Rapport de Stage
Etude et mise en place du Deep Learning sur des données satellites du CNES

Projet de Fin d'Etudes

Maxence Ronzié
Encadrants : Corentin LAPEYRE & Isabele D'AST
Encadrant pédagogique : Mathieu FAVERGE

Table des matières

1 Introduction

1.1 Cadre de l'entreprise

Le CERFACS (Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique) est un laboratoire privé à taille de PME (∼150 employés) soutenu par sept actionnaires majeurs qui sont

- le CNES (Centre National d'Etudes Spatiales)
- EDF (Electricité De France)
- Total
- Météo France
- l'ONERA (centre français de recherche en aérospatiale)
- SAFRAN (groupe international de haute technologie)
- Airbus

En collaboration avec d'autres organismes de recherche comme le CNRS, le CERFACS propose à ses actionnaires de développer des recherches scientifiques et techniques directement liées aux enjeux des actionnaires. Le CERFACS est constitué de six équipes (de tailles hétérogènes) :

- Mécanique des Fluides Numérique (CFD)
- Algorithmes Parallèles (Algo)
- Modélisation du Climat et de son Changement Global (GlOBC)
- Aviation et Environnement
- COOP : équipe récente interdisciplinaire permettant de faire le lien entre toutes les équipes au CERFACS.
- Equipe Informatique et Support Utilisateur (CSG)

C'est entre l'équipe COOP et CSG que mon stage s'inscrit pour les intérêts du CNES.

1.2 Rôle et attentes

Dans ce contexte, mon rôle est d'explorer sur les différentes architectures matérielles internes mais aussi externes au CERFACS la mise en place de ces différents modèles de réseaux neuronaux. Ainsi il y a un rôle double d'étude et de développement pour le compte du CNES. Le CERFACS attend ainsi également de ma part une prise d'initative concernant cette double responsabilité.

De ce fait, j'ai du dans un premier temps me former aux différentes techniques utilisées pour le deep learning. Les attentes me concernant sont donc celles typique d'un ingénieur en recherche et développement : savoir travailler et apprendre de manière autonome, ainsi que prendre des initiatives dans le but d'effectuer des recherches et leurs développement.

Par ailleurs, une communauté data science s'est récemment ouverte au CERFACS avec un nombre croissant de participants. Chaque personne, stagiaire, doctorant ou permanent travaillant dans le domaine est invité à présenter son travail aux autres. Je suis donc amené à communiquer et présenter mes résultats régulièrement.

1.3 Environnement de travail

Mes deux encadrants Corentin Lapeyre et Isabelle D'Ast guident mon travail pour l'utilisation des ressources matérielles et l'environnement informatique ainsi que sur les aspects scientifiques. En effet, chaque semaine un point est fait sur ce qui a était fait et ce qui serait à faire pour la semaine suivante. Même s'il y a un but général vers lequel on souhaite aller avec ce stage, celui-ci reste très ouvert car partant de questions très larges du CNES. Ainsi, on avance pas à pas et le travail de la semaine n + 1 est déterminé à partir de ce qui a été fait à la semaine n.

Concernant les architecture matérielles disponibles, le CERFACS dispose d'une machine nommée IBO disposant d'une carte graphique Tesla V100 (16Gb RAM) pouvant atteindre plus de 120 Tflop/s avec l'activation de TPU (Tensor Processing Unit, créés spécialement pour le deep learning car celui ci utilisent grandement les tenseurs pour ses calculs). D'autres machines internes au CERFACS sont disponibles comme Jadawin mais moins performante car plus ancienne (Quadro M5000 (8Gb RAM) 3.53 TFlop/s)

Par ailleurs, un compte m'a été ouvert sur des architectures externes au CERFACS comme à l'Idris (Institut du Développement et des Ressources en Informatique Scientifique) sur le calculateur prototype Ouessant, disposant de 12 noeuds et de 4 cartes Tesla P100 par noeud (21 TFlop/s & 16Gb RAM chacune)

En ce qui concerne le framework utilisé pour faire du deep learning, la bibliothèque Keras (Python) de Francois Chollet à été choisie. Elle possède en backend TensorFlow. Python est le langage principal pour ce stage, car de manière générale c'est un langage bien fourni en bibliothèques pour le champ de la data science.

Enfin, concernant le jeu de données sur lequel je travaille actuellement, un fichier de 120 Giga-Octets m'a été donné contenant d'une part les images en fichiers .tif et les masques associés. Dans notre cas, un masque est une matrice binaire contenant des 0 et des 1. Si l'élément $m_{i,j}$ du masque est à 1 cela signifie que le pixel en position (i, j) est un pixel d'impact.

