Rapport test technique Stage SAVEP

Rédigé par Yassine AZABI à l'attention d'Amira Mimouna

I- Explication de l'algorithme et des choix techniques :

Le projet est composé de 3 fichiers python : main.py, helper.py et Fine-tuning.py.

Étape 1 : Détection des contours : Avant d'utiliser l'algorithme de détection de contour **cv2.Canny**, on applique un « **flou Gaussien**» à l'image chargé en **niveaux de gris**. Le flou est utilisé pour éviter de détecter trop de **contour du au « bruit »** (erreur en tout genre sur la photo, type qualité, superposition etc.).

Étape 2 : Segmentation sémantique avec U-Net :

- Choix du modèle : Le choix d'**U-net** s'est fait empiriquement sur les performances du modèle (pré-entrainé sur Imagenet) testé sur le dataset Floodnet. (Pas la méthode de sélection idéale)
- Fine-tuning : Le modèle pré-entrainé sur le data-set Imagenet a été chargé depuis PyTorch, puis **fine-tuné** sur notre jeu de donnée Floodnet. (Explication plus détaillé en annexe)
- Application de segment_image() : Fonction de helper.py qui charge et formate l'image pour ensuite la faire passer au model U-net ajusté, puis applique une **sigmoïde** (**problème binaire** : on a 2 classe, praticable et non-praticable.) aux logits.

Étape 3 : Détection des obstacles avec YOLO : On utilise un modèle de détection d'obstacle recommandé dans le sujet pré-entrainé. Pour des résultats corrects, il aurait fallu entrainé et/ou fine-tuné sur FloodNet ou a minima sur un dataset comparable (ex : image satellite avec label.)

Étape 4 : Fusion des résultats (Contours + Segmentation) : On veut maintenant utiliser les 2 travaux précédent pour crée une map « binarisé », chaque contour sera considéré comme une zone non praticable. Cet ajout a pour objectif d'améliorer les potentiels « obstacles » non détecté par le modèle de segmentation.

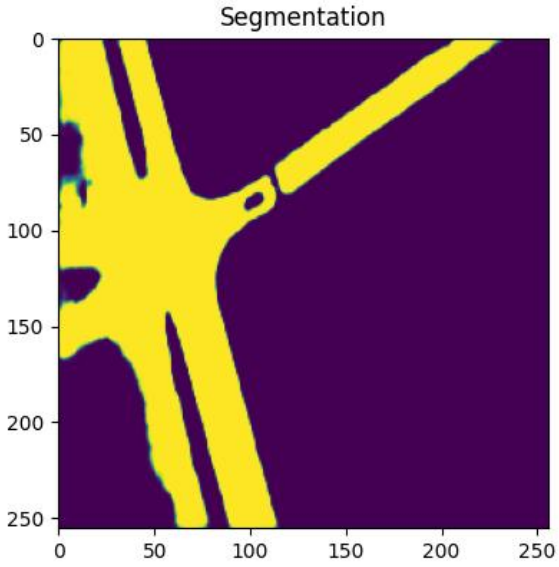
Étape 5 : Détection des points de départ et d'arrivée : Pour mes tests, un choix arbitraire et fixe du point de départ et d'arriver étaient un problème. J'ai alors décidé d'ajouter un choix automatique de points éloignée pour le start et le goal (fin).

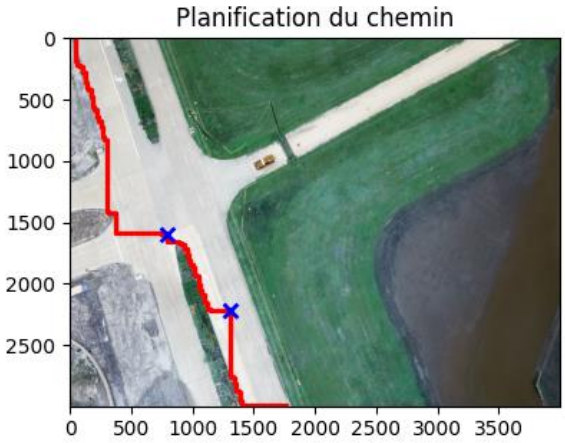
Étape 6 : Planification de chemin avec A* : On choisit d'abord de crée (ici aléatoirement) des checkpoints (**points de passage prioritaires**) parmi les positions « praticables » de la map.

On fait attention à choisir le **chemin le plus optimisé**, pour cela on compare pour chaque **combinaison d'ordre** possible la concaténation des chemins crée par A* entre chaque checkpoint. (On veut éviter de faire des allers-retours)

(Une fonction pour changer de manière automatique le chemin à l'apparition d'un nouvel obstacle est codé dans helper.py mais pas utilisé dans le main.py .)

II- Visualisation finale des résultats :





Pour le tracé du chemin il a fallu faire attention encore une fois à remettre à la bonne échelle les positions (x,y) du chemin. On observe en bleu les **checkpoints**, on observe l'ajouts des **diagonales** à A*, start et goal sont les points praticables les plus éloignées.

