UNIVERSITÉ TOULOUSE III PAUL SABATIER

A Light-Weight Neural Network for Fast and Interactive Edge Detection in 3D Point Clouds

HAJJANE IMAD EDDINE

MENTION INTELLIGENCE ARTIFICIELLE
MASTER INFORMATIQUE

Tuteur(s) :
LOIC BARTHE :
loic.barthe@irit.fr
NICOLAS MELLADO :
nicolas.mellado@irit.fr

Résumé

L'objectif de ce rapport est d'étudier et d'améliorer les différentes architectures de réseaux neuronaux pour la détection des bords dans les nuages de points. Trois approches clés ont été explorées : PCEDNet, un modèle basé sur le Transformeur avec Multi-Head Attention, et un modèle Bidirectionnel LSTM. Chaque architecture a été soigneusement analysée, implémentée et évaluée avec un ensemble de métriques variées telles que la Précision, le Rappel, le MCC, le score F1, l'Exactitude et l'Indice de Jaccard. L'accent a été mis sur la compréhension de leur comportement, leurs forces et leurs limites, ainsi que sur l'identification des domaines où des améliorations pourraient être apportées.

L'étude a également porté sur des éléments clés tels que la taille des lots d'entraînement, les fonctions d'activation, les relations entre les vecteurs d'échelle et d'autres paramètres importants. L'objectif ultime était d'optimiser ces architectures pour obtenir des performances supérieures dans la détection des bords dans les nuages de points.

En somme, cette étude vise à non seulement explorer et comprendre en profondeur ces architectures, mais aussi à apporter des améliorations tangibles pour renforcer leur efficacité et leur applicabilité dans le domaine de la détection des bords.

Table des matières

1 Introduction

La détection précise des bords dans les nuages de points est essentielle pour des applications variées. Ce rapport se penche sur l'amélioration de cette détection en explorant différentes architectures de réseaux neuronaux. Trois approches clés - PCEDNet, modèle basé sur Transformeur avec Multi-Head Attention, et Bidirectionnel LSTM - sont étudiées et évaluées en profondeur. L'objectif est de comprendre leur comportement, d'identifier les forces et faiblesses, et de proposer des améliorations. Ce rapport offre un aperçu des méthodes de détection des bords dans les nuages de points.

2 Pourquoi ce TER ?

En tant que personne ayant déjà des compétences en modélisation et impression 3D, j'ai souvent été confronté à la nécessité de numériser des objets en 3D et d'extraire les contours afin de construire des surfaces et des modèles. Ce projet de fin d'études représente une opportunité unique d'explorer l'utilisation de l'intelligence artificielle pour simplifier le processus de détection de contours.
Le domaine de la détection de contours dans les nuages de points est crucial pour la reconstruction et la modélisation précises d'objets 3D. La capacité d'automatiser ce processus à l'aide d'algorithmes et de modèles d'apprentissage automatique offre un grand potentiel pour accélérer et améliorer la qualité du travail dans le domaine de la modélisation 3D.
En appliquant des techniques d'apprentissage profond, telles que les réseaux neuronaux LSTM et les modèles basés sur l'attention, ce projet peut ouvrir de nouvelles perspectives pour la détection de contours dans les nuages de points. Cette approche pourrait non seulement simplifier le processus, mais aussi améliorer la précision et la cohérence des résultats, ce qui est essentiel pour la création de modèles 3D précis et réalistes.

3 Outils utilisés

Dans le contexte de ce projet, la préférence s'est principalement portée vers l'utilisation du framework TensorFlow en sa version 2.10. Cette itération du framework présente la remarquable opportunité de tirer parti des performances de traitement offertes par les GPU, tout en étant harmonieusement compatible avec Python 3.9 sous l'environnement Windows.

Pour optimiser la gestion et la transformation des données, on a également fait recours à la bibliothèque NumPy. Cette démarche nous a autorisé à accomplir les opérations de conversion et de lecture de données avec aisance et efficacité.

En matière de visualisation des données et des résultats, on a incorporé les bibliothèques Matplotlib et Pandas dans cette approche. Grâce à l'usage de Matplotlib, on a pu concevoir des graphiques et des visualisations approfondies qui enrichissent la compréhension. En parallèle, Pandas a dévoilé des fonctionnalités avancées pour la manipulation et l'analyse approfondie des données.

De plus, pour optimiser les performances de calcul et débloquer le Global Interpreter Lock **(GIL)**, on a utilisé la bibliothèque `Numba`. Cela a permis d'accélérer les calculs, notamment pour l'extraction et le calcul de la carte gaussienne discrète. On a également exploité `ThreadPoolExecutor` pour exécuter en parallèle les tâches de calcul, ce qui a contribué à améliorer l'efficacité du traitement.

En somme, l'utilisation de ces outils et bibliothèques a facilité la mise en œuvre, l'analyse et la visualisation des données, ainsi que l'accélération des calculs, contribuant ainsi à la réalisation efficace et précise des tâches liées au projet.

4 Prétraitement des données

4.1 Nettoyage des données

Dans les ensembles de données d'entraînement et de validation, on a éliminé les points qui contenaient des valeurs NaN ainsi que les étiquettes associées aux données SSM. Ces valeurs avaient un impact sur l'efficacité des modèles utilisant les données SSM au cours de la phase d'entraînement. Cependant, pour les ensembles de test, les modèles prédisent `NaN` pour les entrées ayant des valeurs `NaN`, les interprétant comme des zéros.

4.2 Lecture des données

En employant Numpy, on a transformé les données (ssm, ply, lb) en fichiers .npy (format numpy), tout en désactivant l'utilisation de pickle pour des raisons de

sécurité et de transférabilité, afin garantir l'efficacité, la sécurité et la portabilité des opérations de sauvegarde et de chargement de données Numpy.

4.3 Analyse statistique des données

Les ensembles de données ABC sont classifiés en deux catégories : la classe 0 représente les éléments sans bordure (non edge), tandis que la classe 1 représente les éléments avec bordure (edge).

- Pour l'ensemble d'entraînement : Il se compose de 136 fichiers au format ply, chacun accompagné avec sont fichier (.ssm) et des étiquettes correspondantes (.lb). Cet ensemble contient au total 3 819 745 points étiquetés comme non edge, ce qui représente 99.4 % des données d'entraînement, ainsi que 22 303 points étiquetés comme edge. La prédominance des étiquettes non edge dans l'ensemble d'entraînement peut potentiellement influencer l'entraînement des modèles en favorisant la reconnaissance de cette classe.
- Pour l'ensemble de validation : Composé de 30 fichiers ply, chaque fichier étant accompagné avec son fichier (.ssm) et des étiquettes associées. Cet ensemble contient en tout 196 887 points étiquetés comme non edge, représentant 92.43 % des données de validation, ainsi que 16 105 points étiquetés comme edge.
- Pour l'ensemble de test : Il se compose de 58 fichiers ply, chacun avec son modèle de forme statistique et les étiquettes correspondantes. Cet ensemble comprend un total de 1 279 836 points étiquetés comme non edge, ce qui correspond à 88.76 % des données de test, et 161 956 points étiquetés comme edge.

La répartition inégale entre les classes non edge et edge dans les différents ensembles de données peut potentiellement avoir un impact sur les performances des modèles pendant l'entraînement et l'évaluation. Il est important de prendre en compte cette distribution lors de la conception et de l'analyse des modèles pour s'assurer qu'ils sont équilibrés et performants sur les deux classes.

5 Entrainement

5.1 Les générateurs

Afin de fournir les données nécessaires aux modèles, on a mis en place une classe héritante de `tensorflow.keras.Sequence`. Cette approche nous permet de créer un générateur de données qui assure un approvisionnement fluide en données lors des phases d'entraînement, de validation et d'évaluation des modèles. À chaque étape, le générateur est capable de lire les fichiers au format ply ou ssm et leurs étiquettes

correspondantes. De plus, il prend en charge l'extraction de lots (batches) de données, en veillant à ce que les données soient mélangées de manière appropriée pour garantir une diversité et une robustesse maximales lors de l'apprentissage. Cette approche permet de :

- Gestion de la mémoire : Les fichier sont chargées et prétraitées à la demande plutôt que d'être stockées en mémoire, ce qui s'avère crucial avec de larges ensembles de données qui ne tiennent pas en mémoire RAM.
- Parallélisation : La parallélisation est possible avec `tensorflow.keras.utils.Sequence`, ce qui accélère le processus d'entraînement en exploitant la puissance de calcul.
- Évitement des goulots d'étranglement : Utiliser un générateur basé sur Sequence évite de charger toutes les données en mémoire simultanément, réduisant ainsi les risques de blocage des ressources de stockage et de mémoire.
- Personnalisation et flexibilité : il est possible de personnaliser la façon dont les données sont chargées, prétraitées et fournies au modèle en définissant les méthodes appropriées dans classe dérivée de `tensorflow.keras.utils.Sequence`.
- Prévention des problèmes de mémoire : L'utilisation de Sequence est particulièrement bénéfique lorsque vous manipulez de grands ensembles de données, car elle contribue à éviter les problèmes de mémoire insuffisante ou de blocage du système.

