, ainsi si le problème se présente avec l'un des réseau, nous pourrions toujours espérer faire fonctionner l'autre. De plus nous avons prévu un certain nombre d'actions à mettre en place pour éviter ce problème comme décrit la section 3.4.4.

1.9 Tests

La manière dont nous avons conçu notre application nous suggère de nous concentrer essentiellement sur deux types de tests.

Tests de non-régression

Notre application est conçue en modules pour lesquels les différentes fonctionnalités viennent s'ajouter sur la base de code de l'application. Si cette organisation facilite le développement, il nous faudra cependant être vigilant à ce que les nouvelles fonctionnalités n'altèrent pas le comportement du code déjà présent. Pour cela il faudra conduire des tests de non-régression après l'ajout de chaque nouvelle fonctionnalité, pour vérifier que les anciennes fonctionnalités fonctionnent toujours.

Test d'intégration

Le découpage du projet en modules va nous permettre de travailler indépendamment sur plusieurs parties du code en simultané. Si cela représente un gain de temps précieux pour la réalisation du projet, il sera important de s'assurer que les différents modules fonctionnent correctement les uns avec les autres une fois ajoutés à l'application principale. Il nous faudra pour cela

conduire des tests d'intégration consistant à valider le comportement de chacun des modules de l'application principale dès qu'un nouveau module est ajouté.

2 Analyse des besoins

2.1 Besoins liés à l'obtention de la vérité terrain

Nous avons précédemment soulevé le problème de la vérité terrain manquante. Nous avons donc réfléchi à mettre en place une application afin de créer cette vérité terrain, c'est à dire définir précisément les coordonnées, dans chaque image, des pigments bleus au pixel près. Cette application sera appelé l'*Étiqueteur*.

Pour permettre aux experts de l'Archéosciences de Bordeaux de renseigner au mieux les coordonnées des pigments, l'*Étiqueteur* devra satisfaire trois points :

- **Ergonomique** : Notre solution devait avoir une prise en main facile et permettre, autant que faire se peut, de rendre l'étiquetage des pixels le moins pénible possible.
- **Facilement déployable** : La solution devait être facile à déployer sur les machines de l'Archéopole de Bordeaux en limitant les actions requises pour installer les dépendances.
- **Accessible en réseau** : La solution devait permettre l'étiquetage sur plusieurs ordinateurs avec une base de données centralisée, uniquement accessible via un réseau local.

Nous avons donc déterminé que la meilleure solution pour satisfaire ces trois points était de créer une application web. L'utilisation du HTML, CSS et JavaScript assurerait la possibilité de faire une interface ergonomique et l'application serait accessible via d'autre machine depuis l'IP locale de celle ayant lancé le serveur. La deployabilité dépendrait du type d'application web que nous utiliseront.

2.1.1 Besoins côté serveur

Besoin 1 : Gérer les images présentes sur le serveur

Le serveur gère un ensemble d'images. Il stocke les méta-données nécessaires aux réponses des requêtes (identifiant, nom de fichier, chemin de stockage de

l'image dans les dossiers statiques, fichier json associé, état d'étiquetage).

Sous-besoin 1.1 : Ajouter une ou plusieurs images à la base de données

Lorsque le serveur reçoit une requête POST depuis l'adresse "upload/", il doit créer un nouvel élément image dans la base de données, remplir les métadonnées puis placer l'image reçue dans le dossier de fichiers statiques dédié au stockage des images. Les images sont marquées comme étant non étiquetées.

Sous-besoin 1.2 : Envoyer la liste des images à étiqueter et en cours d'étiquetage présentes sur le serveur

Lorsque le serveur reçoit une requête GET depuis l'adresse "accueil/" il doit retourner la liste de toutes les images de la base de données qui ne sont pas encore étiquetées et en cours d'étiquetage.

Sous-besoin 1.3 : Envoyer une image présente sur le serveur

Lorsque le serveur reçoit une requête GET de la page "labeler/<id>", il doit retourner l'image ayant l'ID correspondant ainsi que le fichier json associé si il existe. A ce moment là l'image dans la base de données doit être marquée comme étant en cours d'étiquetage. Si l'ID ne correspond à aucune image, le serveur doit retourner une réponse http 404.

Besoin 2 : Ajouter un fichier d'étiquetage

Lorsque le serveur reçoit une requête POST de la page "/labeler/<id>", il doit ajouter le fichier json dans le dossier statique prévu à cet effet et mettre à jour le chemin vers ce fichier dans la base de données. Si l'étiquetage est terminé, il faut mettre à jour l'état d'étiquetage de l'image dans la base de données comme étant terminé.

2.1.2 Besoins coté client

Besoin 3 : Ajouter une image au serveur

L'utilisateur peut ajouter une image à celles disponibles sur le serveur depuis son système de fichier. Cet ajout devra se dérouler sur une page spécifique.

Besoin 4 : Visualiser la liste des images présentes sur le serveur

L'utilisateur peut consulter la liste des images présentes sur le serveur. Cette liste affichera les images en train d'être étiquetées et celles qui n'ont pas encore été commencées. Les images dont l'étiquetage a été réalisé ne seront pas affichées.

Besoin 5 : Commencer l'étiquetage d'une image

L'utilisateur peut commencer l'étiquetage d'une image depuis la page d'accueil dans la liste des images à étiqueter ou en se rendant directement sur la page dédié de l'étiqueteur où la première image disponible est choisie. L'image est alors marquée comme étant en cours d'étiquetage.

Besoin 6 : Reprendre l'étiquetage d'une image

Si l'utilisateur choisi de reprendre l'étiquetage d'une image, les pigments précédemment étiquetés devront être marqué comme tel afin d'éviter que l'utilisateur les étiquette de nouveau.

Besoin 7 : Terminer l'étiquetage d'une image

Lorsque l'étiquetage d'une image est effectué l'utilisateur confirme que l'étiquetage est terminé et un fichier json comportant les coordonnées des pixels étiquetés est créé. L'image est notée comme terminée dans la base de données, et supprimée de la liste des images à étiqueter de l'accueil.

Besoin 8 : Appliquer un filtre sur l'image

L'utilisateur peut appliquer un filtre faisant ressortir les pigments présents sur l'image pour une teinte de couleur choisie. L'application du filtre doit être temporaire, l'image d'origine reste inchangée.

Besoin 9 : Translation dans l'image

L'utilisateur peut se déplacer sur l'image afin d'être plus précis sur l'étiquetage.

Besoin 10 : Zoomer dans l'image

L'utilisateur a la possibilité de zoomer dans l'image pour lui permettre une sélection des pixels plus précise.

Besoin 11 : Étiqueter un pigment

L'utilisateur peut sélectionner un ou plusieurs pixels pour étiqueter un pigment trouvé dans l'image. Une fois l'ensemble des pixels du pigment étiquetés, il confirme sa sélection et passe à un autre pigment. A chaque fois que l'utilisateur confirme l'étiquetage d'un pigment, le json associé à l'image est mise à jour avec la nouvelle zone. De cette façon la progression de l'étiquetage est sauvegardé à chaque nouveaux pigments validés.

2.2 Besoins liés au formatage du jeu de données

Comme dit précédemment, la sortie de l'étiqueteur est un ensemble de fichier json. Or les réseaux de neurones que nous allons utiliser (réalisant de la segmentation) ne prennent en entrée que des images et une vérité terrain. Cette dernière est sous la forme de masques qui sont des images en niveaux de gris encodant la classe de chacun des pixels de l'image d'origine par un valeur de gris.

Les images de départ étant d'une très grande résolution, il est impossible de les utiliser telles quelles. Ce module doit donc répondre à ces différents problèmes.

Besoin 12 : Lire des fichiers json

Puisqu'en sortie de l'étiqueteur nous obtenons des fichiers json contenant les coordonnées des pixels de chaque pigments et cela pour chaque image, il faut que le générateur du jeu de données soit capable de lire ces fichiers. En plus de les lire il faudra qu'il puisse les décomposer pour pouvoir les convertir en objet manipulable pour la suite des traitements. Le répertoire contenant les fichiers json sera spécifié par l'utilisateur.

Besoin 13 : Générer des masques en niveau de gris

Le générateur de jeu de données doit être capable de générer des masques (images au format png), d'après la décomposition d'un fichier json. Cette génération devra respecter ces sous-besoins :

Sous-besoin 13.1 : Découpage de l'image d'origine

Comme énoncé précédemment les images doivent être découpées pour que l'entraînement du réseau de neurones s'effectue dans un laps de temps

acceptable. La taille des sous-images (patchs) et l'emplacement des images d'origines, devront être passés en paramètre, pour que l'utilisateur les choisisse lui-même. Ces sous-images comprendront le nom de l'image dont elles sont issues, suivit d'un numéro unique.

Exemple : _DSC4017_23.png où _DSC4017 est le nom de l'image d'origine et 23 représente le numéro de la sous-image.

Sous-besoin 13.2 : Génération du masque d'une sous-image

Chaque pigment d'une sous-image se verra attribué une nuance de gris (255 possible). La couleur noir indiquera l'absence de pigment, et toute valeur supérieur à zéro indiquera la présence d'un pigment. Le résultat sera une image au format png comprenant le nom que la sous-image depuis laquelle il a été généré.

Exemple : M_DSC4017_23.png où le préfixe M signifie qu'il s'agit d'un masque et _DSC4017_23 est le nom de la sous-image.

Besoin 14 : Générer le jeu de données

Une fois que les couples (image-masque) ont été réalisé, le générateur de jeu de données doit constituer un ensemble de couple en fonction d'un pourcentage de couple positif (couple où un pigment est présent dans la sous-image). Ce pourcentage devra être passé en paramètre, pour que l'utilisateur puisse le choisir. L'utilisateur spécifiera aussi le dossier de destination où seront placé les sous-images et les masques en fin de traitement.

2.3 Besoin liés à l'apprentissage automatique

Ces besoins correspondent aux fonctionnalités à inclure dans les modules relatifs à l'entraînement et l'utilisation des modèles d'apprentissage automatique et ont été déterminés en fonction des besoins exprimés par les clients.

Besoin 15 : Entraîner un modèle

Le programme doit permettre à l'utilisateur d'entraîner le modèle avec les images et les paramètres de son choix. Les données pourront au préalable être formatées avec le module de génération de jeu de données et les paramètres sont le nombre d'époques (epochs) et la taille des lots (batch size). Le modèle ainsi entraîné devra être enregistré sur le disque à un emplacement décidé par l'utilisateur.

Besoin 16 : Générer un rapport d'entraînement

Après l'entraînement d'un modèle les données relatives à l'entraînement doivent être consignées dans un rapport au format HTML. Ces données incluent : les paramètres utilisés pour l'apprentissage (jeu de données, nombre d'epochs, batch size, etc.), les résultats d'évaluation du modèle (matrice de confusion, précision, recall) ainsi que des exemples de prédictions réalisées par le modèle sur le jeu de test (sous forme d'image comparant la vérité terrain et la prédiction).

Besoin 17 : Évaluer des images

L'application devra permettre à l'utilisateur d'utiliser un modèle pour évaluer de nouvelles images de tailles quelconques. Les images seront contenues dans un dossier indiqué par l'utilisateur. Le modèle à utiliser sera également indiqué par l'utilisateur. Le programme devra produire des masques en niveaux de gris rendant compte de la prédiction du modèle pour l'image correspondante. Les masques devront avoir les même dimensions que les images correspondantes. Les masques seront sauvegardés sur le disque dans un dossier indiqué par l'utilisateur.

2.4 Autres Besoins

Dans la globalité, les discussions avec nos clients nous ont permis de déceler certains besoins non fonctionnels pour notre application.

Besoin 18 : Ergonomique et intuitif

L'utilisation de l'application doit être simplifiée au maximum car elle va être utilisée par des personnes qui n'ont pas forcément beaucoup de connaissances en informatique.

Besoin 19 : Déploiement simple et portabilité

L'application doit être facilement déployable sur différentes machines(avec différents systèmes d'exploitations par exemple). En effet, celle-ci pourrait être utilisée par plusieurs personnes, sur des machines personnelles lors de déplacement, sur le terrain lors de la prise de nouvelles photographies. Cela implique que les différents modules de notre application soient regroupés et idéalement que le langage de programmation soit commun à tous les modules.