Cependant, on peut observer que notre modèle de segmentation ne considère **PAS** les terrains verts comme praticables. Une raison probable serait que ces terrains verts ne sont pas labelisé comme 0 background et donc considéré comme non praticable par U-Net.

III – Axe d'amélioration :

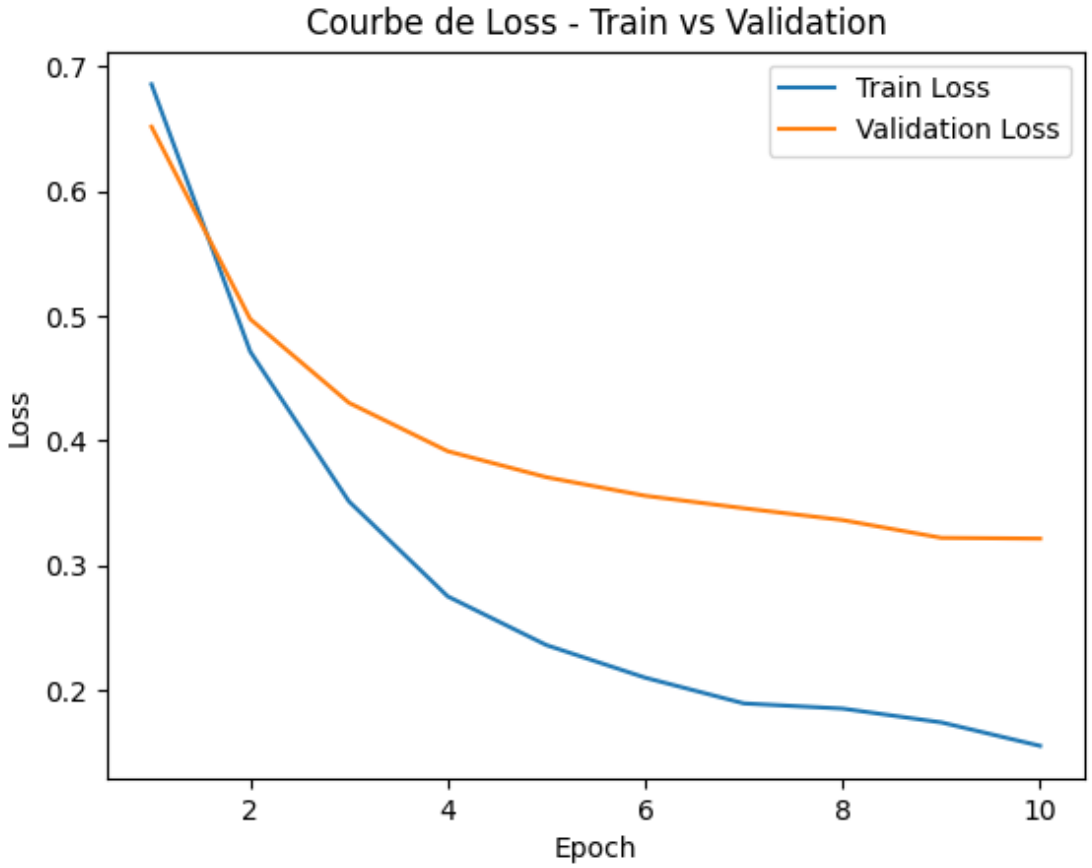
D'abord, comme précisé auparavant, **entrainé le modèle YOLO** de détection d'obstacle ou trouvé un modèle adapté aux image satellite, car pour l'instant cette option est totalement obsolète dans la version actuelle de mon code.

Peut-être comparé et choisir le chemin le plus court donné par **plusieurs algorithmes de planification de chemin** : A*, Dijkstra etc.

Evaluer la pertinence de la recherche de contour pour notre objectif. Es ce que la segmentation à elle seule suffirait ?

Alléger le code et les calculs, mon code pourrait surement être plus efficace et moins long.

ANNEXE : Fine-tuning :



On observe un **plateau** se former sur la courbe de la Validation Loss à partir de la 9eme epoch (clairement) et petit à petit depuis l'epoch 5 à peu près.

Pour pallier à ce problème l'utilisation d'un scheduler pour **ajuster le taux d'apprentissage** (LR) de manière automatique pendant la boucle d'entrainement aurait été judicieux. (Même si Adam adapte dans un autre sens le learning rate aussi.)

La Loss choisit est la **DiceLoss** qui supporte les cas de multi-classe et les **cas binaires**.

L'optimiseur utilisé est l'optimiseur **Adam**, on a un dataset relativement petit pour le fine-tuning, Adam s'adapte bien aux petit datasets. De plus U-Net est un CNN avec beaucoups de couches , Adam empêche la disparition du gradient (vanishing gradients).

Merci d'avoir pris le temps le lire mon rapport.