5.2 La taille des batches, l'Optimisateur, callbacks, métriques et matérial

5.2.1 La taille des batches

Dans le cadre de la formation de ces modèles, on a pris la décision d'adopter des lots d'une taille de **512** ou **16384**, en fonction de la complexité du modèle en cours de développement. Cette stratégie a pour principal objectif d'accélérer le processus d'entraînement en exploitant les avantages inhérents aux lots de grande dimension. L'utilisation de lots de grande taille présente plusieurs avantages concrets. Tout d'abord, elle permet de réaliser des calculs parallèles de manière plus efficace et rapide, en capitalisant pleinement sur les capacités de calcul parallèle du GPU. Cette accélération des calculs se traduit par une amélioration significative des performances du processus d'apprentissage. Une caractéristique importante de cette approche est la stabilité du gradient pendant l'optimisation. En utilisant des lots de grande taille, le calcul du gradient est effectué sur un plus grand nombre d'échantillons, ce qui peut réduire la variabilité du gradient. Cette stabilité contribue à une convergence plus fluide et rapide du modèle. En outre, l'adoption de lots de grande taille peut jouer un rôle similaire à celui des techniques de régularisation, telles que le `Dropout`

et la `Batch-Normalisation`. Elle aide à contrôler le risque de sur-apprentissage en encourageant une généralisation plus robuste du modèle.

5.2.2 L'optimisateur et Callbacks

En ce qui concerne l'optimisation, on a choisi d'utiliser l'optimiseur Adam avec un taux d'apprentissage initial de 0.001. Cependant, une modification est apportée après un certain nombre d'époques. À ce stade, le taux d'apprentissage est multiplié par le facteur exponentiel de décroissance, soit $\exp(-4.1)$, en utilisant l'outil `tensorflow.keras.callbacks.ReduceLROnPlateau`.

```
hist = model.fit(datagen_train, validation_data=
    datagen_validation, epochs=EPOCH,callbacks=[tf.keras.
    callbacks.ReduceLROnPlateau(factor=tf.math.exp(-4.1),
    patience=int(EPOCH*2/10), min_lr=0.0000001)])
```

En somme, l'utilisation de lots de grande taille, combinée à l'optimiseur Adam avec ajustement du taux d'apprentissage, représente une approche équilibrée visant à accélérer le processus d'entraînement, favoriser la convergence du modèle, et contrôler les risques de sur-apprentissage.

5.2.3 Les métriques

La forte majorité des étiquettes attribuées à la classe non edge au sein de l'ensemble d'entraînement peut avoir un impact potentiel sur l'apprentissage des modèles, en inclinant leur apprentissage en faveur de la reconnaissance de cette classe prédominante. Pour aborder cette situation, on a opté pour l'utilisation de trois métriques distinctes : "exactitude"(accuracy), "précision" et "rappel" . Ces métriques me permettent d'évaluer et de surveiller si le modèle présente des biais vers l'une des classes pendant la phase d'entraînement.

- La métrique **"exactitude" (accuracy)** mesure le pourcentage global de prédictions correctes du modèle sur l'ensemble de données. Elle fournit une vue d'ensemble de la performance du modèle, mais peut être influencée par le déséquilibre des classes, comme c'est le cas ici.
- La **"précision" (precision)** est une métrique qui évalue le nombre de prédictions positives correctes parmi toutes les prédictions positives effectuées par le modèle. Elle met en lumière la proportion de prédictions positives qui sont réellement correctes. Une faible précision pourrait indiquer une tendance du modèle à surestimer une classe.

- Le **"rappel" (recall)** mesure le nombre de prédictions positives correctes parmi tous les exemples réels qui appartiennent à la classe positive. Cette métriques permet de détecter si le modèle manque des exemples d'une classe particulière.

En utilisant ces trois métriques, on peut obtenir un aperçu plus complet de la performance du modèle, tout en surveillant tout biais potentiel qui pourrait se développer pendant l'entraînement. Cela nous aide à identifier si le modèle favorise l'une des classes et à prendre des mesures correctives si nécessaire pour garantir un apprentissage équilibré et une généralisation précise.

5.3 SELU vs RELU

Bien que la fonction ReLU **(Rectified Linear Unit)** soit reconnue pour sa simplicité, son efficacité de calcul et sa capacité à engendrer une sparsité qui réduit la redondance des activations, on a fait le choix d'utiliser la fonction SELU **(Scaled Exponential Linear Unit)** [3] pour plusieurs raisons fondamentales.
En effet, la fonction SELU offre des avantages spécifiques malgré la popularité de ReLU :

- **Auto-normalisation** : L'un des points forts de SELU réside dans sa propriété d'auto-normalisation. Cette caractéristique signifie que les moyennes et les variances des activations restent stables à travers les différentes couches du réseau. Cette stabilité favorise une convergence plus rapide et stable pendant la phase d'entraînement. Cette qualité est particulièrement bénéfique pour les réseaux profonds, où la convergence peut être difficile à obtenir.
- **Effet de régularisation intrinsèque** : Une autre raison majeure pour laquelle on a opté pour SELU réside dans son effet de régularisation intrinsèque. Les activations SELU possèdent naturellement une régularisation qui peut aider à contrôler le sur-apprentissage. Ce mécanisme interne peut réduire la nécessité d'utiliser des techniques de régularisation externes, simplifiant ainsi le processus de conception et d'ajustement du modèle.
- **Résolution du problème de "neurones morts"** : Un problème inhérent à la fonction ReLU est celui des "neurones morts", où les activations négatives sont bloquées et empêchent le flux de signal. SELU surmonte ce problème en autorisant une faible activation pour les valeurs négatives, ce qui maintient une dynamique plus fluide et contribue à un apprentissage plus efficace.

$$f(x) = \begin{cases} \lambda \cdot x & \text{si } x > 0 \\ \lambda \cdot \alpha \cdot (\exp(x) - 1) & \text{si} x \leq 0 \end{cases}$$

où $\lambda = 1.050700$ et $\alpha = 1.673263$
La fonction d'activation `SELU`

L'utilisation de la fonction SELU, cependant, demande une attention particulière lors de l'initialisation des poids. Il est recommandé d'employer la méthode d'initialisation de LeCun pour démarrer les poids de manière appropriée, compte tenu des spécificités de SELU et de son comportement dans le réseau.

5.4 Matériel

Pour la phase d'entraînement, on a tiré parti de la puissance du GPU **Nvidia RTX 3070 Ti**, qui dispose d'une mémoire vidéo (VRAM) de 8 Go. L'utilisation de GPU s'avére extrêmement avantageuse pour accélérer le processus d'entraînement de ces modèles.

La capacité de calcul parallèle et la mémoire vidéo généreuse du GPU ont permis d'exécuter les opérations d'entraînement de manière efficace et rapide. La VRAM de 8 Go a offert un espace suffisant pour stocker les poids du modèle, les données d'entrée, les activations intermédiaires et d'autres informations essentielles pour l'entraînement en profondeur.

6 Evaluation

Pour l'évaluation de ces modèles, on a soumis 52 fichiers au format ssm contenant un total de 279 836 points étiquetés comme "non edge" ainsi que 161 956 points étiquetés comme "edge". on a utilisé plusieurs métriques pour évaluer les performances des modèles dans cette tâche. Parmi les métriques utilisées, on retrouve la Précision, qui mesure la proportion de prédictions positives correctes par rapport au nombre total de prédictions positives. Le Rappel (Recall) évalue la proportion de prédictions positives correctes par rapport au nombre total d'exemples réels positifs. Le Coefficient de Corrélation de Matthews (MCC) offre une mesure de la qualité globale des prédictions, tandis que le score F1 combine à la fois la Précision et le Rappel.

En plus de ces mesures, on a également pris en compte l'Exactitude (Accuracy), qui mesure la proportion de prédictions correctes parmi toutes les prédictions effectuées. Enfin, l'Indice de Jaccard (IoU) a été utilisé pour évaluer la similarité entre les masques prédits et les masques réels des bords.

L'utilisation de ces métriques multiples permet d'obtenir une vue holistique des performances des modèles, en prenant en compte différents aspects tels que la précision, la sensibilité, la robustesse et la capacité à identifier les bords correctement. Cela permet une évaluation plus complète et détaillée de la performance des modèles dans la détection des bords dans les nuages de points.

7 Modèles SSM

7.1 PCEDNet

On a concrétisé la mise en place du réseau PCEDNet tel qu'il est décrit dans l'article [1]. Dans cette implémentation, les données d'entrée sont fournies au réseau sous la forme de seize vecteurs de 6 dimensions par point. Chaque vecteur correspond à une échelle spécifique, visant à capturer diverses échelles d'information. Les seize vecteurs d'échelle sont fusionnés deux par deux, générant ainsi huit vecteurs de 12 dimensions. Au cœur de ce processus, le premier vecteur d'échelle est concaténé avec le deuxième, le troisième avec le quatrième, et ainsi de suite. L'objectif fondamental de cette opération progressive est de réduire graduellement le nombre d'échelles à la moitié au fil d'itérations successives, aboutissant en fin de compte à un unique vecteur de 12 dimensions qui agit comme une représentation finale de caractéristiques consolidées. À chaque étape de cette concaténation, un perceptron est appliqué pour traiter ces vecteurs fusionnés. Le réseau PCEDNet est composé de deux couches neuronales : l'une dotée de 16 neurones et l'autre de 12 neurones, suivies d'une fonction d'activation `sigmoid`. Cette organisation en couches permet au réseau de saisir des caractéristiques complexes à partir des données d'entrée, tout en facilitant la détection précise des bords dans les nuages de points. En combinant astucieusement la fusion des échelles et l'utilisation successive des perceptrons, le réseau est en mesure d'acquérir une compréhension des caractéristiques à différentes échelles et d'exploiter cette connaissance pour améliorer sa performance dans la détection des bords.
Pour améliorer encore davantage le réseau, on a apporté des modifications. en choisissant d'utiliser des fonctions d'activation `SELU` pour la majorité des perceptrons, ce qui contribue à favoriser l'auto-normalisation et à atténuer le risque de surapprentissage. De plus, on a ajouté une couche de `Batch-Normalisation` entre les couches de neurones pour faciliter la stabilisation de l'apprentissage et renforcer les capacités de généralisation du réseau. En somme, cette implémentation du réseau PCEDNet, combinant la fusion des échelles et l'utilisation de fonctions d'activation `SELU` avec la `Batch-Normalisation`, vise à enrichir la représentation des caractéristiques et à optimiser la détection des bords dans les nuages de points en capturant efficacement des informations à différentes échelles.