Une définition claire d'impact permettant de le différencier d'un cratère n'existe malheureusement pas du fait qu'un impact n'est autre qu'un cratère récent (qui vient d'avoir lieu donc). L'absence de quantification à ce niveau là (entre ce qu'on entend par 'récent') pose là une difficulté pour le stage et notre réseau de neurone qui doit donc apprendre à différencier des impact à des cratères selon l'étiquettage qui lui à été transmis (le masque). Ce masque à été fait à la main de la manière suivante :

L'être humain pointe avec la souris sur ce qui lui semble être un impact, et par croissance de régions, un logiciel vient compléter la zone correspondante. Il est en effet impensable qu'un humain encode pixel par pixel un ou plusieurs impacts sur des centaines d'images

Ainsi les données comportent des imperfections entre impacts qui auraient été oubliés par l'être humain et ceux rajoutés par erreur humaine.

1.4 Mise en situation

Le CNES a contacté le CERFACS dans le but de développer un code de deep learning permettant de faire de la segmentation d'image sur des données satellites de Mars provenant de la NASA. A l'heure actuelle, il n'y a plus à prouver l'efficacité de ce champ de l'intelligence artificielle pour la segmentation d'image. Nous avons ainsi pu trouver sur le web une compétition Kaggle [1] qui traitait d'un sujet similaire : segmentation d'image satellite.

En particulier, une interview détaillée de l'équipe ayant terminé 3ème est disponible. [2] La majorité des équipes figurant en tête de classement ont utilisé une architecture commune pour le réseau de neurones : un u-net [1] Les compétitions Kaggle sont très sérieuses en data science (récompense de $100 000) ce qui nous permet légitimement de considérer que cette architecture est très bien adaptée au problème. Nous sommes donc partis sur cette solution à implémenter.

Ainsi, comme dit en section 2 (Rôle et attentes) une première étape est de se former à toutes les notions relatives aux réseaux neuronaux. Pour cela il y a les tutoriaux de F.Chollet disponibles sur son github [3]

Ces tutoriaux ne rentrent pas en profondeur dans la théorie (ex : convolution) mais permettent tout de même d'avoir une idée de fond sur les raisons de tel ou tel choix pour tel usage. Ces tutoriaux sous forme de notebook (jupyter) permettent d'avoir un très bon point de vue sur l'implémentation de ces réseaux neuronaux.

En début de stage, le CNES n'avait pas encore fourni ses données satellites sur Mars. De fait, ces premieres semaines ont été utilisées pour se former à travers ces tutoriaux principalement.

1. https ://www.kaggle.com/c/dstl-satellite-imagery-feature-detection
2. http ://blog.kaggle.com/2017/05/09/dstl-satellite-imagery-competition-3rd-place-winners-interview-vladimir-sergey/
3. https ://github.com/fchollet/deep-learning-with-python-notebooks

2 Etude et mise en oeuvre du Deep Learning

Ce stage présente deux facettes : étude et mise en place du deep learning sur les données du CNES puis une analyse des performances obtenues. Cette section se divise selon ces deux parties.

2.1 Analyse de notre réseau de neurone

L'étape suivante est naturellement d'implémenter un u-net et de le tester sur un jeu de données d'image satellite du CNES. Nous sommes ainsi parti d'une versions basique d'un réseau de neurones. Nous souhaitons partir de la base pour maitriser au maximum ce que l'on fait et pouvoir mesurer les effets des choix que l'on fera par la suite (effet sur le temps d'exécution, la pertinence des résultats etc..) La figure ci dessous montre l'architecure du u-net utilisé :

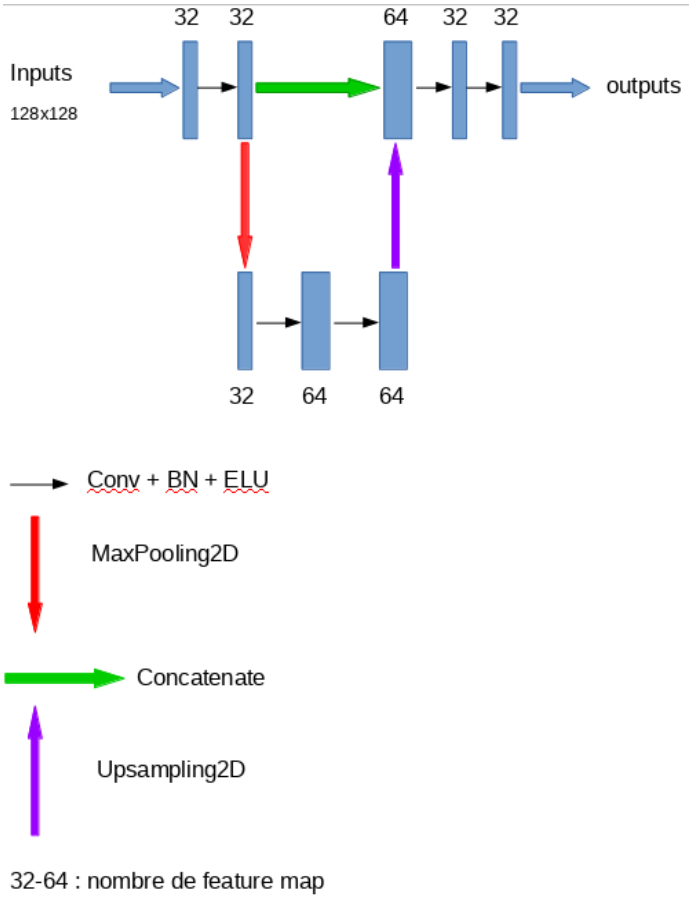

FIGURE 1 – Architecture du U-net utilisé dans cette étude

Les détails de cette implémentation et notamment de cette légende sont données page suivante