3 Conception

3.1 Application générale

Dans le but de rendre une application portable et ergonomique à nos clients, nous avons décidé en amont de réaliser nos différents modules en python. Cela simplifie grandement l'installation et l'utilisation de notre application. Les trois différents modules pourront être exécutés depuis un unique fichier python, avec une ligne de commandes ayant différents arguments pour chaque module.

voici l'architecture générale de l'application :

```
PigmentsDetection
    └── main.py
    └── modules
        ├── dataset-generator
        ├── evaluate
        ├── training
        └── web-labeler
```

Dans l'idéal, un dernier argument devrait être ajouté pour afficher une aide (l'argument -h ou –help par exemple) afin de préciser et faciliter l'usage de l'application en ligne de commande. De plus, notre application nécessitant de nombreuses bibliothèques externes pour fonctionner, un manuel utilisateur est nécessaire afin de faciliter l'installation de l'application, avec un environnement virtuel python pour éviter les conflits. Nous devons aussi réfléchir à une manière simple d'installer toutes les dépendances de bibliothèque facilement lors d'une nouvelle installation.

Voici un schéma représentant la chaîne de traitement de notre application générale, ainsi que les entrées et sorties de chaque module :

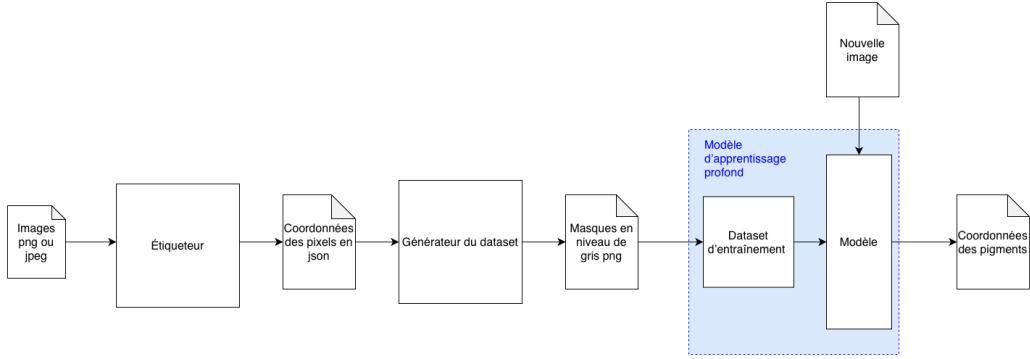


FIGURE 2 – Chaîne de traitement de notre projet.

3.2 Étiqueteur

3.2.1 Entrées et sorties de notre application

Notre application a pour but de recevoir des images en entrée, de format JPG ou PNG, de permettre ensuite l'étiquetage des pixels bleus de ces images afin de récupérer les coordonnées. Nous avons donc choisi de créer des fichiers jsons en sortie de notre application contenant ces coordonnées. Ce format de fichier nous a été demandé par le client lors d'une de nos réunions car si notre modèle d'apprentissage n'arrivait pas à converger vers des résultats satisfaisant, il était plus simple pour eux de traiter des fichiers json, vu la multitude d'outils pour les lire.

3.2.2 Choix de la technologie utilisée

Comme dit dans la partie d'analyse nous avons fait le choix de créer une application web, facilement déployable sur le réseau du pôle archéologie et accessible depuis plusieurs machines de ce réseau. Une application avec une architecture client-serveur semblait être le choix idéal pour nos besoins. De plus, la mise en place et l'utilisation d'intelligence artificielle se faisant principalement en python, le choix de ce langage a été une évidence afin de pouvoir, à terme, regrouper nos différentes applications(étiqueteur, générateur de jeu de données, réseau de neurones) en une seule application générale.

Ayant peu de temps pour mettre en place cette application, nous avons décidé d'utiliser le framework web **Django** [6], qui permet une mise en

place très simple et rapide d'une application web en python.

3.2.3 Architecture générale d'un serveur Django

Pour la partie backend, un serveur Django respecte un pattern de type Modèle-Vue-Template qui est un très bon avantage pour notre application. C'est un pattern de conception dérivé du pattern très connu **Modèle-Vue-Controller**. Dans ce pattern dérivé, le contrôleur est géré directement par le framework **Django**, et n'a pas besoin d'être implémenté par le développeur.

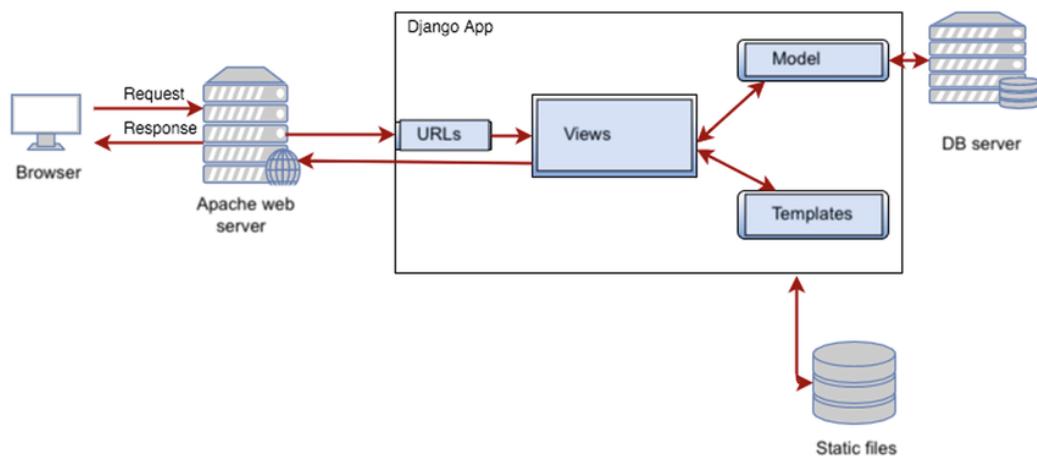


FIGURE 3 – Schéma de l'architecture générale d'un serveur Django. [7]

Un serveur Django possède un serveur Apache pour gérer les requêtes http, une base de données SQLite par défaut et que nous utilisons pour notre projet, ainsi que des dossiers de fichier statiques pour stocker des fichiers(nos images, les jsons créés, fichiers css et javascript). Comme dit précédemment, le cœur d'une application Django est basée sur le pattern MVT, il y a 4 types fichiers python principaux (qui pourraient être considérés comme des packages par exemple en Java) :

- **Model** : en lien direct avec la base de données(une classe de **Model** correspond à une table de la base de données),
- **Views** : exécute la logique métier et interagit avec le modèle. Accepte des requêtes HTTP, et renvoie des réponses HTTP.
- **Templates** : Correspond aux différents designs (pages HTML) de l'application, récupérés par les fonctions de **Views** pour y incorporer

- les données de **Model** avant de renvoyer la vue finale à l'utilisateur.
- **URLs** : Fait office de routeur pour effectuer les fonctions de **Views** relativement aux requêtes reçues par le serveur Apache.

Relativement à notre cahier des besoins définit plus tôt, nous pouvons déjà avoir une idée de ce que vont contenir ces fichiers.

Model

Notre modèle est très simple, nous devons seulement stocker des images. Nous aurons donc une classe Image dans le fichier **Model**, et une table correspondante dans la base.

Templates

Nous allons avoir plusieurs templates sur notre site web, idéalement une page d'accueil avec les images à étiqueter, une page pour étiqueter les pixels sur les images et une page pour envoyer des images sur le serveur. Chacun de ces templates possédera donc son propre fichier HTML, et seront toutes regroupés dans un seul répertoire nommé "templates".

Views

Nous aurons besoin de plusieurs fonctions pour rendre nos templates et interagir avec notre modèle :

- Ajouter des images à la base de données, puis réafficher la page d'envoi d'images sur le serveur,
- Récupérer la liste des images dans la base de données, puis afficher la page d'accueil,
- Récupérer une image de la base de données (son URL), puis l'afficher dans la page d'étiquetage,
- Ajouter un fichier Json de l'étiquetage d'une image, puis passer à l'image suivante à étiqueter.

URLs

Le routeur contiendra des routes vers les différentes pages de **Templates**.

3.2.4 Maquettes de l'Étiqueteur

Pour permettre une meilleure réalisation de l'étiqueteur nous avons conçu des maquettes du site.

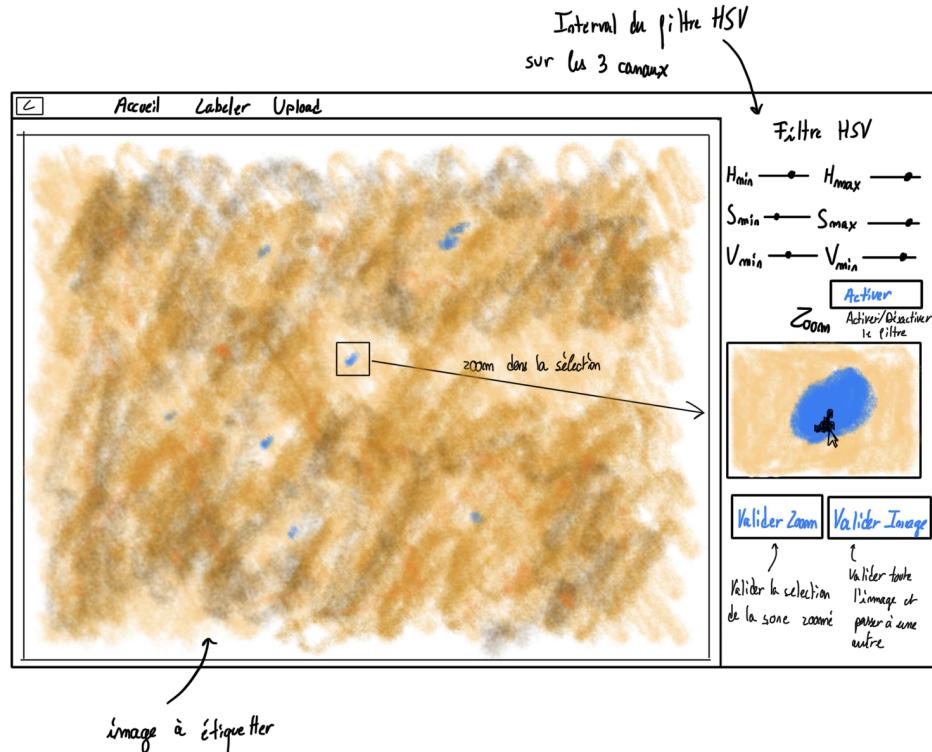


FIGURE 4 – Maquette de la page où se déroule l'étiquetage nomée "Labeler"

On peut voir que l'image à étiqueter occupe un grand espace dans la page. Sur le côté droit on retrouve une barre de menu avec la gestion du filtre et la possibilité d'afficher une zone sélectionnée dans l'image. C'est dans cette zone que la sélection de pixels devra se dérouler. Et finalement on retrouve deux boutons pour valider soit la sélection (pigment) en cours, ou toutes l'image.

En voici une autre représentant la page lorsque l'utilisateur active le filtre.

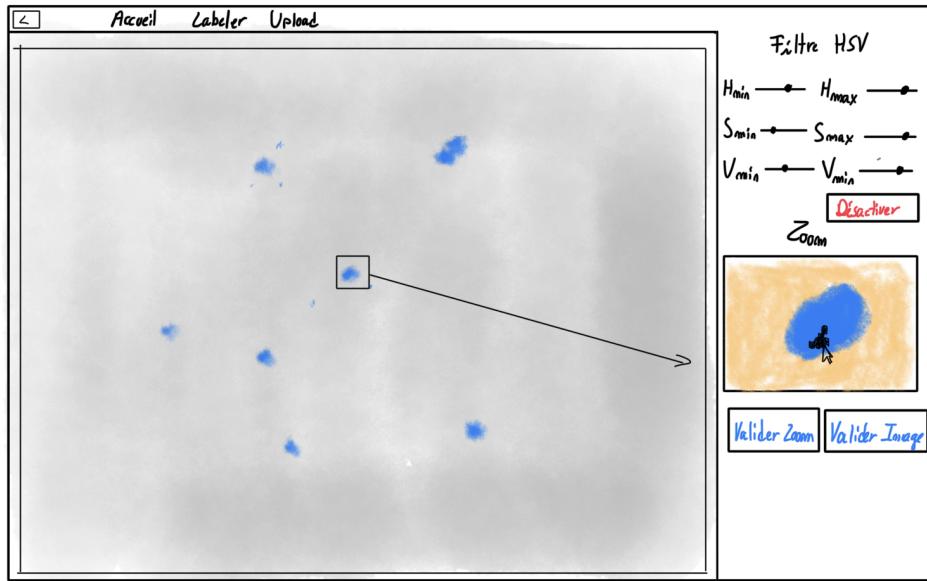


FIGURE 5 – Maquette de la page Labeler avec le filtre activé

Lorsque l'utilisateur active le filtre, les pigments doivent être facilement repérable. Le filtre ne doit pas s'appliquer dans la zone de sélection.