Cette architecture se caractérise par sa légèreté, ne comptant que 1717 paramètres au total. Parmi ces paramètres, 1653 sont dédiés à l'apprentissage, tandis que les 64 restants sont fixes et ne subissent pas d'ajustements au cours de l'entraînement. La conception d'un réseau léger est cruciale pour plusieurs raisons. Tout d'abord, une architecture légère requiert moins de ressources en termes de mémoire et de calcul, ce qui permet une exécution plus rapide et plus efficiente, notamment sur des dispositifs avec des capacités limitées tels que les appareils mobiles ou les

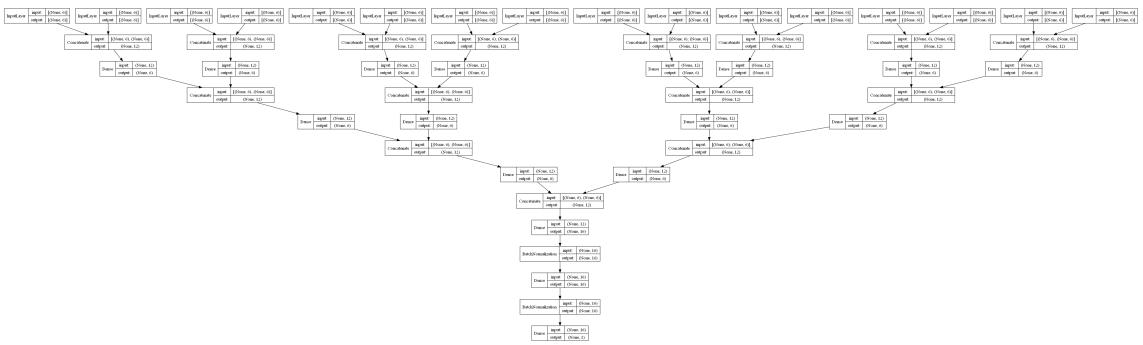


FIGURE 2 – La structure de l'architecture du modèle PCEDNet

systèmes embarqués. De plus, réduire le nombre de paramètres peut contribuer à atténuer le risque de sur-apprentissage, en permettant au modèle de généraliser mieux aux nouvelles données.Malgré sa légèreté, le réseau est conçu pour capturer efficacement les caractéristiques importantes et réaliser des performances de détection de bords compétitives. Cette optimisation du nombre de paramètres est le résultat d'une conception réfléchie et de l'utilisation stratégique des architectures, des couches et des techniques d'apprentissage. En fin de compte, cela permet d'obtenir une solution performante tout en étant efficace en termes de ressources.

Cette architecture a été implémenté dans le but d'être utilisée comme une référence pour évaluer d'autres architectures qu'on implémentera. Son rôle est d'établir un point de comparaison fiable pour mesurer les performances et les avantages des différentes conceptions de réseaux. Cela permet d'identifier les points forts et les faiblesses de chaque nouvelle architecture par rapport à une solution déjà éprouvée et évaluée.

7.2 LSTM bidirectionnel

Lors du premier essai, on a envisagé d'utiliser un réseau récurrent tel qu'un **RNN**. Cependant, cette architecture rencontrait le problème courant de la disparition du gradient, qui peut compromettre la capacité du modèle à apprendre des dépendances. Par conséquent, on a pris la décision de se tourner vers la mise en place d'un modèle LSTM **(Long Short-Term Memory)**, une architecture spécialement conçue pour résoudre le problème de la disparition du gradient et pour favoriser la mémorisation de séquences.
L'avantage principal d'un modèle LSTM réside dans sa capacité à conserver des informations à long terme et à gérer les dépendances temporelles. Ceci se révèle avantageux pour la détection de motifs complexes entre les vecteurs d'échelle, étant donné que ces vecteurs contiennent des informations significatives liées à différentes échelles ordonnées.

Cependant, la démarche ne s'est pas arrêtée là. on a décidé de mettre en œuvre un modèle LSTM Bidirectionnel, qui apporte des avantages substantiels. Contrairement à un modèle LSTM unidirectionnel qui n'analyse les séquences que dans une seule direction temporelle, le modèle Bidirectionnel LSTM exploite à la fois les deux sens, ce qui permet d'améliorer la capacité du modèle à saisir des structures et des motifs à des niveaux variés de granularité. L'implémentation finale repose sur un modèle Bidirectionnel LSTM ayant 16 unité suivi d'une couche de perceptrons comme sortie, dont la fonction d'activation est `sigmoid`.

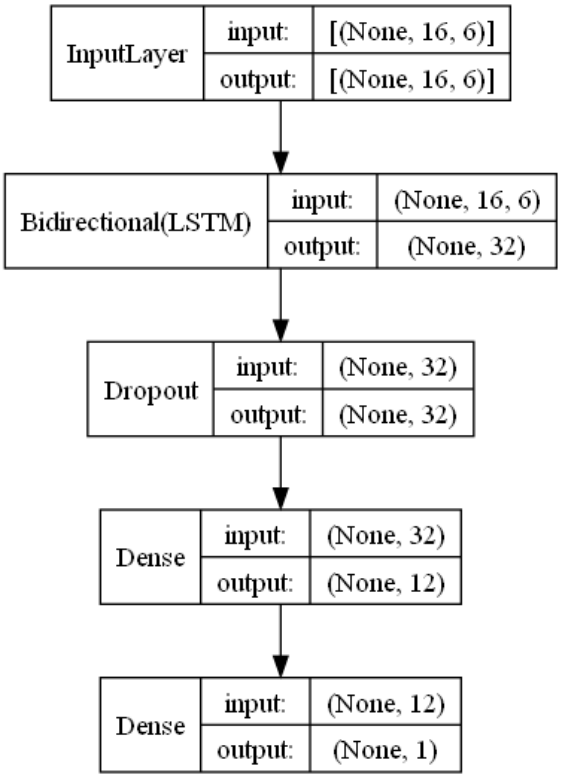


FIGURE 3 – La structure de l'architecture du modèle BiLSTM

On a entrepris une évaluation et une analyse en profondeur du modèle `Bidirectionnel LSTM` comportant 16 unités, en explorant différentes configurations de vecteurs d'échelle. Plus spécifiquement, on a évalué les performances du modèle en ajustant le nombre de vecteurs d'échelle, variant entre 16, 14, 12, 10, 8 et 6. Pour chaque configuration, on a soumis le modèle à un entraînement de dix époques, ce qui a permis d'obtenir une perspective globale sur son comportement et ses performances. L'objectif principal de cette démarche était de comprendre l'impact de la variation du nombre de vecteurs d'échelle sur les performances générales du modèle. En observant comment le modèle réagit à différents nombres d'échelle, on cherche à identifier toute tendance

significative ou tout effet notable sur ses capacités de détection des bords dans les nuages de points.

À travers cette évaluation approfondie, on a examiné les tests obtenus après les

	precision	recall	MCC	F1	accuracy	IoU	scales
0	0.943494	0.693653	0.790430	0.799510	0.962212	0.665986	16
0	0.936378	0.711240	0.797636	0.808427	0.963226	0.678454	14
0	0.941586	0.737192	0.816065	0.826946	0.966384	0.704952	12
0	0.933316	0.748044	0.817648	0.830472	0.965433	0.710092	10
0	0.928926	0.751183	0.818070	0.830653	0.966730	0.710356	8
0	0.928847	0.755311	0.820531	0.833139	0.967207	0.714000	6

FIGURE 4 – La structure de l'architecture du modèle BiLSTM

dix époques d'entraînement et l'evaluation pour chaque configuration de vecteurs d'échelle. Cette analyse nou a fourni une vision approfondie du comportement du modèle et de sa capacité à généraliser à différentes conditions. on a également mesuré la variance (table 1) entre les différentes configurations, ce qui a permis de mieux appréhender la stabilité et la cohérence des performances à travers les essais.

TABLE 1 – Variation des métriques selon les configurations

Métriques	Variation
Precision	0.000039
Recall	0.000619
MCC	0.000161
F1	0.000197
Accuracy	0.000004
IoU	0.000401

En examinant les données recueillies, notamment les mesures de précision, de rappel, de MCC, de F1, d'exactitude et d'indice Jaccard (IoU), on a pu conclure que l'impact du nombre de vecteurs d'échelle sur les performances du modèle Bidirectionnel LSTM est minime. Ces constatations renforcent la robustesse et l'efficacité du modèle dans la détection précise des bords dans les nuages de points, confirmant ainsi sa performance globale et sa capacité à généraliser à travers différentes configurations.