2.1.1 Couche de convolution

Le choix de partir sur des couches de convolutions se justifie par la nature de cette opération. Elle permet d'apprendre les pattern locaux des données, ce qui convient parfaitement à la segmentation d'image. Elle est souvent oppposée aux couches denses pour mieux comprendre son intérêt et son fonctionnement. Une couche dense prendra la donnée entière et apprendra les caractéristiques globales. Tandis qu'une couche de convolution va segmenter la donnée pour en apprendre les caractéristiques locales. Un exemple concret : supposons que l'on veuille faire apprendre à une machine à dire s'il y a un chat sur une image ou non. Si le jeu de données est majoritairement constitué de chat pris de face, alors le réseau apprendra qu'un chat possède deux yeux, deux oreilles, deux pattes avant, deux paires de touffes de moustaches etc.. car il a appris les caractéristiques globales. Que se passe-t-il si sur une image un chat se trouve sous un autre angle. Le réseau ne verra alors plus qu'un oeil, une oreille, une touffe de moustaches. Il ne détectera pas l'animal. En revanche, la couche de convolution quant à elle aura appri les caractéristiques locales, et donc aura appris qu'un chat possède des moustaches, des oreilles poitues, des yeux dont la pupille est ovale etc .. Si donc le chat se présente sous forme allongée, debout, de face, d'arrière etc.. il le détectera.

Si nous voyons les images en couleur et que cette tâche nous paraît facile, une machine elle ne voit les images que comme des matrices de pixel donc des matrices réelles. Ainsi le réseau de convolution analyse des sous blocs de matrices et étudie la disposition des pixels au sein de ces sous blocs. Le produit de convolution est une opération mathématique qui peut se faire dans le cas discret. Elle induit de nombreuses multiplication matricielles. Ci-après se un exemple d'un produit de convolution illustré par une figure : On note $I \in \mathcal{M}(R)$ la matrice réelle d'input et $K \in \mathcal{M}(R)$ la matrice des poids appelé souvent Kernel (les poids qu'on applique à chaque paramètre entrainable) Le produit de convolution est donné par :

$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, i-n)K(m,n)$

Ce qui est illustré par la figure qui suit :

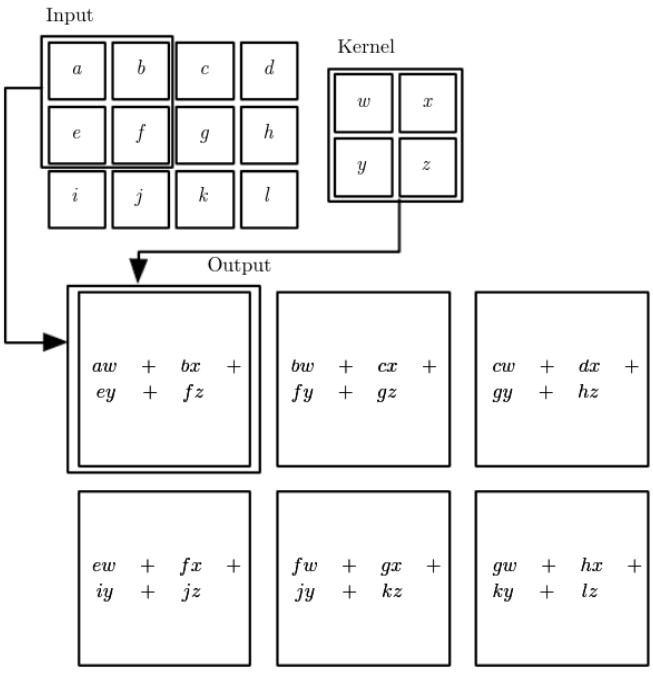

FIGURE 2 – Convolution

Plus que pour satisfaire la curiosité du lecteur, ce dernier paragraphe est surtout là pour donner un apercu des lourds calculs que peuvent représenter un réseau de neurones comportant de nombreuses couches de convolution et donc vient justifier la problématique sur l'aspect performance que souhaite approfondir le CNES.