3.3 Générateur du jeu de données

Nous avons donc décidé de réaliser un module python permettant de découper toutes les images du set d'entraînement en plusieurs patchs de taille plus raisonnable (512x512 pixels, voir 256x256), pour que notre réseau puisse accepter le set d'entraînement. Cette taille pourra être entrée par l'utilisateur. Le module devra également construire les masques correspondants aux images de plus faible taille à partir de la vérité terrain encodée dans les fichiers Json. De manière standard, les masques ainsi générés encoderont les pixels correspondant à des pigments en niveaux de gris, et en noir et les autres pixels (correspondant à de la pierre). Suivant si l'utilisateur souhaite réaliser de la segmentation sémantique ou de la segmentation d'instances, les pixels correspondant aux pigments devront être respectivement encodés en blanc

ou chacun avec un niveau de gris propre à chaque groupe de pixels (à chaque pigment). Les pixels appartenant à un même pigment seront déterminés à l'aide d'un algorithme de "region -rowing".

Nous avons aussi besoin d'une certaine modularité pour ce problème afin de pouvoir mettre en forme le set d'entraînement de différentes manières(en changeant la taille des patchs par exemple) dans le but de trouver les paramètres optimaux pour l'entraînement du réseau.

3.4 Modèle d'apprentissage profond

Il nous faut maintenant décider du modèle d'apprentissage que nous allons utiliser pour segmenter nos images. Plus précisément, les archéologues souhaitent pouvoir obtenir pour une image donnée, une image en niveaux de gris indiquant les pixels identifiés comme étant des pigments. Dans notre cas on peut s'orienter soit vers un modèle réalisant de la segmentation sémantique soit de la segmentation d'instance. La segmentation sémantique consiste à classifier chaque pixel selon la classe à laquelle il appartient (ici 'pigment' ou 'autre'). La segmentation d'instances en revanche cherche à identifier les éléments uniques appartenant à une image (ici des groupes ou de pigments ou taches) puis à les classifier.

Pour notre problème les deux approches sont valides car on ne cherche pas forcement à identifier les taches de pigments de manière uniques donc cela nous ouvre 2 pistes pour mettre au point notre modèle.

3.4.1 Approche par segmentation sémantique (U-net)

Pour l'approche par segmentation sémantique nous envisageons d'utiliser un réseau de neurones U-net [2] qui a la particularité de présenter de bons résultats dans sur des problèmes semblables au notre [3] et de produire en sortie des masques correspondant à ce qu'ont demandé les archéologues. Les réseaux U-net fonctionnent selon un principe d'"Encodeur-Decodage" : Une première partie du réseau, l'encodeur, va effectuer plusieurs divisions successives de l'image en entrée, en extrayant à chaque fois des caractéristiques de ces sous-images. La seconde partie du réseau, le décodeur, va ensuite construire le masque de prédiction en se basant successivement sur les caractéristiques extraites par le décodeur, de celles des plus petites subdivisions, à celles des plus grandes. Un schéma illustrant ce fonctionnement se trouve en annexe 7.1

3.4.2 Approche par segmentation d'instances (Mask R-CNN)

Pour l'approche par segmentation d'instances nous envisageons d'utiliser un réseau de neurones Mask R-CNN [4]. Cette architecture de réseau se base sur le R-CNN (region-based convolutional neural network ou réseau de neurones convolutif basé sur les régions) qui est un réseau de détection d'objet qui prédit et classe des boîtes englobantes rectangulaires pour des objets dans des images. Le Mask R-CNN a la particularité de prédire un masque pour chacun des objets dans les images en plus des prédictions faites par ce R-CNN classique. Le principe général de ce réseau est le suivant : une première partie du réseau se charge de prédire les boîtes englobantes des différents objets de l'image (ici les taches de pigments mais éventuellement aussi d'autres détails dans la roche) et d'en extraire des caractéristiques à l'aide d'une opération de RoIPool (Region of Interest Pooling). Ces caractéristiques sont ensuite utilisées par la seconde partie du réseau afin d'affiner les boîtes englobantes, de les classifier en fonction de leur contenu et de calculer les masques correspondants aux objets situés dans les boîtes. Ce modèle prédit donc un masque pour chacun des objets dans l'image, ce qui nous permettrait ensuite de reconstituer un masque unique indiquant tous les pixels prédits comme étant des pigments. Ce réseau étant plus complexe que le précédent il pourrait être pertinent de partir d'un modèle déjà entraîné afin de simplement lui faire apprendre les spécificités de notre problème.

3.4.3 Pré-traitements et post-traitements

Pour chacun des réseaux évoqués il conviendra d'extraire les données nécessaires à l'entraînement et de les convertir en tenseurs utilisables par le réseau. Parmis les données extraire il faudra notamment, pour le Mask R-CNN, déterminer les boîtes englobantes des taches de pigment présentes dans le jeu d'entraînement. Ces opérations pourront être réalisées à la volée de manière algorithmique en parcourant les masques créés par le générateur de jeu de données décrit plus haut. Enfin, il faudra mettre au point pour chacun des réseaux une chaîne de post-traitement permettant à partir des tenseurs en sortie du réseau, de recréer une image masque correspondant aux prédictions du réseau.

3.4.4 Déséquilibre de classe et choix de la fonction objectif

Notre jeu de donnée présente un très fort déséquilibre de classe : moins de 1% des pixels présents correspondent à des pigments et plus de 90% des patchs de taille 256 ne présentent aucune trace de pigment. Cela signifie que si notre modèle ne prédit que des masques noires (aucun pigment) il atteindra tout de même un précision de plus de 0.99. Pour empêcher cela, une première approche peut être de supprimer du jeu de données un certain nombre d'images ne contenant pas de pigment. On peut estimer qu'il ne faut pas que les patchs sans pigments représentent plus de 60% du set d'apprentissage. Ce paramètre sera implémenté par le générateur de jeu de donnée. Même si les patchs contenant des pigments représentent une bonne partie du jeu d'entraînement, les pixels représentant des pigments, au sein d'une image, restent très minoritaires. Pour que cela n'influence pas trop notre modèle, il convient de définir une fonction objectif (ou fonction de coût) qui ne valorise pas trop les faux négatifs (les pixels correctement classifiés comme étant de la pierre). Parmis les fonction objectif utilisées pour la segmentation d'image présentées dans un survey de 2020 [8], nous en avons retenu 2 :

Dice loss

Cette fonction objectif présente l'avantage de ne diminuer que lorsque le modèle prédit correctement un pixel positif (pigment). Ainsi, la détection de pixels négatifs (pierre) n'est pas pénalisée mais ne fait pour autant pas diminuer la fonction objectif. Le seul moyen pour le modèle de s'améliorer est donc de prédire correctement des pigments.

Tversky loss

Cette fonction objectif est en quelque sorte une généralisation de la précédente. Elle permet cependant, en positionnant un coefficient β , de sanctionner plus fortement les faux négatifs ou les faux positifs. En sanctionnant plus fortement les faux négatifs, on peut espérer "dissuader" le modèle de prédire des masques complètement négatifs.

3.4.5 Évaluation des modèles

Pour évaluer la qualité de notre modèle, on souhaite pouvoir obtenir ses prédictions sur un jeu de test et calculer les valeurs suivantes :

- Le nombre de taches présentes dans la vérité terrain correctement prédites (TP)
- Le nombre de taches présentes dans la vérité terrain non prédites (FN)
- Le nombre de taches prédites n'étant pas dans la vérité terrain (FP)

Cela nous permettra de calculer la précision du modèle, qui rend compte de la pertinence des taches prédire :

$$Precision = \frac{TP}{TP+FP}$$

Ainsi que le "recall" qui indique la capacité du modèle à trouver les taches présentes dans la vérité terrain :

$$Recall = \frac{TP}{TP+FN}$$

Notre objectif sera principalement de minimiser le recall qui témoigne de la capacité du modèle à trouver les tâches présente dans la vérité terrain car les archéologues nous ont indiqué qu'ils préféraient obtenir des faux positifs plutôt que des faux négatifs.

Ces données pourront également nous permettre de créer une matrice de confusion.

Ces données d'évaluation ainsi que des données sur la phase d'entraînement seront consignées dans un rapport HTML.

3.4.6 Choix de la technologie

Nous allons implémenter notre programme d'entraînement en python avec le module Pytorch car non seulement c'est un module gratuit et open-source mais aussi car ce module est un standard dans la recherche sur les réseaux de neurones et la plupart des articles publiés sur le sujet sont accompagnés de programmes développés à l'aide de Pytorch que nous pourrons récupérer et adapter pour notre projet.

4 Réalisation

4.1 Architecture générale de l'application

L'architecture générale de l'application que nous avons réfléchi durant la conception à été respectée. Nous avons implémenté la possibilité de lancer le serveur web depuis l'application principale qui effectue un exec, tandis que les autres modules sont intégrés comme des bibliothèques appelées depuis le programme principal. De plus nous avons rédigé un manuel utilisateur afin de le délivrer à nos clients. Celui-ci est disponible en annexe de ce rapport. Afin de préciser l'utilisation de l'application, une commande d'aide a également été ajoutée à l'aide de la bibliothèque `argparse`.

Voici en détail les commandes disponibles de l'application :

4.1.1 Afficher l'aide

```
> python main.py -h
```

4.1.2 Lancer le serveur d'étiquetage

```
> python main.py runserver
```

4.1.3 Lancer le générateur de dataset

```
> python main.py generate-dataset
```

avec les arguments :

```
-i INPUT_DIR  
      Path to the directory containing  
      'images\' and 'jsons\' subfolders.  
  
-o OUTPUT_DIR  
      Path to the directory where the new  
      dataset will be written.  
  
(OPTIONAL) -s SUB_IMAGE_SIZE
```

Size of the images to be created
for the dataset (Default 256).

(OPTIONAL) -pp POSITIVE_PERCENTAGE
Percentage of positive images to include
in the generated dataset (Default 100%)

(OPTIONAL) -b If set, mask will be binary (black/white),
else each instance instance will be
encoded with a different shade of grey.

(OPTIONAL) -h show the help message.

4.1.4 Lancer l'entraînement du réseau

> python main.py train

avec les arguments :

-d DATASET_DIR
Path to the dataset to use for training.

-o OUTPUT_DIR
Path to the directory where the model
and report will be written.

-e EPOCHS
Number of epochs the model will be
trained for.

-b BATCH_SIZE
Number of epochs the model will be
trained for.

(OPTIONAL) -h show this help message.

4.1.5 Lancer une évaluation avec le réseau

> python main.py eval

avec les arguments :

```
-i INPUT_DIR  
        Path to the directory containing images  
        to evaluate.  
  
-o OUTPUT_DIR  
        Path to the directory where the predicted  
        masks will be written.  
-m MODEL  
        Pytorch model (.pt file) to use for  
        evaluation.  
  
(OPTIONAL) -s SUB_IMAGE_SIZE  
        Size of the cropped images to pass  
        to the model (Default 256).  
  
(OPTIONAL) -h  
        show this help message.
```

4.2 Réalisation de l'étiqueteur

4.2.1 Implémentation de l'architecture Django

Pour décrire notre implémentation de l'application de Django en accord avec nos besoins, nous allons décrire chaque fichiers principaux énumérés dans la partie conception.