7.3 Multi-Head Attention (Tssm)

Cette approche adopte la partie encodeur de l'architecture du Transformeur, mettant en avant l'utilisation de la technique de Multi-Head Attention pour instaurer des connexions de grande importance entre les vecteurs d'échelle. L'objectif

sous-jacent de cette démarche est d'amplifier la communication et la compréhension réciproque entre ces vecteurs, ce qui aboutit à une exploitation plus optimale des informations. Contrairement au modèle `PCEDNet`, qui établit des liaisons entre les vecteurs d'échelle deux par deux, le `Multi-Head Attention` se consacre à l'apprentissage des liens entre tous les vecteurs, enrichissant ainsi l'interaction globale.
L'élément clé de cette approche est le Multi-Head Attention, qui opère comme un mécanisme sélectif. Il identifie et privilégie les relations les plus pertinentes et significatives entre les vecteurs d'échelle, tout en atténuant l'impact des relations moins marquées. Ce faisant, il permet d'aborder divers aspects des interactions entre les vecteurs.
Dans cette mise en œuvre, chaque tête d'attention agit comme un filtre qui saisit les relations spécifiques entre les vecteurs d'échelle, en fonction de leur degré d'importance. L'idée sous-jacente est d'accorder une attention accrue aux relations qui apportent une valeur significative à la tâche de détection des bords dans les nuages de points.

L'usage de cette architecture, inspirée par la partie encodeur du Transformeur, présente un potentiel d'amélioration en établissant des connexions plus sophistiquées entre les vecteurs d'échelle. Elle vise à renforcer les performances de détection des bords, en mettant en avant les relations pertinentes et en atténuant celles de moindre importance.

Lors de la configuration d'un encodeur, il est nécessaire de définir plusieurs paramètres cruciaux, notamment le nombre de têtes d'attention h, les dimensions des requêtes d_v et des clés d_k, le nombre d'encodeurs empilés n et la dimension du modèle d_{model}. Dans l'approche, on a porté une attention particulière aux paramètres h, d_v et d_k, qui jouent un rôle fondamental dans la détection des relations spécifiques entre les vecteurs d'échelle, en fonction de leur importance respective.

Le nombre de têtes d'attention h détermine la diversité et la richesse des perspectives prises en compte lors de la construction des liens entre les vecteurs d'échelle. Plus h est élevé, plus le modèle est en mesure de capturer des interactions complexes et variées entre les éléments du jeu de données. Cela s'avère particulièrement pertinent pour la détection précise des relations spécifiques et la mise en évidence de leurs degrés d'importance.
Quant aux dimensions des requêtes d_v et des clés d_k, elles influent sur la capacité du modèle à comprendre et à représenter les relations entre les vecteurs d'échelle. Ces dimensions agissent comme des canaux pour exprimer l'importance respective des relations dans l'espace des vecteurs d'échelle. En choisissant soigneusement les valeurs de d_v et d_k, on peut orienter le modèle vers une meilleure discrimination des liens essentiels, renforçant ainsi sa capacité à identifier les relations spécifiques qui contribuent à la détection des bords.

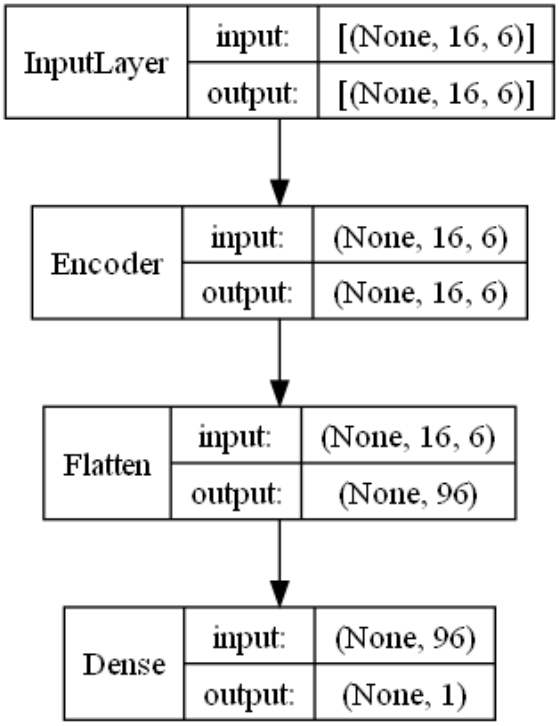


FIGURE 5 – La structure de l'architecture du modèle Tssm

L'approche de sélection des paramètres, spécialement h, d_v et d_k, à pour but d'améliorer la détection des relations cruciales entre les vecteurs d'échelle. En manipulant habilement ces paramètres, on cherche à accroître la sensibilité du modèle aux relations importantes, tout en renforçant sa capacité à identifier et à comprendre les relations spécifiques, ce qui joue un rôle essentiel dans la précision de la détection des bords dans les nuages de points.

Pendant la phase d'entraînement, lorsque les valeurs de d_v et d_k dans la technique de la `Multi-Head Attention` sont relativement faibles, une tendance à favoriser la classe non edge se manifeste. Ceci s'explique par la déséquilibre inhérent dans la distribution des classes non edge et edge, comme discuté dans la section Analyse statistique des données. En effet, cette section (4.3) met en lumière la prédominance des points classifiés comme non edge dans l'ensemble de données.

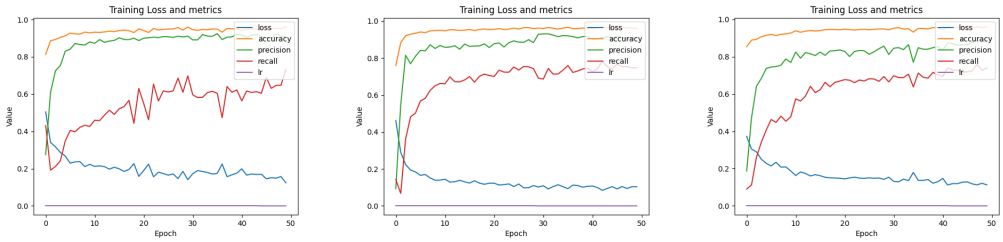
Après une évaluation approfondie, il est apparu que le modèle tend à prédire la classe non edge pour toutes les données de l'ensemble de test. Cette tendance peut s'expliquer par le fait que le modèle a été biaisé par la fréquence plus élevée des exemples non edge dans les données d'entraînement. Par conséquent, il manifeste des difficultés à généraliser correctement les motifs et caractéristiques des exemples edge, ce qui se traduit par une prédiction uniforme de la classe non edge dans l'ensemble de test.

Dans notre cas, nous avons utilisé un modèle avec les paramètres suivants :

h	d_k	d_v	d_{ff}	d_{model}
4	128	128	128	16

Ces observations soulignent l'importance d'une stratégie d'équilibrage des classes lors de l'entraînement du modèle, ainsi que la nécessité de réguler l'impact des paramètres comme d_v et d_k pour une meilleure généralisation et une prédiction plus équilibrée.

7.4 Resultat et comparaison



(a) PCEDNet (b) BiLSTM (c) Multi-Head Attention

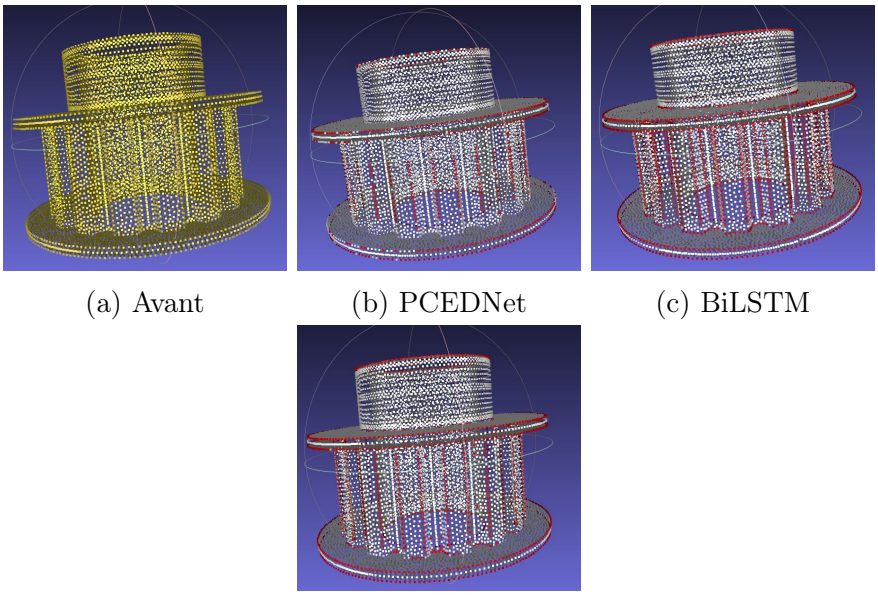
FIGURE 6 – Évolution des courbes d'apprentissage sur 50 époques

TABLE 2 – Performances des modèles selon différentes métriques

Modèle	Parameters	Precision	Recall	MCC	F1	Accuracy	IoU
PCEDNet	1653	0.8285	0.7600	0.7706	0.7928	0.9583	0.6567
BiLSTM	3353	**0.9307**	**0.8070**	**0.8521**	**0.8644**	**0.9728**	**0.7613**
Tssm	15621	0.9019	0.7961	0.8265	0.8457	0.9624	0.7327