2.1.2 Couche de Batch Normalization

La couche de BatchNormalization vient normaliser les valeurs entre 0 et 1 par exemple. En toute rigueur, la BatchNormalization normalise les données en soustrayant la moyenne du batch de données et en divisant par sa moyenne. Cette opération va permettre de rendre le réseau plus stable et aussi va permettre d'accélerer sa phase d'apprentissage. D'un point de vue mathématique, on ne perd pas d'information en passant de valeurs entre -10 et 10 qu'entre -1 et 1. Cela permet en outre de réduire les écarts grossier qui pourraient avoir un trop gros impact sur la mise à jour des poids (lors de la descente du gradient). Par ailleurs elle permet aussi de le rendre plus générique en terme de prédiction. En effet supposons qu'on ait entrainé un réseau à détecter des chats dans des centaines d'images mais que les chats soient majoritairement d'une couleur sombre (noir, gris). Sans BatchNormalisation, cette couleur aura mettons pour valeur 5. Si maitenant le réseau est confronté à des chats dont la robe est de couleur plus vive comme le orange ou le blanc, ce champ de valeur pour ces couleurs allant de 50 à 100, le réseau serait perdu et reconnaitrait moins les chats de cette couleur. Cet exemple montre l'importance d'une couche de BatchNormalisation, et ce après chaque couche de convolution si possible.

2.1.3 Activation ELU

Une couche d'activation permet au neurone $n+1$ dadapter la donnée et ses valeurs qu'il recoit de la part du neurone n. Il existe une multiude de fonction d'activation de base disponible sur Keras. Nous avons choisi la fonction ELU mais nous aurions pu choisir aussi SoftPlus. Ces deux fonctions sont en effet très proches d'un point de vue mathémtique. Mis à part pour des cas d'utilisation bien spécifique, ce paramètre est celui qui joue le moins sur le résultat final dans notre cas. Il faut toutefois noter que la fonction d'activation est indispensable car elle introduit une non linéarité dans le réseau. En effet si les n couches étaient linéaires entre elles leur enchaînement reserait équivalent à une seule couche linéaire.

2.1.4 Couche de MaxPooling

Afin de limiter le nombre de calculs et de limiter le nombre de paramètres entrainables, il est nécessaire d'effectuer des couches de MaxPooling visant à réduire la dimension de la feature map (la matrice de sortie d'une couche de convolution). Cette phase comprend alors inévitablement une perte d'information. Le MaxPooling vient faire en sorte de diminuer la dimension de la feature map tout en conservant les informations les plus importantes. Ci-dessous une figure vient expliquer cette opération

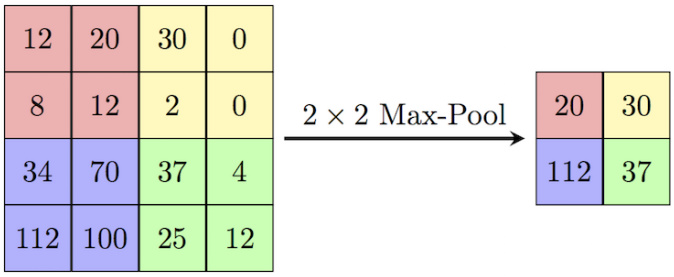

FIGURE 3 – MaxPooling2D 2×2

2.1.5 Upsampling et Concatenate

La phase d'upsampling peut être vue comme l'inverse de la phase de MaxPooling. Sauf qu'elle vient après la phase de MaxPooling ce qui fait qu'il lui est impossible de pouvoir passer d'une matrice 2×2 à 4×4 en retrouvant l'information perdue lors de la phase de MaxPooling. C'est précisemment pour cela qu'ensuite vient directement la phase de concatenation, celle qui permet au réseau de retrouver l'information perdue lors du MaxPooling. On ne retrouve pas parfaitement l'information, mais le résultat est bien plus satisfaisant que de retrouver une matrice 4×4 avec des blocs de 2×2 dont tous les éléments sont identiques même si la phase de concaténation n'empêche pas toujours d'avoir des blocs homogènes. Ex : dans la figure 3, la phase d'Upsampling seule aurait donné la matrice suivante en partant de la matrice 2×2 :

$$\begin{pmatrix} 20 & 20 & 30 & 30 \\ 20 & 20 & 30 & 30 \\ 112 & 112 & 37 & 37 \\ 112 & 112 & 37 & 37 \end{pmatrix}$$

2.2 Phase de data preprocessing

De manière générale, en data science, les données n'arrivent jamais sous une forme directement exploitable par le data scientist. Le Deep Learning ne fait pas exception. En réalité, le plus gros du travail consiste à directement travailler sur les données : forme, composition etc.. Lorsque le jeu de données est arrivé du CNES, je me suis par exemple apercu que certaines images n'avaient pas de masque. Elles ne sont donc plus du tout exploitables. En effet, lors de la phase d'entrainement le réseau ne saura pas vers quelles valeurs se tourner pour apprendre des patterns de cette image, et lors de la phase de prédiction, nous n'aurons aucun moyen de vérifier rigoureusement ce que fait le réseau dessus. Ceci implique donc de faire un filtrage des données. Ce n'est qu'un exemple parmi d'autres. Toute la phase de data preprocessing à été réalisé en Python grâce à ses librairies particulièrement pratiques comme numpy.