Model

Le fichier Model de l'application Django se compose en Classe. Chaque classe inscrite dans le Model va correspondre à une table de la base de données. Dans notre cas nous n'avons à gérer seulement des images donc une seule table dans la base et une seule classe dans le model. Notre classe possède 5 attributs :

1. **ID** : l'id est un attribut par défaut créé par Django afin de distinguer les objets de la classe. Chaque nouvel objet se voit attribuer un ID unique, celui ci va servir de clé primaire dans la base de données.
2. **title** : Nom de la photo stockée dans le serveur.

3. **type** : Le type de fichier de la photo, principalement jpg ou png.
4. **img** : Objet de type ImageField(objet Django) caractérisant une image, contenant principalement le chemin relatif de stockage de l'image dans le dossier statique prévu à cet effet.
5. **json** : Objet de type FileField(objet Django). Utilise le même principe que ImageField mais pour stocker toute sorte de fichier. Cet attribut est optionnel, au début les images n'ont pas de json associés(lorsqu'elles ne sont pas encore étiquetées).
6. **workingStatus** : chaîne de caractère permettant de définir l'état du traitement de l'image, il y a trois états possibles("Unlabeled", "InProgress", "Labeled").

Templates

Les templates sont les squelettes de nos pages html. Ils sont récupérés par les fonctions du fichier Views afin de les afficher. Voici la liste de nos templates :

1. **accueil.html** : La page d'accueil de notre application web, elle est utilisée pour afficher les images à étiqueter.
2. **labeler.html** : La page la plus importante de notre application, c'est sur celle-ci que sont effectués les étiquetages des images.
3. **upload.html** : La page permettant de téléverser de nouvelles images sur le serveur.

Views

Le fichier Views contient toutes les fonctions de gestion du modèle. c'est toutes ces fonctions qui récupèrent les templates pour y incorporer les données du modèle avant de retourner la vue complète à l'utilisateur. Elles sont appelées lorsque le serveur reçoit des requêtes http. voici les différentes fonctions :

1. **accueil** : Récupère toutes les images non étiquetées et en cours d'étiquetage pour les incorporer au template **accueil.html** pour afficher la page d'accueil.
2. **labeler** : Possède un id optionnel en paramètre, récupère l'image correspondant à cet id pour l'incorporer au template **labeler.html**

pour afficher la page d'étiquetage. Si l'id n'est pas défini, la première image disponible pour l'étiquetage est incorporée.

3. `uploadRequest` : Ajoute toutes les images présentes dans la requête à la base de données avant de renvoyer le template `upload.html` pour afficher la page de téléversement d'images vers le serveur. Si la requête ne contient pas d'images, on affiche simplement la page de téléversement.
4. `uploadJson` : possède obligatoirement un id, ajoute le fichier json contenu dans la requête à l'image ayant comme id celui en paramètre dans la base de données. Une redirection est ensuite effectuée sur la page d'étiquetage avec l'image à étiqueter suivante.

URLs

Le fichier URLs fait office de routeur, c'est celui-ci qui va appeler les bonnes fonctions du fichier Views en fonction de la requête reçue par le serveur Apache.

Voici les routes existantes :

1. '' : La route par défaut, elle redirige l'utilisateur directement à la page d'accueil de l'application.
2. '/upload' : La route d'accès à la page de téléversement de nouvelles images sur le serveur.
3. '/labeler' : La route d'accès à la page d'étiquetage, affichant une image à étiqueter disponible dans le serveur.
4. 'labeler/<int:id>/' : La même route que la précédente, mais affichant l'image à étiqueter correspondant à l'id de la route.

4.2.2 Fonctionnalités

L'ensemble des besoins détaillés dans notre parties de conception ont bien été implémentés. Une fois lancé l'Étiqueteur est accessible via le port 8000 de localhost. L'application est donc composée de trois pages :

Accueil : Sur cette page on retrouve l'ensemble des images qui sont en train d'être étiqueté et celles qui n'ont pas encore été commencé.

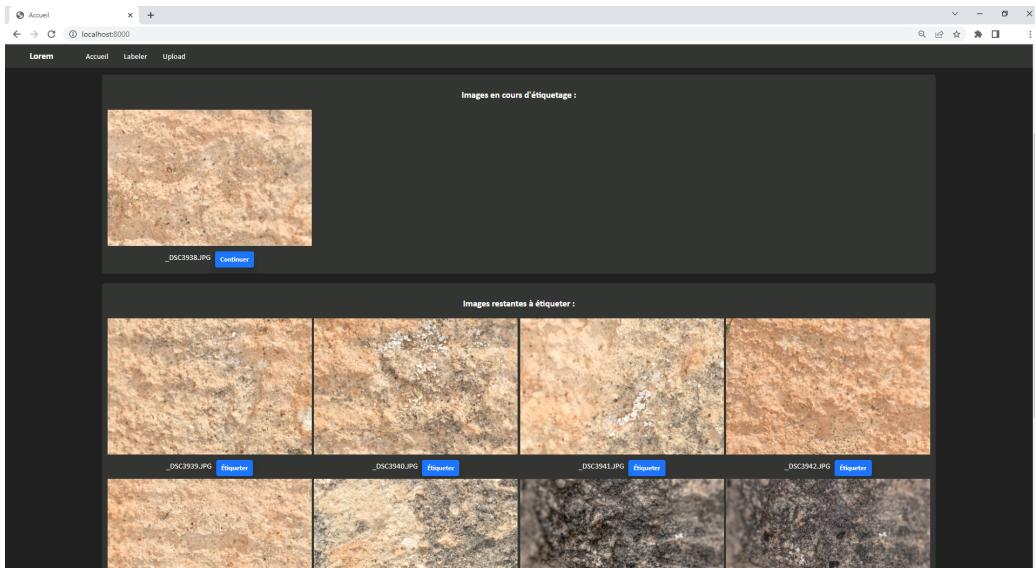


FIGURE 6 – Page d'accueil.

Lors du rendu de la page depuis le backend Django, l'ensemble des images est récupéré. Cet ensemble est ensuite filtré pour ne garder que les images en cours d'étiquetage (premier bloc de la page) et celles dont l'étiquetage n'a pas encore commencé (deuxième bloc). Cette opération de filtrage est réalisé dans le template de la page côté backend (serveur Django). Les images étant d'une résolution élevée, leurs affichages sur la page est optimisé à l'aide de propriétés HTML et CSS sur les éléments *img*. Nous avons utilisé un rendu asynchrone à l'aide de la propriété *decoding="async"* et un rendu "paresseux" (l'image n'est chargé que si elle visible, à l'aide de la propriété *loading="lazy"* et *visibility="auto"*).

Labeler : C'est sur cette page que l'étiquetage se déroule. On peut y accéder depuis la barre de navigation ou depuis l'accueil.

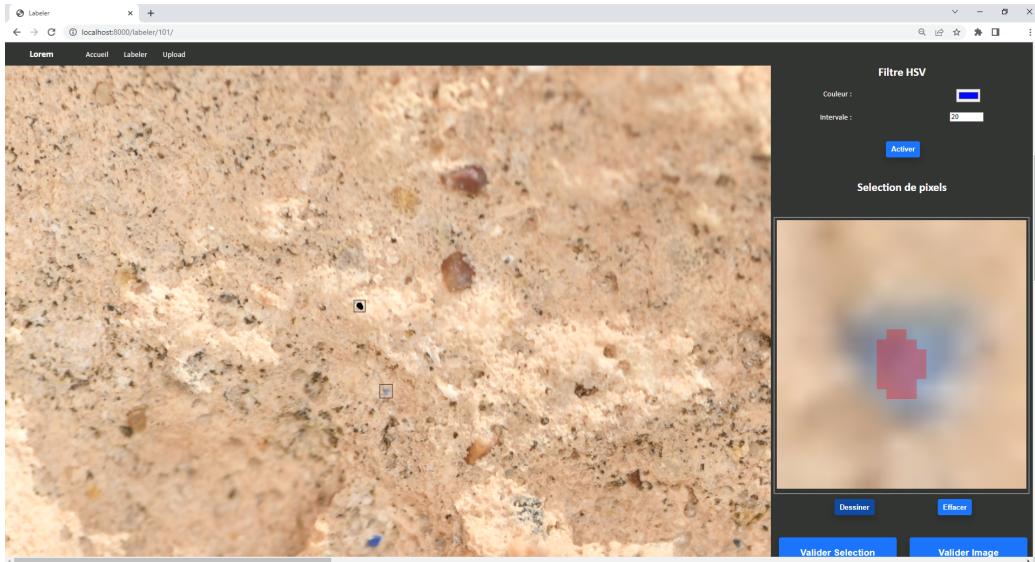


FIGURE 7 – Page Labeler.

On retrouve deux grandes parties sur cette page :

- Image à étiqueter, dont on ne voit qu'une partie ici et qui occupe 75% de la page. La navigation dans l'image se fait de la même façon que pour une page classique. Les techniques utilisées pour l'affichage des images et de la sélection des pixels est détaillé dans la prochaine partie 4.2.3.
- Barre latérale, dans laquelle on retrouve le filtre HSV , le zoom dans l'image à étiqueter (sélecteur de pixels), et les boutons de validation. Le sélecteur de la teinte à filtrer est un élément `input[type=color]`, incluant directement une représentation HSV et une pipette. C'est dans le zoom dans l'image que s'effectue l'étiquetage, avec la possibilité de peindre les pixels un à un, et aussi d'effacer la sélection. Les boutons *Valider Sélection* et *Valider Image* sont respectivement là pour finaliser l'étiquetage d'un pigment ou de l'image en entière.

Upload : Cette page est responsable de l'ajout d'image(s) au serveur.

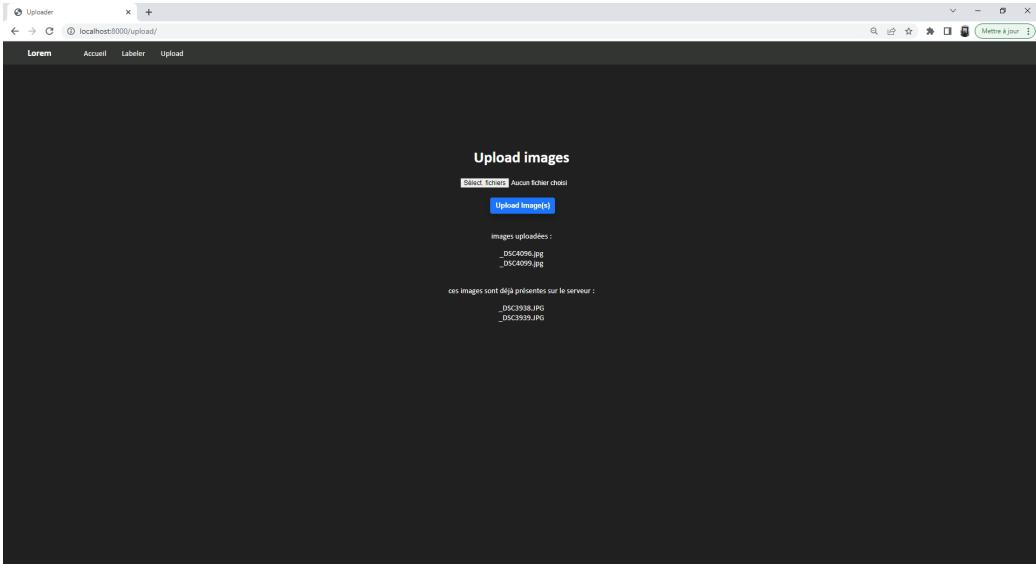


FIGURE 8 – Page Upload.

Le téléversement d'image sur le serveur se fait au travers d'un formulaire HTML. Il y a donc deux champ à ce formulaire sous la forme de deux boutons, le premier en gris est là pour sélectionner les images à l'aide de l'explorateur de fichier, le second en bleu est là pour valider la sélection et commencer le téléversement. L'utilisateur peut choisir une ou plusieurs images. Une fois cliqué sur *Upload image(s)*, les images sont envoyées au backend pour être enregistrées dans le modèle. La page est ensuite rechargée pour afficher les noms des images qui ont bien été téléchargées. Si une image était déjà présente dans la base de données elle n'est pas ajoutée et elle est citée comme étant déjà présente.