TABLE 3 – Temps d'exécution

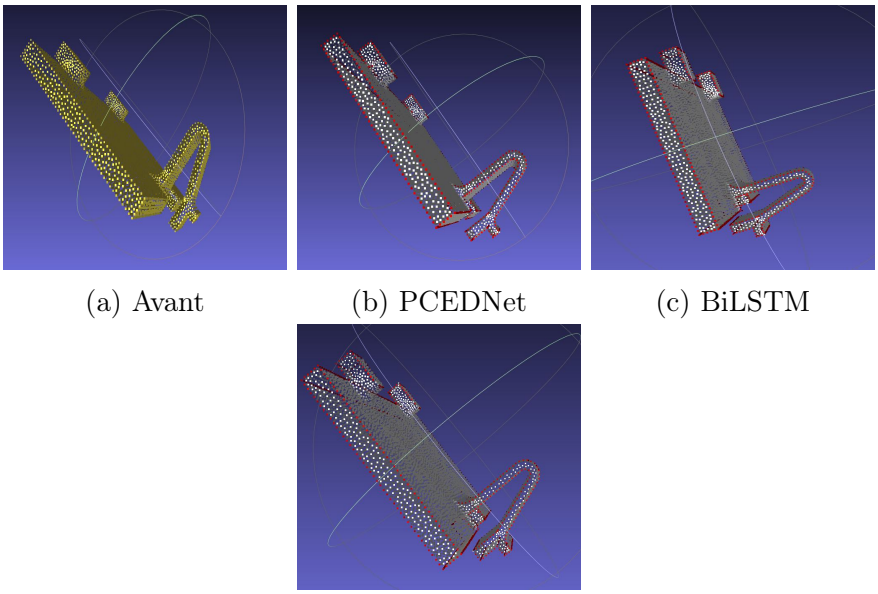
Modèle	Entrainement	Evaluation	Exécution (sur 52 fichiers .ply)
PCEDNet	2min 41s	3min 39s	3min 54s
BiLSTM	3min 4s	3min 6s	2min 40s
Tssm	29min 23s	1 min 57s	3min 34s



(a) Avant (b) PCEDNet (c) BiLSTM

(d) Tssm

FIGURE 7 – Application 0070.ply



(a) Avant (b) PCEDNet (c) BiLSTM

(d) Tssm

FIGURE 8 – Application 0014.ply

8 Modèle Gaussian map

Cette section extrait de l'article [2] avait pour objectif d'exploiter la carte gaussienne discrète en conjonction avec un réseau de neurones de type CNN (Convolutional Neural Network) afin de réaliser une classification entre les points edge et non edge.

La carte gaussienne discrète, utilisée comme élément clé de cette approche, permet de modéliser et de représenter les caractéristiques géométriques et spatiales des points dans le nuage. L'idée sous-jacente était d'utiliser cette carte pour extraire des informations significatives sur les relations spatiales entre les points, tout en capturant les contours et les structures essentielles.

L'intégration de la carte gaussienne discrète avec le CNN vise à fournir au modèle des informations supplémentaires sur la topologie et la disposition spatiale des points, améliorant ainsi sa capacité à discriminer efficacement entre les points edge et non edge.

8.1 La carte gaussienne discrète

Soit N_p le voisinage de p contenant les k voisins les plus proches, et I_p l'ensemble d'indices de N_p. Soit T l'ensemble de toutes les triangulations possibles de p avec ses points de voisinage, soit $k(k-1)$

$$T = \{\Delta_{ij} = \Delta(p, p_i, p_j) \,|\, i \neq j,\ i, j \in I_p\}$$

Le Vecteur normal du traingle Δ_{ij} est :

$$n_{ij} = \overline{pp_i} \times \overline{pp_j}$$

Noté que $n_{ij} = -n_{ji}$

La carte gaussienne discrète du voisinage de $p$peut maintenant être définie comme la correspondance de T sur la sphère unitaire S^2 centrée en p, comme suit :

$$G_p : T \to S^2$$

$$\Delta_{ij} \mapsto x_{ij} := p + \frac{n_{ij}}{\|n_{ij}\|}$$

8.2 Prétraitement des données

Dans cette section, j'ai employé la technique de **sur-échantillonnagede la classe minoritaire**. L'objectif était d'augmenter artificiellement le nombre d'exemples de la classe "edge" en appliquant des transformations légères aux cartes gaussiennes

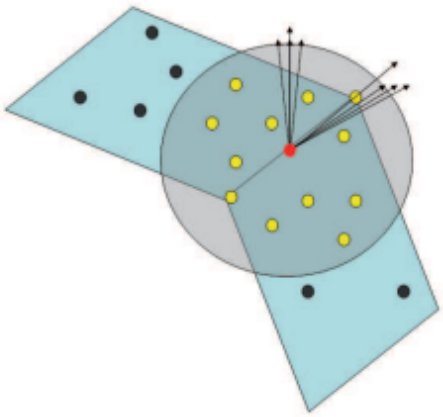


FIGURE 9 – Calcul des vecteurs normaux utilisés pour l'identification des caractéristiques dans un voisinage local [2].

discrètes existantes afin de générer de nouvelles données. Cette démarche vise à équilibrer les effectifs des classes pendant la phase d'entraînement.

Pendant l'entraînement, la majorité des données étaient classées comme "non edges" (représentant plus de 90 %). Cette situation engendra un biais en faveur de la classe "non edge" durant l'entraînement, affectant ainsi les performances du modèle. Le **sur-échantillonnage de la classe minoritaire** aide à remédier à ce déséquilibre en générant de nouvelles données de la classe "edge". Cela permet d'améliorer la capacité du modèle à apprendre des caractéristiques distinctives des deux classes et à effectuer des prédictions plus équilibrées.

8.3 CNNGM

Le modèle **CNNGM** est une architecture fonctionnelle basée sur un réseau de neurones convolutifs (CNN). Il est composé de trois couches convolutives avec des filtres de taille 64, 128 et 256 respectivement. Chaque couche de filtre est ensuite suivie d'une couche de `Dropout` et `MaxPooling2D` pour la réduction de la dimension spatiale, ainsi qu'une couche `Flatten` pour convertir les données en un vecteur 1D.

Les sorties de ces couches sont concaténées et servent d'entrée à deux couches de neurones de taille 128. Enfin, la sortie du modèle est obtenue à travers une couche d'activation `sigmoid`. Cette architecture CNN vise à extraire des caractéristiques significatives des données d'entrée en utilisant des convolutions et des opérations de réduction, avant de les soumettre à des couches de neurones pour la prise de décision finale.

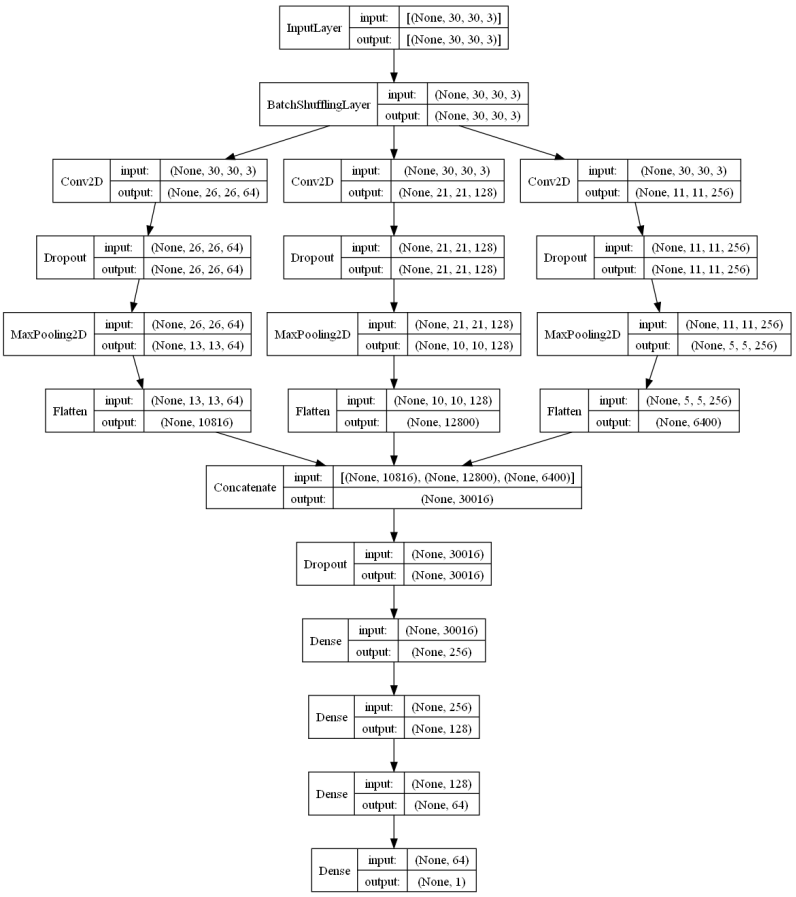


FIGURE 10 – La structure de l'architecture du modèle CNNGM

TABLE 4 – Performances du modèle `CNNGM` selon différentes métriques

Modèle	Parameters	Precision	Recall	MCC	F1	Accuracy	IoU
CNNGM	8076417	0.5009	0.6700	0.0037	0.5732	0.4990	0.4018