2.2.1 Analyse des données

Tant que faire se peut, il est toujours conseillé de visualiser les données sur lesquelles on souhaite travailler par la suite. Cela permettra ensuite de faire de meilleurs choix sur la nature et les caractéristiques du jeu de données que l'on présente au réseau. Après avoir filtré les images sans masques on aimerait connaitre les caractéristiques des images que l'on utilisera pour le réseau, surtout en termes de dimension. Cela à en effet une importance dans une perspective de data augmentation (image cropping, flipping ..). La figure suivante montre la grande hétérogénéité des dimensions des images :

```
[ronzie@ibo ~]$ cat /space/ronzie/CNES_DL/data/data.csv
file_name,height,width
D17_033980_1956_XI_15N262W,7784,5269
G04_019707_2071_XI_27N114W,35705,11144
D17_034014_1842_XI_04N109W,6567,4702
D22_035623_1905_XI_10N118W,9174,6573
D16_033368_1971_XN_17N113W,8973,5326
D15_033078_1953_XI_15N115W,10343,6808
D21_035473_1885_XI_08N342W,81786,13896
B19_017038_1791_XI_00S322W,5193,3246
G11_022533_1697_XN_10S223W,35835,8059
G14_023571_1836_XI_03N124W,5502,5809
B19_017003_1834_XI_03N087W,27423,8560
G11_022536_1831_XI_03N306W,5191,3278
G01_018589_1886_XI_08N189W,2766,3119
B03_010635_1780_XI_02S113W,6720,5831
B22_018192_1986_XI_18N150W,5176,3642
D14_032577_1918_XI_11N116W,9143,6494
B11_014050_1726_XI_07S108W,5497,5745
G05_020052_2206_XI_40N174W,5157,4667
F05_037944_1827_XI_02N121W,9159,6240
```

FIGURE 4 – Dimensions De Mars

On peut s'apercevoir sur les deux dernières colonnes que certaines images dépassent les 30000×10000 pixels tandis que d'autres sont en dessous des 5000×5000. Si le jeu de données n'est composé que d'un peu moins de 500 données, en revanche elles sont suffisamment grande pour permettre d'en faire des dizaines voir des centaines à partir d'une image.

Par ailleurs, on s'intéresse à la détection d'impact, on aimerait alors savoir comment sont représentés ces impacts sur le jeu de données. Ci dessous figure une image taguée d'un impact :