Comme énoncé dans les besoins, à chaque fois que l'utilisateur effectue l'étiquetage d'un pigment, un fichier json est créé ou mis à jour. Cette opération est donc réalisée à chaque fois que l'utilisateur appuie sur le bouton *Valider Sélection* de la page de l'Étiqueteur. Nous avons donc défini l'architecture du fichier json dont voici un exemple :

```
{
  "width": 6192,
  "height": 4128,
  "area_count": 10,
  "RGB3-7-252": {
    "area0": "[[239,1890],[240,1890],[240,1889],[240,1888],[241,1888],[241,1887]]",
    "area1": "[[636,25],[637,24],[637,23],[637,22],[638,22],[639,23],[639,24]]",
    "area2": "[[1689,639],[1689,638],[1690,637],[1690,638],[1690,639],[1690,640]]",
    "area3": "[[2496,951],[2497,951],[2497,950],[2498,950],[2499,950],[2499,949]]",
    "area4": "[[2697,1381],[2698,1380],[2698,1379],[2699,1379],[2700,1379],[2700,1380]]",
    "area5": "[[2100,2692],[2101,2692],[2101,2691],[2102,2691],[2103,2691],[2103,2690]]",
    "area6": "[[2669,895],[2670,895],[2670,894],[2671,894],[2671,893],[2672,893],[2673,893]]",
    "area7": "[[3748,3312],[3749,3311],[3750,3310],[3750,3309],[3751,3309],[3751,3308]]",
    "area8": "[[3991,3371],[3991,3370],[3993,3369],[3994,3369],[3994,3368],[3994,3367]]",
    "area9": "[[4172,3632],[4172,3631],[4172,3630],[4172,3629],[4172,3628],[4171,3628]]"
  }
}
```

FIGURE 9 – Architecture d'un fichier json.

4.2.3 Difficultés rencontrées

La difficulté principale que nous avons rencontré dans la réalisation du frontend de l'étiqueteur fût la gestion de la sélection des pixels dans les images. Un point clé de l'application rendu difficile du fait de la grande résolution des images(6192x4098 minimum). Notre première approche fût de créer une grille à l'aide de la propriété css *display : grid*; sur une div superposée à l'image. Au chargement de la page on crée alors une case de la grille par pixel de l'image.

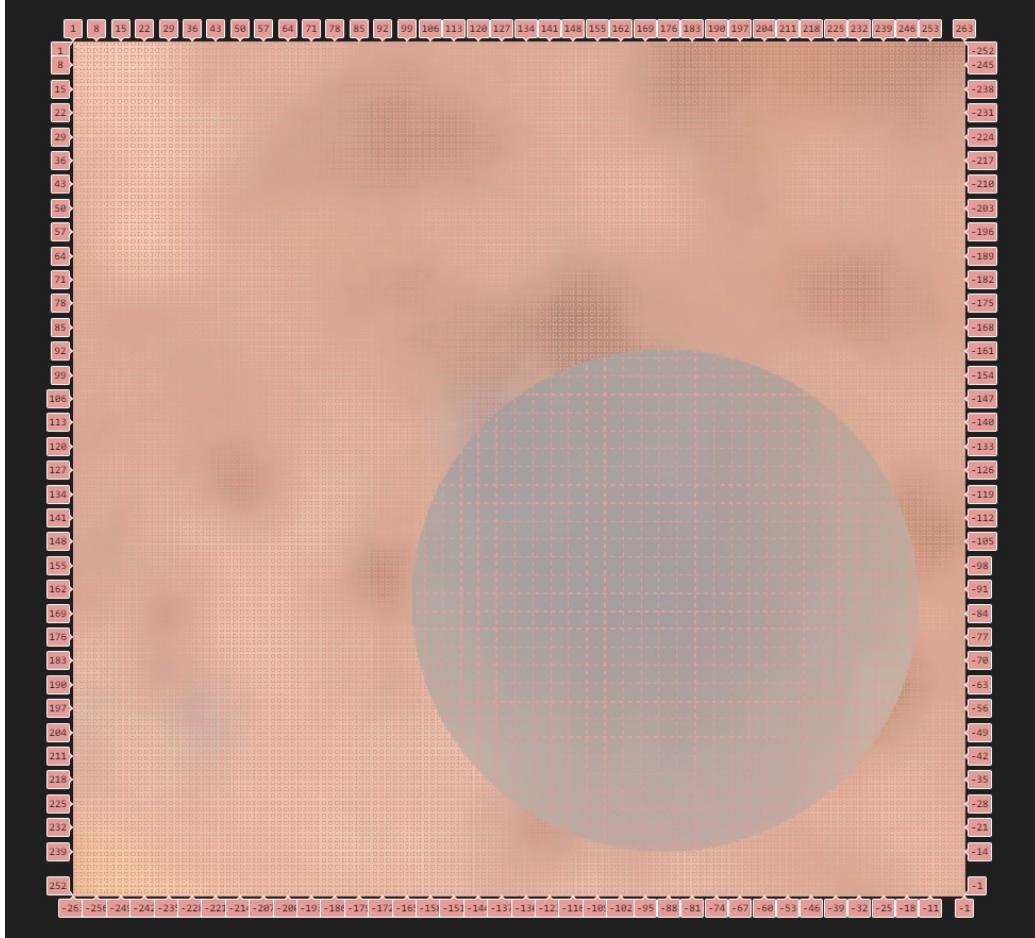


FIGURE 10 – Grille de la première version de l’étiqueteur avec une image test de 262x251 pixels.

Le problème avec cette technique est que créer autant d’éléments HTML (élément de la grille) pour chaque pixel de l’image est très demandant pour le navigateur. Sur nos images avec une résolution 4K où le nombre de pixel est supérieur à 25 000 000, le navigateur abandonne tous simplement le rendu de la page.

Nous avons donc choisi d’utiliser une autre technique en se basant sur l’élément html *canvas*. Cet élément permet de définir une zone de dessin dans laquelle on peut afficher des images. Nous avons donc choisi de superposer

deux canva pour afficher l'image en grand : un pour l'affichage de l'image appelé *canva_allImg* et un autre pour dessiner des zones où zoomer ainsi que les pigments une fois étiqueté appelé *canva_drawing*.

La partie sélectionnée par l'utilisateur dans le grand canva affichant l'image en entière est ensuite affichée dans un troisième élément canva dans la barre latérale. Sur ce canvas, appelé *canva_selectedImg*, est superposée une grille comme dans la première version de l'étiqueteur. De cette façon la grille est beaucoup plus petite et donc demande une quantité de ressource acceptable. Chaque élément de la grille est animé pour permettre à l'utilisateur une meilleure expérience.

4.3 Réalisation du générateur de jeu de données

4.3.1 Description

Nous avons réalisé le générateur du jeu de données en Python à l'aide de la bibliothèque Pillow pour réaliser les opérations de manipulation d'images.

4.3.2 Fonctionnalités

Ce module prend en entrée des images de tailles quelconques et des fichiers Json encodant la position des pigments et produit en sortie des couples patch-masque de plus petite taille, couvrant l'ensemble des images en entrée.

Comme défini lors de la conception, ce module permet de formater le jeu de données en laissant à l'utilisateur la possibilité de choisir :

- La taille des patchs et des masques à créer
- Le pourcentage de patchs positifs (au moins 1 pigment) dans le jeu de données à créer
- La possibilité ou non d'enregistrer les masques vides (sans pigments)
- La possibilité de générer des masque binaires (noir et blanc) plutôt que d'attribuer une valeur de gris à chaque tâche.

Voici un exemple de couple patch-masque créé par le module :

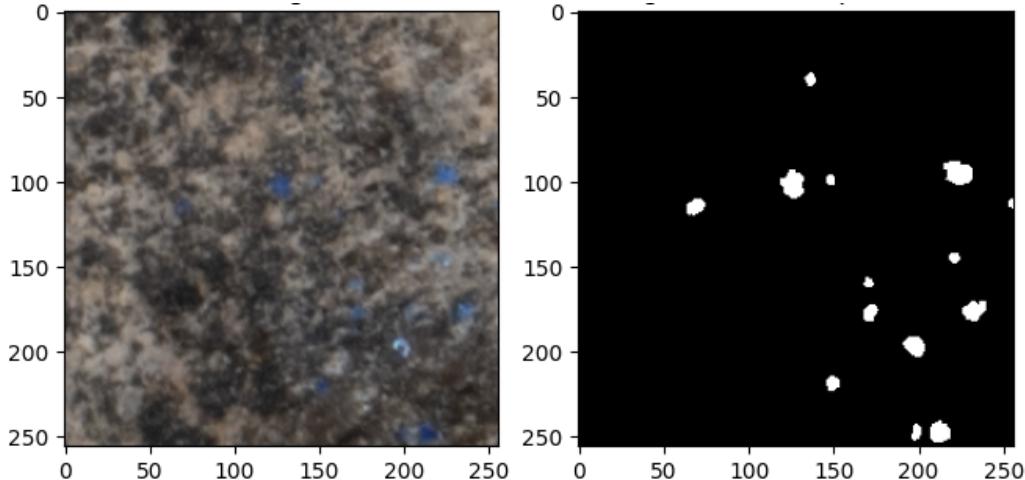


FIGURE 11 – Patch et masque binaire extraits à partir d'une image de grande taille et du fichier Json correspondant

4.3.3 Difficultés rencontrées

Dans la réalisation de ce module nous avons dû implémenter un algorithme de "region-growing" permettant d'identifier les pixels appartenant à une même tâche. Notre première implémentation était correcte sur le plan algorithmique mais utilisait des appels récursifs qui causaient un plantage de python lors du traitement de taches de grande taille. Nous avons donc du "dé-recursifier" cet algorithme pour qu'il fonctionne sur toutes les images.

4.4 Réalisation du modèle d'apprentissage profond

4.4.1 Unet

Nous avons tout d'abord souhaité implémenter et tester le réseau U-net [2]. Pour cela nous sommes parti du code publié avec l'article qui présentait une application du U-net à la détection de bâtiment [3]. Nous avons réécrit une bonne partie du code afin qu'il soit compatible avec notre jeu de données et les versions actuelles des bibliothèques. Un premier test avec les paramètre

utilisés dans l'article nous a permis d'obtenir un premier modèle. Cependant comme nous le craignons le modèle se contentait de prédire des masques noirs (sans pigment) quelque soit l'image en entrée. A partir de cela nous avons testé un grand nombre de modification afin d'essayer d'obtenir un modèle pertinent.

Parmi ces modification nous avons essayé :

- L'utilisation de la fonction de perte Dice Loss
- L'utilisation de la fonction de perte Tversky Loss avec différentes valeurs pour le coefficient β
- La modification de la taille des images (128, 256 et 512)
- Différentes valeurs pour le nombre d'epochs
- Différentes valeurs pour la taille des lots (batch size)
- L'ajout de couches de "batch normalization" après chaque couche de convolution
- Différentes valeurs pour le taux d'apprentissage
- La mise en place d'un taux d'apprentissage dynamique avec différentes valeurs de paramètres.

Aucune de ces modifications n'a permis d'obtenir autre chose qu'un modèle prédisant systématiquement des masques vides. Ne comprenant pas l'origine du problème nous nous sommes tournés vers le Mask R-CNN.

4.4.2 Mask R-CNN

Pour l'implémentation du Mask R-CNN nous avons encore une fois utilisé le code fournit avec l'article associé [4]. Cependant, le réseau étant plus complexe que le U-net, il nous a fallut du temps pour faire fonctionner le code avec notre jeu de données et les version actuelles des nombreuses bibliothèque utilisées. Nous avons également pu récupérer un modèle publié en même temps que l'article [4] qui a au préalable été entraîné sur le COCO Dataset [5] ce qui constitue un point de départ pour l'apprentissage. Une fois que nous avons réussi à faire fonctionner le programme nous avons pu lancer les premiers tests nous avons pu constater que les modèles produit était beaucoup plus pertinents que pour le U-net. Nous avons alors développé toute la partie relative à l'évaluation du modèle et la génération du rapport d'entraînement et d'évaluation. Parmi les fonctionnalités notables nous avons fait en sorte qu'au fil des epochs seul le meilleur modèle soit conservé. Enfin ce module permet à l'utilisateur de donner, grâce à l'interface en ligne de commandes, en entrée un jeu de données formatté ainsi que des paramètres

d'entraînement et d'obtenir en sortie un modèle entraîné sur ses données et un rapport récapitulant le déroulement de l'entraînement et les performances du modèle sur le jeu de test.