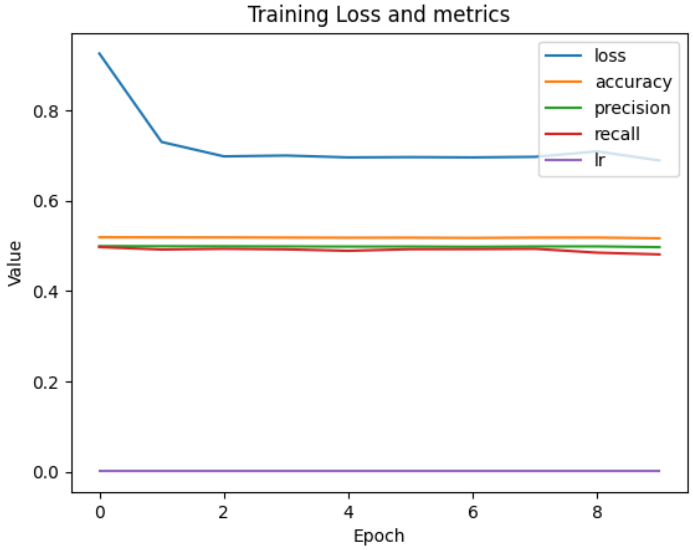


FIGURE 11 – Évolution des courbes d'apprentissage sur 10 époques

8.4 Resultat

Regretttablement, le temps limité n'a pas permis l'exploitation d'autres configurations afin d'améliorer les performances du modèle CNNGM. La recherche de la configuration optimale requiert souvent des expérimentations approfondies et un processus itératif pour ajuster les hyperparamètres et examiner différentes architectures. Dans ce contexte, il aurait été souhaitable d'avoir plus de temps pour mener des expériences plus approfondies et découvrir les réglages les plus efficaces pour le modèle en utilisant ce type de données .

9 Conclusion

En conclusion, cette étude a exploré diverses approches pour la détection des bords dans les nuages de points en utilisant des architectures de réseaux neuronaux avancées. À travers une série d'expérimentations, nous avons évalué les performances de plusieurs modèles clés : `PCEDNet`, `Bidirectional LSTM` et `l'architecture avec Multi-Head Attention`.

Le modèle `PCEDNet` s'est révélé efficace pour exploiter les caractéristiques spatiales et géométriques des points, aboutissant à une détection précise des bords

dans les nuages de points. L'ajout de l'architecture `Bidirectional LSTM` a permis de mieux capturer les dépendances séquentielles entre les vecteurs d'échelle, contribuant ainsi à identifier des motifs complexes au sein des données.

L'introduction de `l'architecture avec Multi-Head Attention` a apporté une dimension nouvelle à notre étude. Elle a démontré sa capacité à établir des liens significatifs entre les vecteurs d'échelle, favorisant une meilleure communication et compréhension mutuelle entre ces vecteurs. Cette approche s'est révélée pertinente pour la détection des bords dans les nuages de points en exploitant les interactions entre les échelles.

En observant la comparaison entre les données issues de **SSM** et celles extraites de la **carte gaussienne discrète**, une tendance se dégage. Les données provenant de **SSM** semblent atténuer le biais favorisant la classe "non edge". Ces données se sont révélées cohérentes et produisent des résultats pertinents lorsqu'associées aux modèles tels que `PCEDNet`, `Bidirectional LSTM` et `l'architecture avec Multi-Head Attention`.

En revanche, les données obtenues à partir de la **carte gaussienne discrète** nécessitent l'utilisation du processus de **sur-échantillonnage de la classe minoritaire** pour compenser le déséquilibre entre les classes. Cependant, cette démarche produit des résultats moyens en comparaison aux autres modèles.

Cette observation souligne la supériorité des données **SSM** dans la réduction du biais et l'amélioration des performances des modèles de détection des bords. En privilégiant l'utilisation de ces données, il est possible d'obtenir des résultats plus fiables et cohérents, tout en éliminant la nécessité d'appliquer des techniques de sur-échantillonnage pour équilibrer les classes.

Cependant, des limitations liées aux contraintes de temps et de ressources ont restreint notre capacité à explorer en profondeur différentes configurations et architectures pour le `CNNGM`. Malgré cela, les résultats obtenus sont prometteurs et ouvrent la voie à de nouvelles perspectives de recherche pour améliorer davantage les performances de ces modèles de détection des bords.

En somme, cette étude contribue à l'évolution du domaine de la détection des bords dans les nuages de points en combinant différentes architectures de réseaux neuronaux. Les avancées réalisées sont encourageantes et appellent à des investigations futures pour optimiser les modèles existants et explorer de nouvelles voies pour une détection des bords encore plus précise et robuste.

10 Annexe

10.1 Code

10.1.1 Code :Nettoyage des données

```
def nanClean(path:str, extension:str):
```

```
    """ Remove point with its correponding label which
       containing Nan value"""
    for numpy_file in get_numpy_file(path, extension=extension
       , shuffle=False):
        ssm_file = np.load(os.path.join(path, numpy_file[0]))
        lb_file = np.load(os.path.join(path, numpy_file[1]))
        ssm_mask = np.isnan(ssm_file).any(axis=2).any(axis=1)
        np.save(os.path.join(path, numpy_file[0]), ssm_file[~
           ssm_mask], allow_pickle=False)
        np.save(os.path.join(path, numpy_file[1]), lb_file[~
           ssm_mask], allow_pickle=False)
```

1

10.1.2 Code :Lecture des données

```
def ToNumpy(extension:str):
    """convert a ssm, lb, ply to numpy file"""
    def ssm2numpy(path, name, start=None):
        start_ = (12 if start == None else start)
        with open(os.path.join(path, name)) as f:
                l = f.readlines()
                return np.array([[np.float32(s) for s in l_.
                   split()] for l_ in l[start_:]], dtype=np.
                   float32).reshape(
                    (int(l[2].split()[0]), 16, 6))
    def lb2numpy(path, name, start=None):
        start_ = (1 if start == None else start)
        with open(os.path.join(path, name)) as f:
            return np.array([float(s) for s in f.readlines()[
               start_:]], dtype=np.float32)

    def ply2numpy(path, name, start=None):
        start_ = (13 if start == None else start)
        with open(os.path.join(path, name)) as f:
            l = f.readlines()
            return np.array([[np.float32(s) for s in l_.split
               ()] for l_ in l[start_:]], dtype=np.float32)

    if extension == EXTENSION_SSM:
        return ssm2numpy
    elif extension == EXTENSION_LB :
        return lb2numpy
```

1. le reste du code est dans Convert2numpy.py

```
    else :
        return ply2numpy


def save2numpy(array, path:str, extension:str)->None:
    """Save to a NumPy file while preventing the use of pickle
       ."""
    np.save(f'{path.rstrip(extension)}_{extension}.npy', array
       , allow_pickle=False)

def convertAndSave(from_:str, to:str, extension:str, start=
   None)->None:
    """Convert and save file(ssm, ply) with the corresponding
       lb file to numpy extension"""
    if extension != EXTENSION_LB and extension !=
       EXTENSION_SSM and extension != EXTENSION_PLY:
        raise UnknownExtensionException()
    if not os.path.exists(to):
        os.mkdir(to)
    for file in tqdm(get_ssm_lb_ply_file(from_, extension),
       total=len(list(get_ssm_lb_ply_file(from_, extension))))
       :
        save2numpy(ToNumpy(extension)(from_, file, start), os.
           path.join(to, file), extension)
```

2

10.1.3 Code :Generateur de batches

```
class DataGenerator(tf.keras.utils.Sequence):
    def __init__(self, path, f_input, batch_size=BATCH_DEFAULT
       , split=1 ,extension="_.ssm.npy", shuffle=True):
        self.batch_size = batch_size
        self.split = split
        self.paths = path
        self.index = 0
        self.gen_input = f_input
        self.shuffle = shuffle # If it's True it will shuffle
           batches
        self.extension = extension
        self.total_points = self.get_total_point()
        self.generator = get_numpy_file(path, extension=self.
           extension, shuffle=shuffle)
```

2. le reste du code est dans Convert2numpy.py

```
        self.load_()
        self.diff = 0

    def get_input_shape(self):
        if self.df_x.size != 0:
            return self.df_x.shape

    def load_(self):
        """
        Import a file (either ssm or ply format) along with
           its corresponding .lb file.
        df_x = for ssm or ply coordinates
        df_y = labels
        It raise an exception when there is no file left
        """
        try:
            x, y = next(self.generator)
            self.df_x = np.load(os.path.join(self.paths, x))
            self.df_y = np.load(os.path.join(self.paths, y))
        except StopIteration:
            self.on_epoch_end()
            raise StopIteration()

    def on_epoch_end(self):
        """On the epoch end it create a new generator"""
        self.generator = get_numpy_file(self.paths, extension=
           self.extension, shuffle=self.shuffle)
        self.index = 0
        self.diff = 0

    def get_total_point(self):
        """ return the total point on sm or ply files"""
        return np.array([np.load(os.path.join(self.paths, i
           [0])).shape[0] for i in get_numpy_file(self.paths,
           extension=self.extension, shuffle=self.shuffle)]).
           sum()

    def __len__(self):
        """calculate the numbre of batches """
        total_length = self.total_points*self.split
        rest = total_length % self.batch_size
        q = (total_length - rest)/self.batch_size
        return int(q) + int(rest != 0)
```

```
    def __getitem__(self, index):
        """return a batches """
        batches = []
        labels = []
        if (self.index + 1) * self.batch_size + self.diff <=
           self.df_x.shape[0]:
            batches.append(self.df_x[self.index * self.
               batch_size+self.diff:(self.index + 1) * self.
               batch_size+self.diff])
            labels.append(self.df_y[self.index * self.
               batch_size+self.diff:(self.index + 1) * self.
               batch_size+self.diff])
            if (self.index + 1) * self.batch_size + self.diff
               == self.df_x.shape[0] and index < self.__len__
               () - 1:
                self.load_()
                self.diff = 0
                self.index = 0
            else:
                self.index += 1

        else:

            batches.append(self.df_x[self.index * self.
               batch_size + self.diff:])
            labels.append(self.df_y[self.index * self.
               batch_size + self.diff:])
            self.diff += int((self.index + 1) * self.
               batch_size - self.df_x.shape[0])
            self.index = 0
            try:
                self.load_()
                if self.diff > self.df_x.shape[0]:
                    while True:
                        batches.append(self.df_x)
                        labels.append(self.df_y)
                        self.diff -= self.df_x.shape[0]
                        if self.diff <= self.df_x.shape[0]:
                            break
                labels.append(self.df_y[:self.diff])
                batches.append(self.df_x[:self.diff])
            except StopIteration :
                batches[-1], labels[-1] = tile(batches[-1],
                   labels[-1], batch_size=self.batch_size)
```

```
        batches, labels = np.concatenate(batches, axis=0), np.
            concatenate(labels, axis=0)
        return self.gen_input(batches), labels.reshape((labels
            .shape[0], 1))
```