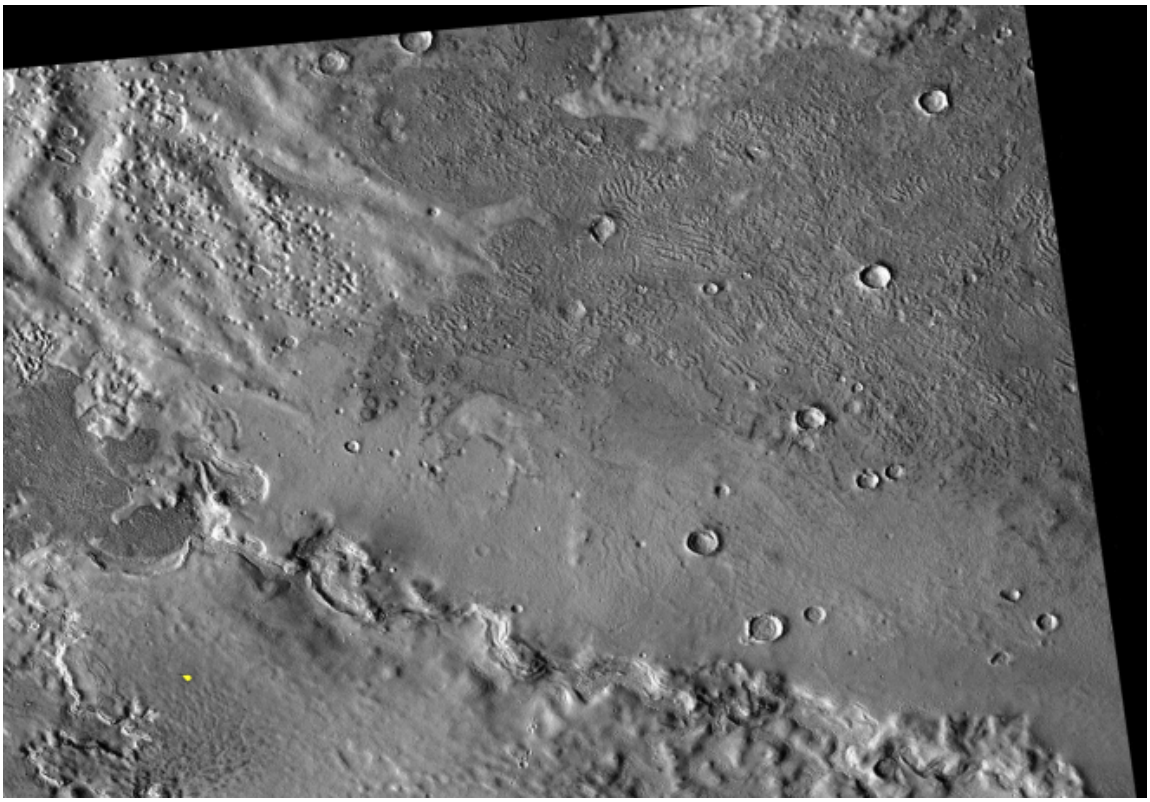

FIGURE 5 – Impact sur Mars

Sur cette image en bas à gauche, on peut y voir un impact en jaune. Après analyse de toutes les images à notre disposition, cette image est révélatrice du jeu de donnée : en moyenne, un impact représente 0.48% de l'image et il ya peu d'impact par image. Cela nous pousse à prendre des mesures vis à vis des données. En effet, si on donne des images comme tel au réseau, il apprendra que dans plus 99.5% des cas, dire que tous les pixels n'appartiennent pas à un impact est une bonne réponse. Ce qui lui attribuera une bonne note en terme de précision mais en réalité c'est tout l'inverse. On voit clairement dans ce cas que l'on doit agir sur les données.

2.2.2 Equilibrage de la base d'apprentissage

La solution consiste à équilibrer la base d'apprentissage en utilisant le cropping d'image. Puisque ces images ne sont pas aptes à être utilisées comme telles (sans parler des contours noirs qui rajouteraient du bruit lors de la prédiction), il nous faut faire des zoom sur des endroits intéressants. Par zoom, on entend découper des zones interessantes de l'image. Une première approche serait de découper des images de 128 par 128 centrées autour d'un impact de sorte à ce que l'impact prenne une proportion plus importante que précédemment. La section suivante montre les résultats obtenus suite à différentes manière de cropper l'image.

3 Résultats

Ce stage comporte deux facettes : une sur la mise en place d'un code de deep learning et une sur une analyse de performance. Cette section est divisée en deux grandes parties qui les décrivent.

3.1 Méthode générale

Lorsqu'on entraine un réseau sur un jeu de données, il faut diviser ce jeu en trois parties : une première pour la phase d'apprentissage (train), une seconde pour la phase de validation et une dernière pour la phase de prédiction. En effet, il n'est pas rare qu'un réseau sur-appprenne le jeu d'apprentissage. Si donc on fait des prédictions sur ce même jeu, les résultats paraitront certes très bon mais à la venue d'un nouveau jeu de données les résultats peuvent avoir une nette différence. Ce phénomène est appelé l'overfitting, et pour pouvoir le détecter, on a besoin du jeu de validation (distinct donc du jeu d'apprentissage) Ce phénomène est illustré ci-dessous :

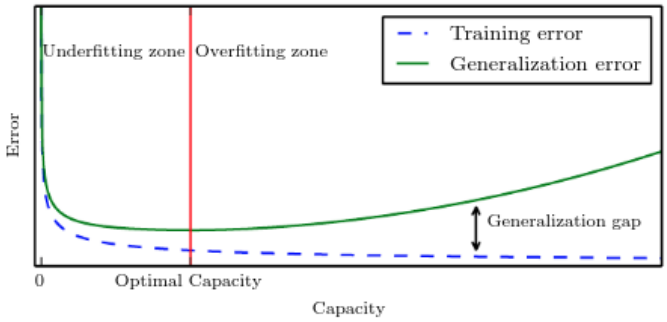

FIGURE 6 – Overfitting

La courbe verte montre l'erreur sur le jeu de validation (appelée validation loss dans la littérature), et la courbe en pointillés bleue montre l'erreur sur le jeu d'entrainement (training loss). L'axe des abscisses représente le nombre de fois où l'on passe sur le jeu de d'apprentissage (appelé epoch). On voit qu'à partir d'un moment, symbolisé par la droite verticale rouge, le réseau apprend trop de son jeu d'entrainement et commence à avoir de moins bon résultats sur le jeu de validation. Le but est de détecter le nombre d'epoch qui minimise l'erreur sur le jeu de validation. C'est là que les poids calculés par le réseaux sont optimaux.

3.2 Analyse des résultats du u-net

Comme dit en amont, le modèle choisi initialement était très simple, avec un réseau le plus simplifié possible. Petit à petit , afin d'améliorer les résultats, ce réseau a été rendu plus profond. Par ailleurs, en même temps que la profondeur du réseau à été agrandie, la manière de faire du cropping à été adaptée.