4.4.3 Rapport d'entraînement

Nous avons aussi essayé de regrouper dans le rapport le plus d'informations pertinentes possibles sur le modèle et l'entraînement. Le rapport contient les informations suivantes :

- La date et l'heure de création du rapport
- le jeu de donnée utilisé
- le nombre d'epochs pour l'entraînement
- La taille des lots (batch size) utilisée pour l'entraînement
- la taille du jeu de données d'entraînement
- la taille des images utilisées
- La durée de l'entraînement
- L'évolution de la valeur de la fonction objectif en fonction des epochs
- La matrice de confusion des prédictions du modèle sur le jeu de test
- La précision du modèle sur le jeu de test
- Le recall du modèle sur le jeu de test
- Des exemples de prédictions montrant l'image d'origine, le masque cible et le masque prédit.

Des captures d'écrans d'un rapport type sont disponibles en annexe 7.2.1

4.4.4 Prédiction sur de nouvelles images

Enfin nous avons développé un petit module supplémentaire permettant à l'utilisateur de donner en entrée un ensemble d'images de taille quelconque et un modèle entraîné, et d'obtenir en sortie le masque des prédictions du modèle pour chaque image.

Exemple :

Si l'utilisateur donne en entrée une image comme celle-ci :



FIGURE 12 – Image type pour laquelle l'utilisateur pourrait souhaiter faire évaluer le modèle

Le module retourne un masque global représentant les prédition des positions des pigments faites par le modèle :



FIGURE 13 – Masque résultant de l'évaluation de l'image par le modèle

5 Résultats

Après avoir entraîné plusieurs modèles avec des paramètres différents, nous vous présentons ici le meilleur modèle que nous avons obtenu.

5.1 Entrainement

Ce modèle a été entraîné sur un jeu de donnée contenant 268 patchs de taille 256 et testé sur un échantillon de 67 patchs. Il a été entraîné pendant 12 époques avec une taille de lots (batch size) de 5.

Voici ici l'évolution de la fonction objectif au cours de l'entraînement :

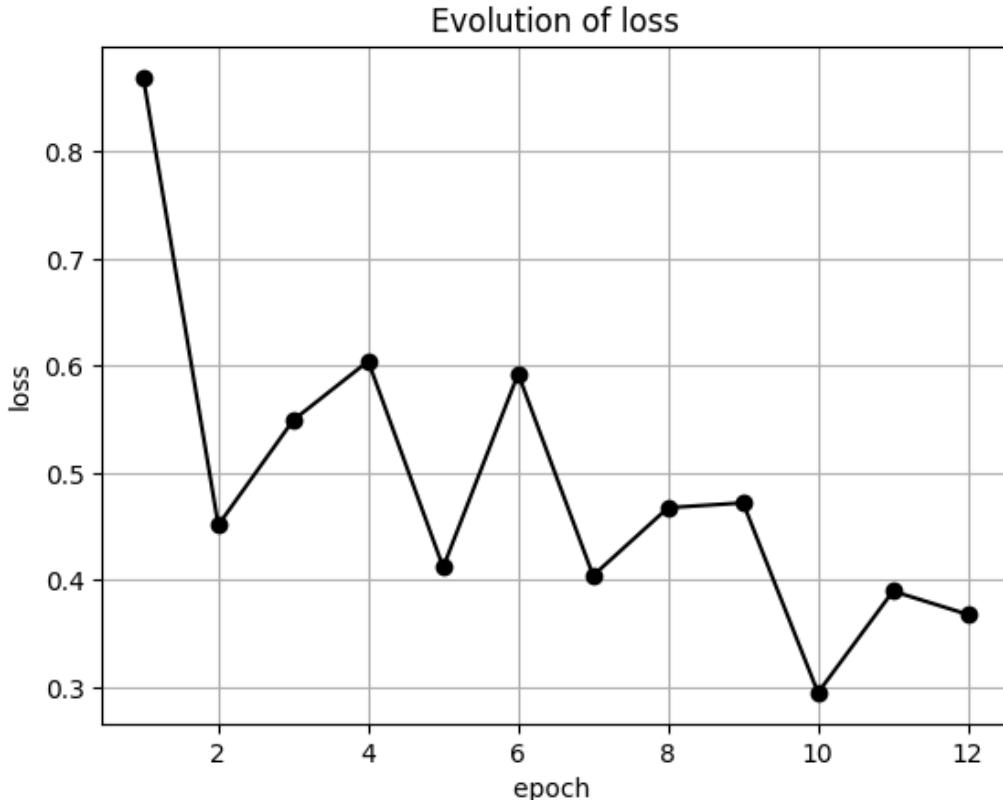


FIGURE 14 – Évolution de la fonction objectif en fonction des epochs (image extraite du rapport d'entraînement)

Même si on constate un évolution en "dents de scie" la fonction objectif diminue au fil des epochs ce qui indique que notre modèle s'améliore au fur et à mesure de l'entraînement. Aussi ici le minimum a été atteint à l'epoch 10, c'est donc à ce moment là que le modèle final a été sauvegardé.

5.2 Évaluation du modèle

Nous avons évalué notre modèle sur un panel de 67 images inconnues du modèle en comparant la position des taches de pigments prédites à celles présentes dans la vérité terrain. Nous n'évaluons pas la forme précise de ces

pigments car les archéologues nous ont indiqué qu'ils s'intéressent surtout à la positions des taches prédites bien plus qu'à leurs formes. On peut regrouper ces résultats dans une matrice de confusion permettant de comparer le nombre de pigments correctement prédis aux nombres de pigments incorrectement prédis :

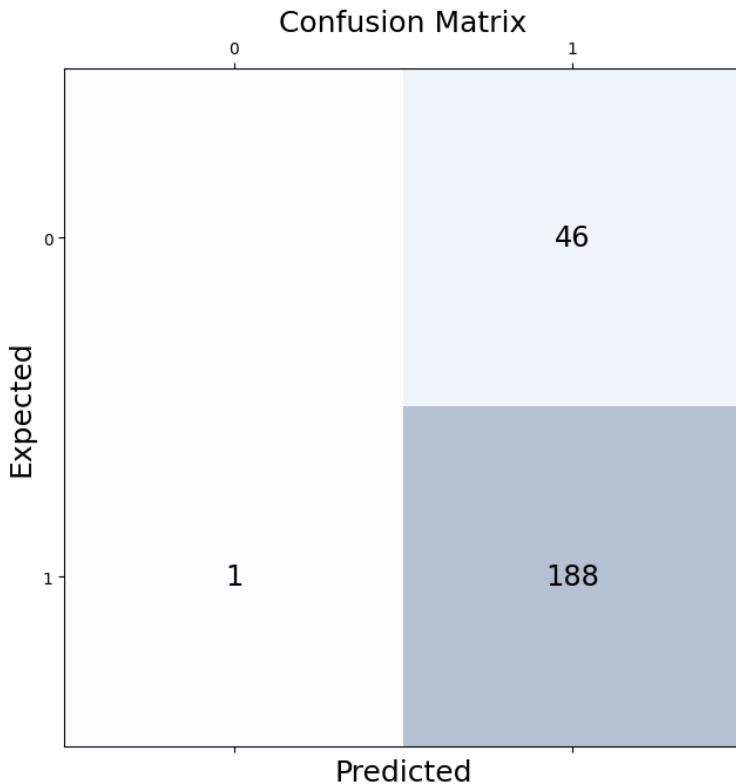


FIGURE 15 – Matrice de confusion des taches prédites sur le jeu de test (image extraite du rapport d'entraînement)

On observe les résultats suivants :

- 188 des 189 pigments présents dans la vérité terrain ont correctement été identifiés par le modèle (188 TP).
- 46 des 234 pigments prédits ne sont pas renseignés dans la vérité terrain et sont donc probablement des erreurs (46 FP).
- Un seul des 189 pigments présents dans la vérité terrain n'a pas été identifié par le modèle.

Ces valeur nous permettent de calculer la précision est le recall du modèle :

$$Precision = 0.80$$

$$Recall = 0.99$$

On obtient une précision relativement faible car le nombre de faux positifs est élevé ce qui indique que les pigments prédis par le modèle correspondent à la vérité terrain dans seulement 80% des cas. Cependant le recall est très élevé car la quasi-totalité des taches présentes dans la vérité terrain sont trouvées par le modèle.

La faible valeur de précision est cependant à nuancer. En effet, les archéologues nous avaient fait part du fait qu'il préfèrent obtenir des faux positifs que des faux négatifs car ils peuvent facilement vérifier si les taches "suspectes" prédites par le modèle sont bien des pigments ou non. Également un certain nombre de ces faux positifs sont également dues à des erreurs dans la vérité terrain, comme le montre l'exemple ci-dessous où on distingue effectivement 3 taches bleues dans l'image alors que la vérité terrain n'en indique qu'une seule :

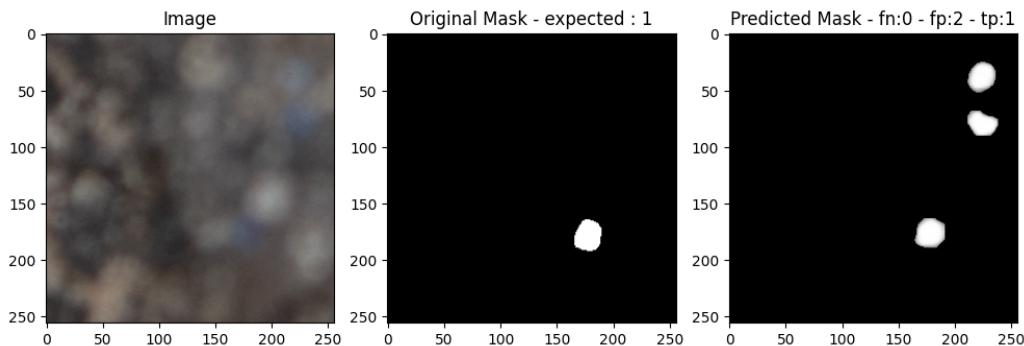


FIGURE 16 – Exemple de faux positifs lié à une erreur dans la vérité terrain

5.3 Exemples

Enfin voici quelques exemples de prédictions réalisées par le modèle afin de donner une idée des performances de ce dernier :