3

10.1.4 Code :PCEDNet

```
def generate_input(x):
    return [x[:, i, :] for i in range(x.shape[1])]


class PCEDNet:

    def __init__(self):
        pass

    def create_tree(self, i, l):
        if i == 0:
            input1 = tf.keras.layers.Input(shape=(6))
            input2 = tf.keras.layers.Input(shape=(6))
            l.extend([input1, input2])
        else:
            input1 = self.create_tree(i - 1, l)
            input2 = self.create_tree(i - 1, l)
        return (tf.keras.layers.Dense(6, activation="selu",
            use_bias=True, kernel_initializer=tf.keras.
            initializers.lecun_uniform())
                (tf.keras.layers.Concatenate(axis=-1)([input1,
                    input2])))

    def build(self):
        l = []
        input1 = self.create_tree(2, l)
        input2 = self.create_tree(2, l)
        dense1 = tf.keras.layers.Dense(16, activation="selu",
            use_bias=True, kernel_initializer=tf.keras.
            initializers.lecun_uniform(), bias_initializer=tf.
            keras.initializers.lecun_uniform())(
            tf.keras.layers.Concatenate(axis=-1)([input1,
                input2]))
        dense1 = tf.keras.layers.BatchNormalization()(dense1)
```

3. le reste du code est dans DataGenerator.py

```
        dense2 = tf.keras.layers.Dense(16, activation="selu",
            use_bias=True, kernel_initializer=tf.keras.
            initializers.lecun_uniform(), bias_initializer=tf.
            keras.initializers.lecun_uniform())(dense1)
        dense2 = tf.keras.layers.BatchNormalization()(dense2)
        output = tf.keras.layers.Dense(1, activation="sigmoid"
            )(dense2)
        self.model = tf.keras.models.Model(inputs=l, outputs=
            output)
        return self.model
```

10.1.5 Code :LSTM bidirectionnel

```
class BiLSTM:

    def __init__(self, unit = 16):
        self.activation_func = "selu"
        self.activation_func2 = "sigmoid"
        self.forward_layer = tf.keras.layers.LSTM(unit,
            return_sequences=False)
        self.backward_layer = tf.keras.layers.LSTM(unit,
            go_backwards=True, return_sequences=False)
        self.bidirectional = tf.keras.layers.Bidirectional(
            self.forward_layer, backward_layer=self.
            backward_layer, merge_mode='concat')
        self.dropout = tf.keras.layers.Dropout(0.1)
        self.dense_1 = tf.keras.layers.Dense(12, activation="
            selu", use_bias=True, bias_initializer=tf.keras.
            initializers.lecun_uniform())
        self.dense_2 = tf.keras.layers.Dense(1, activation="
            sigmoid")

    def build(self):
        input_ = tf.keras.layers.Input(shape=(16, 6))
        output = self.bidirectional(input_)
        output = self.dropout(output)
        output = self.dense_1(output)
        output = self.dense_2(output)
        self.model = tf.keras.models.Model(inputs=input_,
            outputs=output)
        return self.model
```

4

4. le reste du code est dans BiLSTM.py

10.1.6 Code :Multi-Head Attention

```
class EncoderLayer(tf.keras.layers.Layer):

    def __init__(self, h, d_k, d_v, d_ff, d_model, rate,
       activation_function_nn="selu", **kwargs):
        super(EncoderLayer, self).__init__(**kwargs)
        self.multihead_attention = tf.keras.layers.
           MultiHeadAttention(h, d_k, d_v, dropout=rate)
        self.dropout1 = tf.keras.layers.Dropout(rate)
        self.add_norm1 = AddNormalization()
        self.feed_forward = NN(d_ff, d_model,
           activation_function_nn=activation_function_nn)
        self.add_norm2 = AddNormalization()

    def call(self, x, padding_mask, training):
        multihead_output = self.multihead_attention(x, x,
           padding_mask, return_attention_scores=False)
        multihead_output = self.dropout1(multihead_output,
           training=training)
        addnorm_output = self.add_norm1(x, multihead_output)
        output = self.feed_forward(addnorm_output)
        return self.add_norm2(output, addnorm_output)


class Encoder(tf.keras.layers.Layer):
    def __init__(self, h, d_k, d_v, d_ff, d_model, n, rate,
       activation_function_nn="selu", **kwargs):
        super(Encoder, self).__init__(**kwargs)
        self.dropout = tf.keras.layers.Dropout(rate)
        self.encoder_layer = [EncoderLayer(h, d_k, d_v, d_ff,
           d_model, rate, activation_function_nn=
           activation_function_nn) for _ in range(n)]

    def call(self, input_scales, padding_mask, training):
        x = self.dropout(input_scales, training=training)
        for i, layer in enumerate(self.encoder_layer):
            x = layer(x, padding_mask, training)
        return x


class TssmNet:

    def __init__(self, input_shape = (16, 6), h=4, d_k=64, d_v
       =64, d_ff=12, d_model=32, n=1, dropout_rate=0.1,
```

```
        activation_function="selu"):
         self.activation_func = activation_function
         self.activation_func2 = "sigmoid"
         self.input_1 = tf.keras.layers.Input(shape=input_shape
            )
         self.encoder = Encoder(h, d_k, d_v, d_ff, d_model ,n,
            dropout_rate)
         self.dropout = tf.keras.layers.Dropout(0.1)
         self.flatten = tf.keras.layers.Flatten()
         self.dense = tf.keras.layers.Dense(1, activation="
            sigmoid", use_bias=True, bias_initializer=tf.keras.
            initializers.lecun_uniform())

    def build(self):
         input_ = self.input_1
         output = self.encoder(input_, None, True)
         output = self.flatten(output)
         output = self.dense(output)
         self.model = tf.keras.models.Model(inputs=input_,
            outputs=output)
         return self.model
```

5

10.1.7 L'extraction de la carte gaussienne discrète

```
def gauss_map(array_obj, point, n_neighbors=N_NEIGHBORS):
    coordinates = get_neighbors(array_obj, point, n_neighbors=
       n_neighbors + 1)
    sub_vec = (coordinates - point)[1:]
    dgm = np.zeros((n_neighbors, n_neighbors, 3))
    for i in prange(n_neighbors):
        for j in prange(n_neighbors):
            if i != j:
                nvt = np.cross(sub_vec[i], sub_vec[j])
                norm = np.linalg.norm(nvt)
                dgm[i, j] = point + (nvt / norm if norm != 0
                   else nvt)
                norm_dmg = np.linalg.norm(dgm[i, j])
                dgm[i, j] = (dgm[i, j] / norm_dmg if norm_dmg
                   != 0 else dgm[i, j])
    return dgm.reshape((1, n_neighbors * n_neighbors, 3))
```

5. le reste du code est dans TssmNet.py

```
@njit(nogil=True)
def create_gauss_feature_single(point, array_obj, n_neighbors=
    N_NEIGHBORS):
    return gauss_map(array_obj, point.reshape((1, 3)),
        n_neighbors=n_neighbors)


def create_gauss_feature(array_obj, n_neighbors=N_NEIGHBORS):
    with ThreadPoolExecutor() as executor:
        results = typed.List(
            list(executor.map(lambda point:
                create_gauss_feature_single(point, array_obj,
                n_neighbors), array_obj)))
    return results


@njit(nogil=True)
def numba_concatante(arr_list):
    num_arrays = len(arr_list)
    total_length = num_arrays * arr_list[0].shape[0]
    result = np.empty((total_length, arr_list[0].shape[1],
        arr_list[0].shape[2]), dtype=arr_list[0].dtype)
    index = 0
    for arr in arr_list:
        result[index:index + arr.shape[0], :, :] = arr
        index += arr.shape[0]
    return result


def ply2gauss_map(path_from: str, path_to: str, n_neighbors=
    N_NEIGHBORS, split=1, add_lb=True):
    total = len(list(get_numpy_file(path_from, extension="_.
        ply.npy", shuffle=False)))
    stop = int(total * split)
    index = 0
    with tqdm(total=stop) as bar:
        for filename in get_numpy_file(path_from, extension="_
            .ply.npy", shuffle=False):
            array_obj = np.load(os.path.join(path_from,
                filename[0]))[:, :3]
            if array_obj.shape[0] <= 30000:
                gauss_map_features_list = create_gauss_feature
                    (array_obj, n_neighbors=n_neighbors)
```

```
                gauss_map_features = numba_concatante(
                    gauss_map_features_list)
                np.save(os.path.join(path_to, filename[0]),
                    gauss_map_features, allow_pickle=False)
                if add_lb:
                    np.save(os.path.join(path_to, filename[1])
                        , np.load(os.path.join(path_from,
                        filename[1])),
                            allow_pickle=False)
                index += 1
                bar.update(1)
                if index >= stop:
                    break
```