3.2.1 Cropping 1024×1024 autour d'un impact

En premier lieu nous avons donc choisi de rendre l'impact un peu plus significatif et donc de cropper l'image autour de celui-ci. Il faut cependant essayer de respecter le fait qu'il y a de nombreuses zones où il n'y a pas d'impact. ll a alors été décider de cropper une nouvelle fois cette image 1024×1024 en 64 images 128×128. Cela veut dire que le réseau sera confronté en moyenne qu'à une image sur 64 où il y a un impact. Voici les résultats visuels obtenus sur une prédiction de 5 images

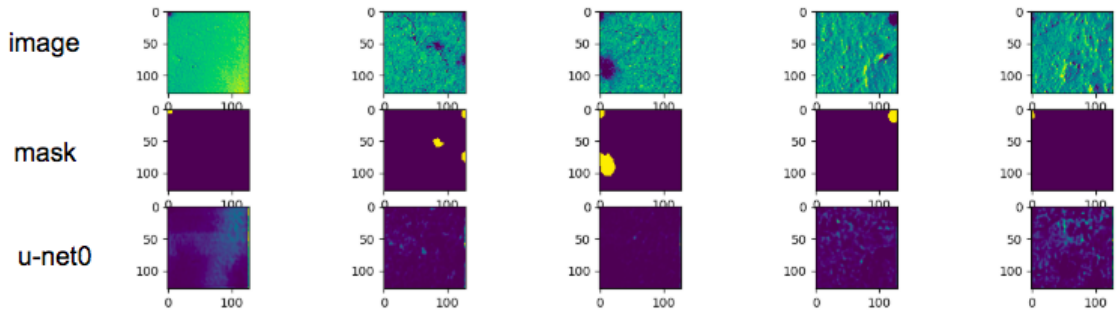

FIGURE 7 – U-net basique sur cropping 1024×1024 puis sur cropping 128×128

La première ligne correspond à l'image d'entrée, la seconde à son masque (la vraie réponse attendue) et enfin la dernière la prédiction faite par le premier réseau simple (unet0) présenté sur la figure 1. A l'oeil, on s'apercoit que le résultat est mauvais, en réalité, on n'échappe pas encore au cas qu'on souhaitait éviter lorsque le réseau apprend à dire qu'il n'y a pas d'impact tout le temps. Ici nous sommes allé cherché les cas où il y avait des impacts, mais le réseau s'en sort bien dans plus de 99% des cas avec une validation loss inférieure à 1%

3.2.2 Solution au problème

Pour régler le problème précédent, nous avons choisis de montrer au réseau 50% d'images 128×128 (issues du même cropping que précédemment) avec impact et 50% d'images 128×128 sans impact. Le résultat est alors bien meilleur, en gardant exactement le même réseau de neurones. Il est nommé unet1.0 par la suite car ses poids diffèrent du premier (unet0) étant donné qu'il a été soumis à un jeu de donnée mieux choisi.

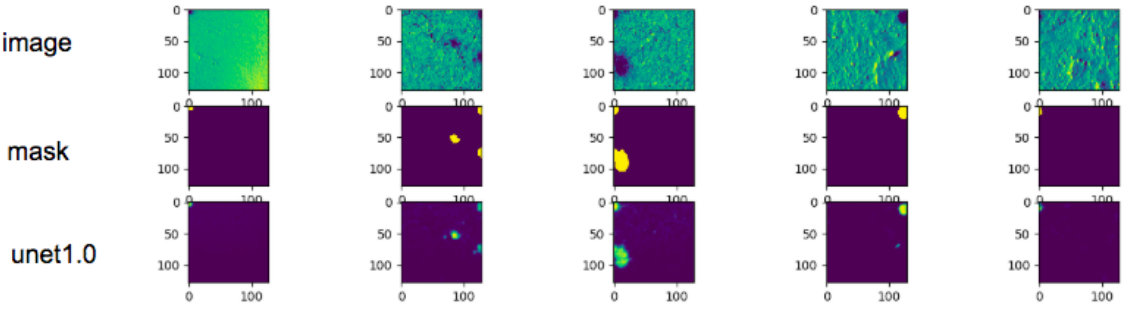

FIGURE 8 – U-net basique sur moitié d'images avec et sans impact

Le résultat visuel est satisfaisant. A ce niveau là aucune tentative de quantification de l'erreur commise n'a été faite. Elle sera faite plus tard, en calculant tout simplement la M.S.E (Mean Square Error) de la différence entre la prédiction de sortie et le masque associé (qui ne sont autres que des matrices réelles).

On s'est ensuite intéressé à ce que faisait le réseau sur les cratères, qui ne sont autres que de vieux impacts. Voici le résultat :

FIGURE 9 – Prédiction du réseau de base sur un cratère

La colonne de gauche montre qu'aussi, les masques sont parfois imparfaits. Rappelons que c'est un être humain qui a eu la charge de pointer tous les impacts, des erreurs sont donc présentes.