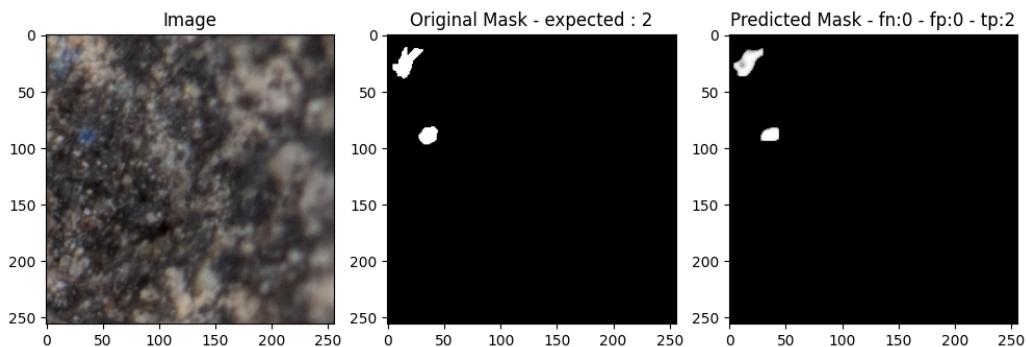


FIGURE 17 – Exemple de prédiction 1

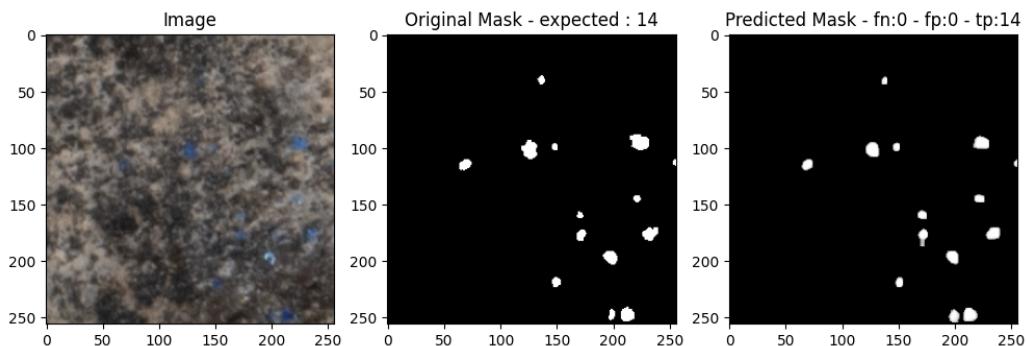


FIGURE 18 – Exemple de prédiction 2

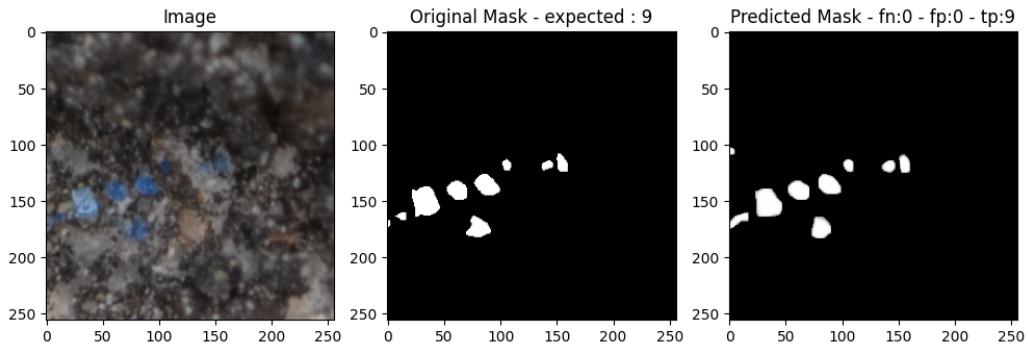


FIGURE 19 – Exemple de prédiction 3

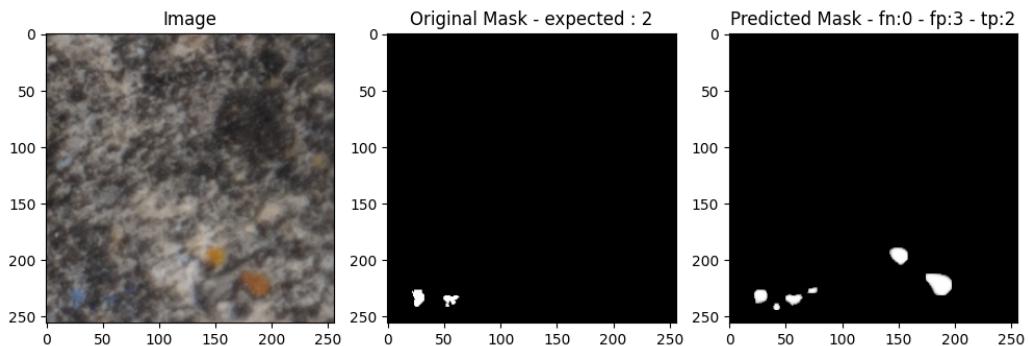


FIGURE 20 – Exemple de prédiction 4

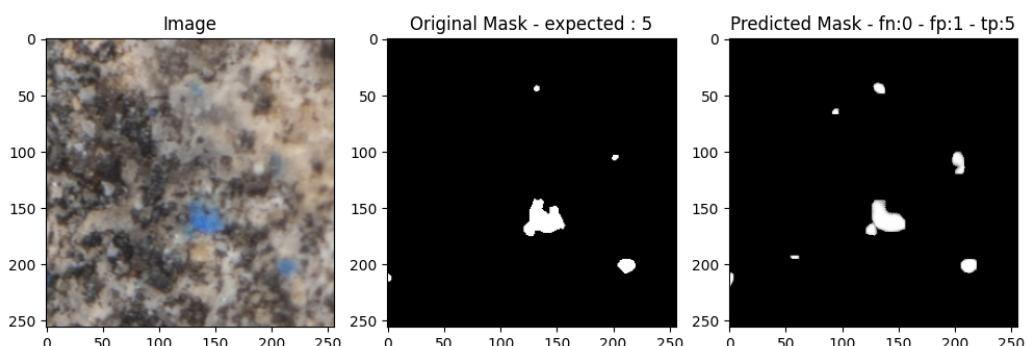


FIGURE 21 – Exemple de de prédiction 5

6 Améliorations et extensions

6.1 Étiqueteur

Comme dit précédemment l'Étiqueteur est un logiciel minimaliste, qui intègre les fonctionnalités nécessaire à la création de la vérité terrain. Nous l'avons cependant imaginé pour pouvoir être amélioré à l'avenir si notre client voulait le faire évoluer.

Premièrement sur la page Labeler 4.2.2, la barre latérale n'a pas de taille prédéfinie et s'adapte à son contenu. On peut donc imaginer à l'avenir, d'autres outils pour aider l'expert dans son étiquetage.

Deuxièmement et toujours sur la page Labeler, la sélection des pixels pourrait se faire à l'aide de "region-growing", où à minima en pouvant changer la taille du "pinceau". Au lieu de sélectionner les pixels un à un, sélectionner un carré avec une taille variable. La grille permettra une implémentation plus simple de cette amélioration car chaque élément qui la compose possède une propriété contenant ses coordonnées.

La troisième amélioration possible que nous envisageons est le fait de pouvoir directement zoomer dans la grande image. Même si elle occupe une grande partie de la page et que le filtre aide à repérer les pigments, nous avons remarqué qu'un comportement typique (des experts ou personnes tiers) était de vouloir directement agrandir l'image. Cette amélioration permettrait d'améliorer l'expérience utilisateur et de rendre l'étiqueteur plus intuitif.

La dernière amélioration est celle de pouvoir supprimer l'étiquetage d'un pigment ou de recommencer complètement l'étiquetage d'une image.

6.2 Modèle d'apprentissage profond

Le modèle d'apprentissage que nous avons obtenu est satisfaisant mais pourrait bien sûr être amélioré.

Il serait intéressant de l'entraîner pendant un plus grand nombre d'epochs car on constate que lors de l'entraînement de notre modèle la fonction objectif ne semblait pas avoir encore convergé.

Aussi il serait toujours intéressant d'étoffer le jeu de données en étiquetant de nouvelles photos ou peut-être d'utiliser de l'augmentation de données afin d'entraîner le modèle sur un nombre d'images plus important.

Enfin une dernière piste qui pourrait-être intéressante serait de modifier la structure même du réseau Mask R-CNN pour essayer de la rendre plus

performante sur notre projet mais cela demande de solides compétences et connaissances en apprentissage automatique.

7 Annexes

7.1 Schéma des modèles d'apprentissage

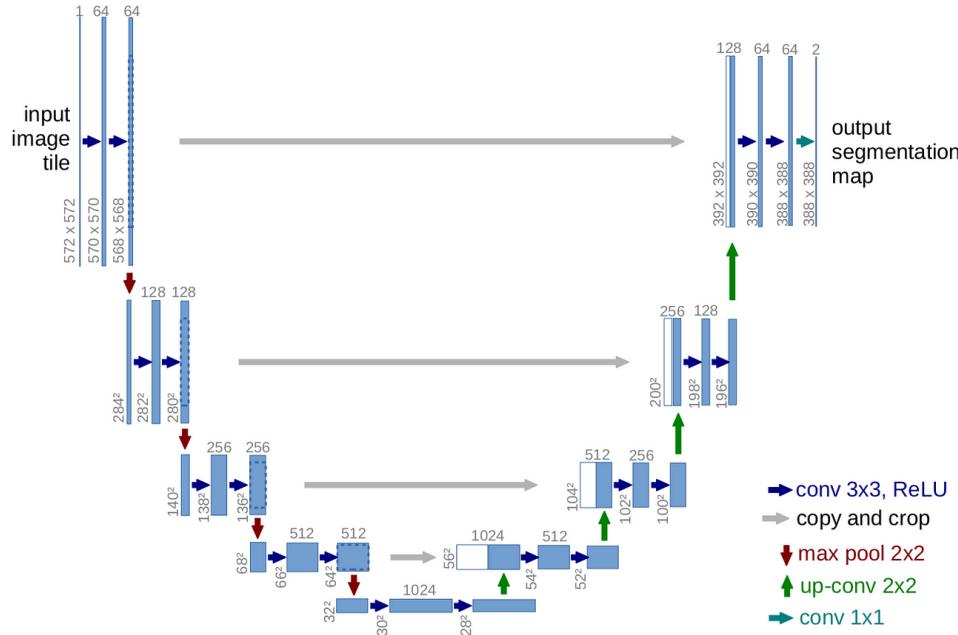


FIGURE 22 – Architecture du réseaux U-net. Source : U-Net : Convolutional Networks for Biomedical Image Segmentation [2]

7.2 Captures d'écran

7.2.1 Rapport d'entraînement

Training Report 2023-04-21 11:13:57

Parameters

Dataset : /autofs/unitytravail/travail/lDupouey/rendu-pdp/dataset

Epochs : 12

Batch Size : 5

Image Size : 256

Size of training dataset : 268

Size of test dataset : 67

Training duration: 30.664494466781616 min

Training summary

Loss evolution :

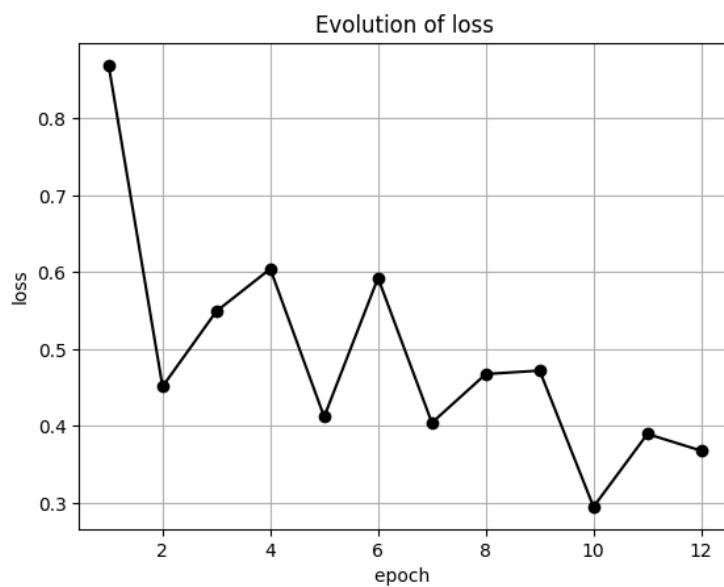
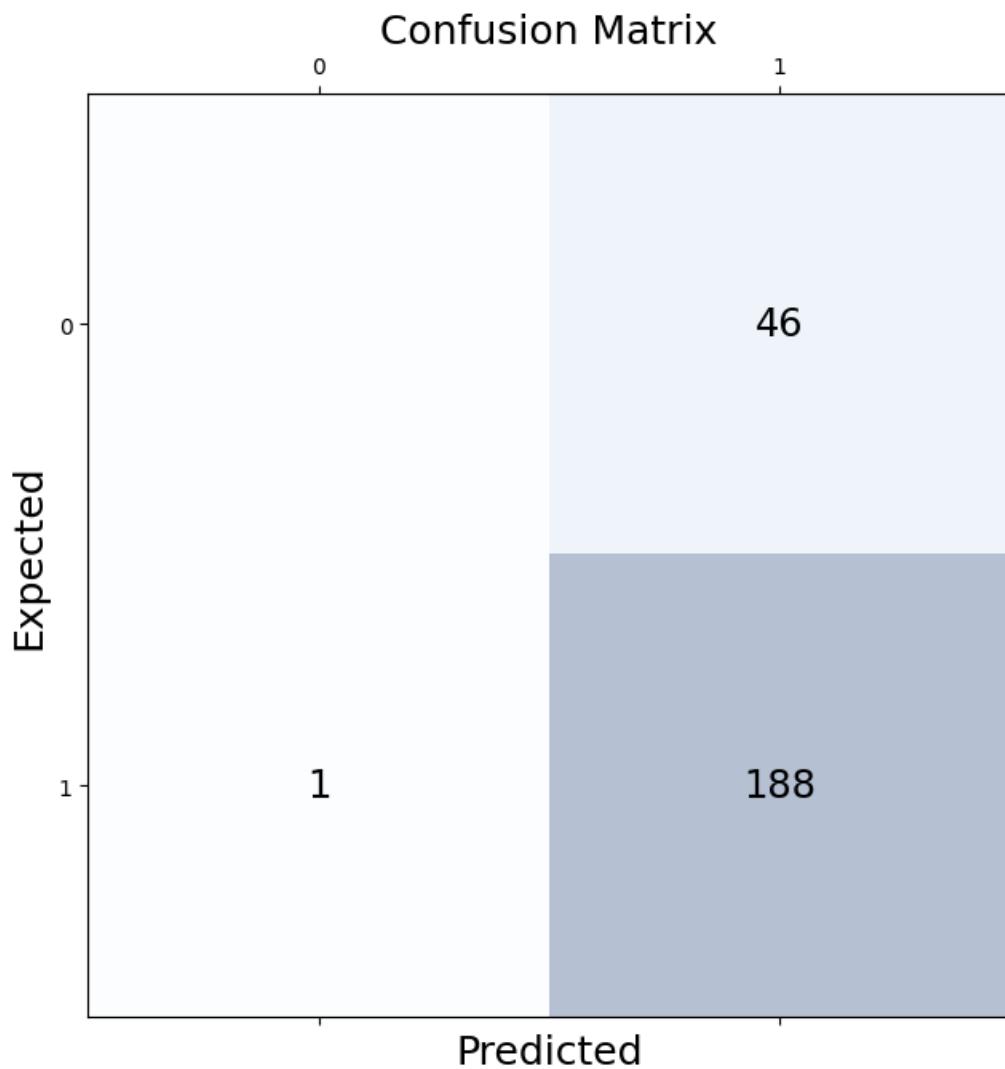