6

10.1.8 Le sur-échantillonnagede la classe minoritaire

```
def oversample_minority_class(gauss_maps, labels, target_ratio
    =0.5):
    unique_labels, counts = np.unique(labels, return_counts=
        True)
    majority_class_label = unique_labels[np.argmax(counts)]
    minority_class_label = unique_labels[np.argmin(counts)]

    majority_class_indices = np.where(labels ==
        majority_class_label)[0]
    minority_class_indices = np.where(labels ==
        minority_class_label)[0]

    oversampled_minority_indices = np.random.choice(
        minority_class_indices,size=int(target_ratio * len(
        majority_class_indices)), replace=True)

    mask = np.concatenate([majority_class_indices,
        oversampled_minority_indices])
    np.random.shuffle(mask)
    oversampled_gauss_maps = gauss_maps[mask]
    oversampled_labels = labels[mask]

    return oversampled_gauss_maps, oversampled_labels
```

6. le reste du code est dans Cal_discrete_gauss_map.py

```
def oversample_class(path_from: str, target_ratio=0.5):
    total = len(list(get_numpy_file(path_from, extension="_.
       ply.npy", shuffle=False)))
    with tqdm(total=total) as bar:
        for filename in get_numpy_file(path_from, extension="_
           .ply.npy", shuffle=False):
            oversampled_gauss_maps, oversampled_labels =
               oversample_minority_class(
                np.load(os.path.join(path_from, filename[0])),
                np.load(os.path.join(path_from, filename[1])),
                target_ratio=target_ratio)
            np.save(os.path.join(path_from, filename[0]),
               oversampled_gauss_maps, allow_pickle=False)
            np.save(os.path.join(path_from, filename[1]),
               oversampled_labels, allow_pickle=False)
            bar.update(1)
```

7

10.1.9 CNNGM

```
class CNNGM:

    def __init__(self, input_shape, activation_function="selu"
       ):
        self.activation_func = activation_function
        self.input_1 = tf.keras.layers.Input(shape=input_shape
           )

        self.batchShuffle = BatchShufflingLayer()

        self.conv2D_1 = tf.keras.layers.Conv2D(64, (5, 5),
           activation=self.activation_func , use_bias=True,
                                               kernel_initializer
                                                  =tf.keras.
                                                  initializers.
                                                  lecun_uniform
                                                  (),
                                               bias_initializer=
                                                  tf.keras.
                                                  initializers.
                                                  lecun_uniform
                                                  ())
```

7. le reste du code est dans Cal_discrete_gauss_map.py

```
        self.conv2D_2 = tf.keras.layers.Conv2D(128, (10, 10),
            activation=self.activation_func , use_bias=True,
                                               kernel_initializer
                                                  =tf.keras.
                                                  initializers.
                                                  lecun_uniform
                                                  (),
                                               bias_initializer=
                                                  tf.keras.
                                                  initializers.
                                                  lecun_uniform
                                                  ())
        self.conv2D_3 = tf.keras.layers.Conv2D(256, (20, 20),
            activation=self.activation_func , use_bias=True,
                                               kernel_initializer
                                                  =tf.keras.
                                                  initializers.
                                                  lecun_uniform
                                                  (),
                                               bias_initializer=
                                                  tf.keras.
                                                  initializers.
                                                  lecun_uniform
                                                  ())

        self.activation_func = activation_function
        self.activation_func2 = "sigmoid"

        self.dense_1 = tf.keras.layers.Dense(256, activation=
            self.activation_func , use_bias=True,
                                               kernel_initializer
                                                  =tf.keras.
                                                  initializers.
                                                  lecun_uniform
                                                  (),
                                               bias_initializer=
                                                  tf.keras.
                                                  initializers.
                                                  lecun_uniform
                                                  ())

        self.dense_2 = tf.keras.layers.Dense(128, activation=
            self.activation_func , use_bias=True,
                                               kernel_initializer
                                                  =tf.keras.
```

```
                                                    initializers.
                                                    lecun_uniform
                                                    (),
                                                bias_initializer=
                                                    tf.keras.
                                                    initializers.
                                                    lecun_uniform
                                                    ())

        self.dense_3 = tf.keras.layers.Dense(64, activation=
            self.activation_func , use_bias=True,
                                                kernel_initializer
                                                    =tf.keras.
                                                    initializers.
                                                    lecun_uniform
                                                    (),
                                                bias_initializer=
                                                    tf.keras.
                                                    initializers.
                                                    lecun_uniform
                                                    ())
        self.dense_4 = tf.keras.layers.Dense(1, activation=
            self.activation_func2 , use_bias=True,
                                                kernel_initializer
                                                    =tf.keras.
                                                    initializers.
                                                    lecun_uniform
                                                    (),
                                                bias_initializer=
                                                    tf.keras.
                                                    initializers.
                                                    lecun_uniform
                                                    ())

    def build(self):
        input__ = self.input_1
        input_ = self.batchShuffle(input__)
        concat = tf.keras.layers.Concatenate()([tf.keras.
            layers.Flatten()(tf.keras.layers.MaxPool2D((2, 2))(
            tf.keras.layers.Dropout(0.1)(conv2D(input_)))) for
            conv2D in [self.conv2D_1, self.conv2D_2, self.
            conv2D_3]])
        output = tf.keras.layers.Dropout(0.1)(concat)
        output = self.dense_1(output)
        output = self.dense_2(output)
```

```
        output = self.dense_3(output)
        output = self.dense_4(output)
        self.model = tf.keras.models.Model(inputs=input__,
            outputs=output)
        return self.model
```

8

10.2 Architectures

8. le reste du code est dans CNNGM.py

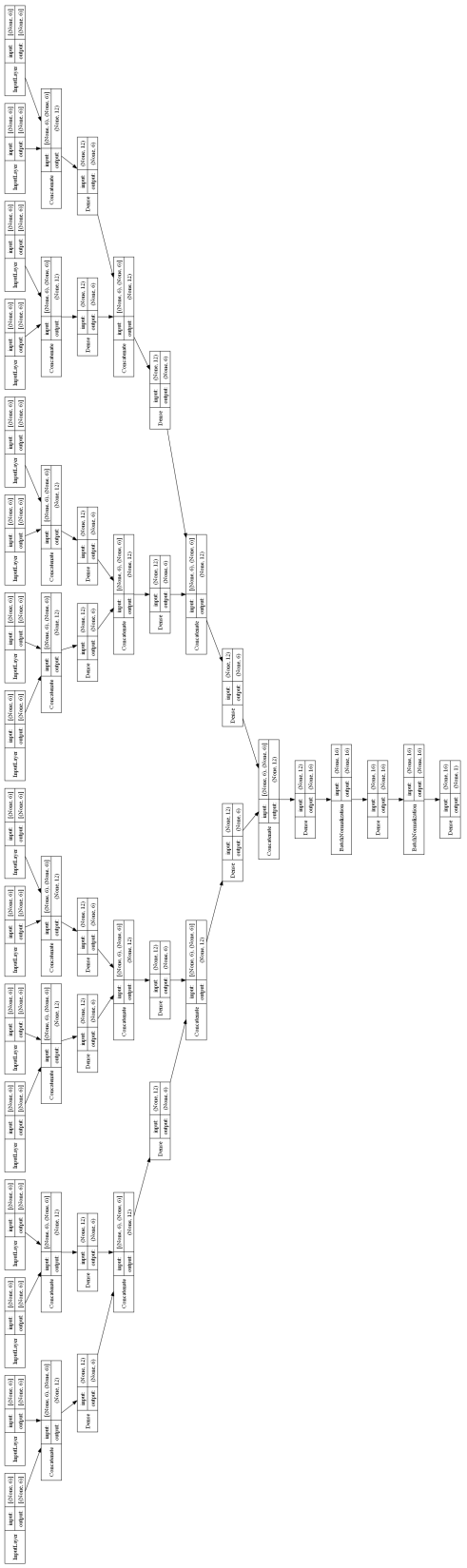


FIGURE 12 – PCEDNet

Références

[1] THOMAS PELLEGRINI MATHIAS PAULIN LOIC BARTHE CHEMS-EDDINE HIMEUR, THIBAULT LEJEMBLE and NICOLAS MELLADO. PCEDNet : A Lightweight Neural Network for Fast and Interactive Edge Detection in 3D Point Clouds. 2021.

[2] Christopher Weber, Stefanie Hahmann, and Hans Hagen. Sharp Feature Detection in Point Clouds. 2010.

[3] Gunter Klambauer, Thomas Unterthiner, and Andreas Mayr. Self-Normalizing Neural Networks. 2017.