FIGURE 23 – Capture d'écran du rapport HTML d'entraînement : partie 1

Confusion Matrix



Precision and Recall

Precision : 0.8034188034188035

Recall : 0.9947089947089947

FIGURE 24 – Capture d'écran du rapport HTML d'entraînement : partie 2

Examples

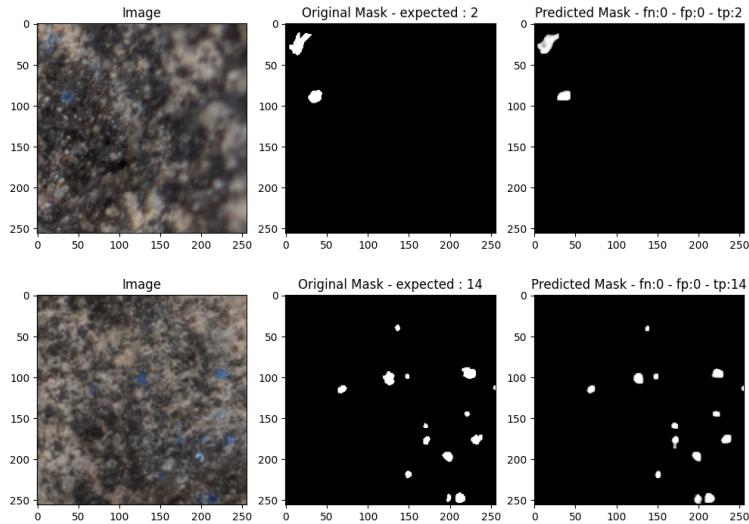


FIGURE 25 – Capture d’écran du rapport HTML d’entraînement : partie 3

7.3 Manuel utilisateur



Detection automatique de pigments

Manuel Utilisateur

Version 1.0

Barrey Victor, Dupouey Léo, Moze Jonathan
April 23, 2023

Table des matières

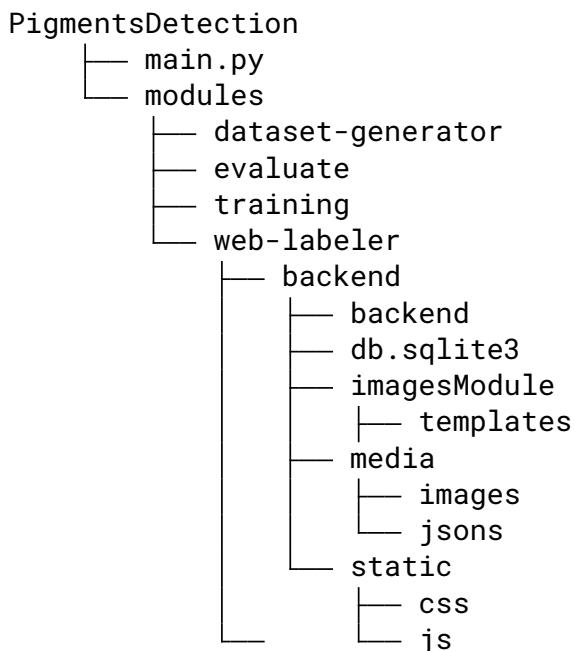
1	Introduction	2
2	Architecture de l'application	2
3	Guide d'installation	3
4	Utilisation de l'application	4
4.1	Etiqueteur web	4
4.2	Générateur du dataset	5
4.3	Entrainement du modèle d'apprentissage automatique	6
4.4	Evaluation d'un ensemble d'images	7

1 Introduction

Ce document est le manuel utilisateur de l'application de détection de pigments par apprentissage automatique. Il est composé de trois modules. Le premier est une application web permettant d'étiqueter les pigments d'une image, le second est un module permettant de générer le set d'apprentissage depuis les fichiers jsons d'étiquetage des images. Enfin, le dernier module est le module d'apprentissage automatique par réseau de neurones.

2 Architecture de l'application

Voici l'architecture générale de l'application. Nous y ferons référence plus tard dans le manuel.



3 Guide d'installation

Cette application est développée en **Python** sous la version 3.10.11, Il est donc recommandé d'utiliser cette version pour le garantir son bon fonctionnement. Python 3.10 est disponible à cette adresse :

<https://www.python.org/downloads/release/python-31011/>

Une fois python installé, nous allons créer un environnement virtuel python afin d'éviter les conflits avec les librairies que nous allons installer plus tard. Pour créer un environnement, utilisez la commande :

```
> python -m venv <environment name>
```

Ensuite, pour activer l'environnement virtuel, executez :

```
> source env/bin/activate
```

(sous windows executez) :

```
> env/Scripts/activate
```

À ce stade, votre terminal (selon celui que vous utilisez) ajoutera probablement le nom de votre environnement au début de chaque ligne de votre terminal (le nom donné à la place de **<environment name>**).

Nous allons maintenant installer les librairies dont nous avons besoin. Pour cela placez vous dans le dossier **PigmentsDetection**. Une fois cela fait, exécutez les commandes :

```
> pip install -r requirementsTorch.txt  
> pip install -r requirements.txt
```

ces commandes installent toutes les librairies contenues dans les deux fichiers texte. Il est très important de les effectuer dans le bon ordre pour garantir une bonne installation de toutes les librairies.

Nous sommes maintenant en capacité d'exécuter notre application.

4 Utilisation de l'application

L'application fonctionne en ligne de commande, il suffit simplement de se placer dans le dossier racine **PigmentsDetection**.

4.1 Etiqueteur web

Avant de lancer l'application web, si vous souhaitez que le serveur soit accessible sur un réseau local, il faudra au préalable récupérer l'adresse IPV4 locale de la machine où le logiciel est installé. Pour cela, sous linux tapez la commande **ifconfig** et l'adresse se trouve à la balise **inet**. Sous windows, tapez **ipconfig** dans un invite de commande et l'adresse se trouve à la balise **Adresse IPv4**.

Une fois l'adresse IPV4 récupérée, nous allons devoir la placer dans la balise **ALLOWED_HOSTS** du fichier **settings.py** dont le chemin relatif est :

```
> /modules/web-labeler/backend/backend/settings.py
```

```
ALLOWED_HOSTS = ['localhost', '192.168.1.151']
```

Figure 1: Exemple de la balise remplie avec une adresse IPV4

le serveur est maintenant prêt à être démarré. depuis le dossier racine, executez la commande suivante :

```
> python main.py runserver
```

Pour accéder au serveur depuis la machine où il est lancé, allez sur un navigateur web et entrez l'url "<http://localhost:8000>". Si vous voulez y accéder depuis une autre machine en réseau local, entrez l'url "<http://<ip>:8000>" avec <ip> qui est l'adresse IPV4 que nous avons récupéré précédemment.

L'étiquetage peut ensuite être effectué. Les fichiers jsons créés sont stockés dans le dossier **jsons** dont le chemin relatif est :

```
> /modules/web-labeler/backend/media/jsons
```

4.2 Générateur du dataset

Une fois que les jsons sont récupérés, il faut maintenant générer le set d'entraînement.

tapez la commande :

```
> python main.py generate-dataset
```

avec les arguments :

-i INPUT_DIR

Path to the directory containing
'images\' and 'jsons\' subfolders.

-o OUTPUT_DIR

Path to the directory where the new
dataset will be written.

(OPTIONAL) -s SUB_IMAGE_SIZE

Size of the images to be created
for the dataset (Default 256).

(OPTIONAL) -pp POSITIVE_PERCENTAGE

Percentage of positive images to include
in the generated dataset (Default 100%)

(OPTIONAL) -b

If set, mask will be binary (black/white),
else each instance instance will be
encoded with a different shade of grey.

(OPTIONAL) -h

show the help message.

Une fois terminé, les patchs se trouvent dans le dossier **batch** dont le chemin relatif est :

```
> OUTPUT_DIR/batch
```

et les masques associés dans le dossier **mask** dont le chemin relatif est :

```
> OUTPUT_DIR/mask
```

Ce que contiennent ces deux dossiers forment le dataset d'entraînement dont nous aurons besoin pour entraîner ensuite le modèle.

4.3 Entrainement du modèle d'apprentissage automatique

pour lancer l'entraînement, tapez la commande :

```
> python main.py train
```

avec les arguments :

-d DATASET_DIR

Path to the dataset to use for training.

-o OUTPUT_DIR

Path to the directory where the model
and report will be written.

-e EPOCHS

Number of epochs the model will be
trained for.

-b BATCH_SIZE

Number of epochs the model will be
trained for.

(OPTIONAL) -h show this help message.

Une fois l'entraînement terminé, le dossier de sortie contiendra des rapports sur l'entraînement effectué et un fichier **Model.pt** qui contient le réseau qui vient d'être entraîné.

4.4 Evaluation d'un ensemble d'images

maintenant que le modèle est entrainé, il est temps d'évaluer de nouvelles images.

pour cela placez vos nouvelles images dans un nouveau dossier à la racine de l'application.

Ensuite, tapez la commande :

```
> python main.py eval
```

avec les arguments :

-i INPUT_DIR

Path to the directory containing images
to evaluate.

-o OUTPUT_DIR

Path to the directory where the predicted
masks will be written.

-m MODEL

Pytorch model (.pt file) to use for
evaluation.

(OPTIONAL) -s SUB_IMAGE_SIZE

Size of the cropped images to pass
to the model (Default 256).

(OPTIONAL) -h show this help message.

Les masques créés sont les prédictions d'emplacement des pigments bleus
sur l'image associée.

Références

- [1] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Keh-tarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning : A survey. *CoRR*, 2020.
- [2] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net : Convolutional networks for biomedical image segmentation. *CoRR*, 2015.
- [3] Guillaume Chhor and Cristian Bartolome Aramburu. Satellite image segmentation for building detection using u-net, 2017.
- [4] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. Mask R-CNN. *CoRR*, 2017.
- [5] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Doll'a r, and C. Lawrence Zitnick. Microsoft COCO : common objects in context. *CoRR*, 2014.
- [6] William S Vincent. *Django for Beginners : Build websites with Python and Django*. WelcomeToCode, 2021.
- [7] Joao Bioco and Álvaro Rocha. Web application for management of scientific conferences. In *New Knowledge in Information Systems and Technologies*, pages 765–774, 04 2019.
- [8] Shruti Jadon. A survey of loss functions for semantic segmentation. In *2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*. IEEE, oct 2020.