Pour le cratère, sans surprise le réseau confond les zones sombres avec un impact. Il faut rappeler que le réseau est très peu profond. Keras permet de connaitre le nombre de paramètre entrainable dans un modèle de réseau et dans notre cas, nous ne sommes seulement qu'à environ 500000 paramètres entrainables. Ce nombre est à mettre en comparaison avec des réseau type u-net plus élaborés qui dépassent facilement les 10 millions de paramètres entrainables. Ce qu'il faut en déduire, c'est que le réseau n'est pas assez profond pour pouvoir apprendre des pattern plus sophistiqués de l'images comme par exemple les petites vaguelettes au centre d'un cratère ou encore que dans la plupart des cas, un cratère présente une face sombre et une face brillante.

3.2.3 Amélioration du réseau

Le réseau a donc montré ses limites, ce qui était attendu. Nous avons rendu le réseau de neurones plus profond. Concrètement, pour se donner une idée de comment rendre plus profond le réseau, l'interview donnée en lien montre un u-net bien plus profond que le notre. La figure suivante montre la progression de la prédiction liée à l'augmentation de la profondeur du réseau de neurones. Les deux premières lignes concernent l'image et le masque associé. Les trois dernières lignes concernent, du haut vers le bas, la prédiction du réseau le moins profond au plus profond. On passe d'un réseau à 500000 paramètres entrainables (trainable parameters t.p) à 1500000. F.P désigne l'ajout de faux positifs. Ceux-ci ont été identifié grâce à des prédictions très érronées du u-net2.0 sur le jeu des données autre que les jeux de tests, d'entrainements et de validations, pour ne rien toucher à la disposition mis en place (voir section suivante).

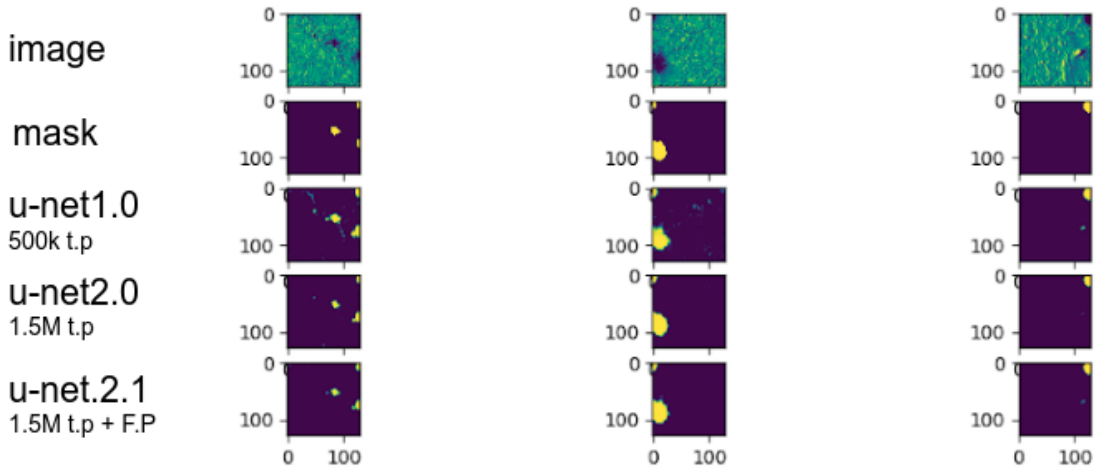

FIGURE 10 – Progression des réseaux sur la prédiction

Une analyse attentive montre que les prédictions perdent du bruit (détection d'impact érronée) à mesure que le u-net s'approfondit. C'est une bonne chose, et ce même si les contours de certains impacts deviennent moins bon, car l'objectif reste de donner à un humain les endroits où se situent les impacts

3.2.4 Traitement des cratères

Pour traiter le problème lié aux cratères, nous nous sommes rendus compte que le réseau méritait d'être confronté à plus d'images sur lesquels étaient présent des cratères. Car même en faisant gagner de la profondeur au réseau, celui ci, confronté à peu de cratère, n'aurait pas l'opportunité de profiter pleinement de sa capacité pour détecter les patterns intimement liés aux cratères comme les vaguelettes à l'intérieur.

La méthode qui suit à donc été adoptée : Sur la version précédente, le jeu de données à donc été divisé en trois parties : sur 17000 images de 128×128, 10000 étaient pour le jeu d'entrainement, 4000 pour la validation et 3000 pour la prédiction. Sur les 10000 du jeu d'entrainement, seules 700 avaient été retenues car seulement 350 comportaient des impacts et on voulait avoir un jeu d'entrainement composé de moitié d'images avec impact et moitié sans impact. Ce qui fait qu'au total, plus de 9000 images n'avaient pas été utilisées. Sur ces 9000, on a fait des prédictions et récupéré les gros faux positifs : c'est-à-dire les images où le réseau prédisait un impact alors qu'il n'y en avait pas, et donc principalement les cratères (c'étaient là que le réseau avait les plus grossières erreurs). Et ces gros faux positifs ont été réinjecté dans le jeu d'entrainement ce qui faisait au passage passer de 700 images à 1200 environ.

Voici deux tableaux récapitulatifs de cette méthode :

Base de données : 17000 images		
Train : 10000 images	Val : 4000 images	Test : 3000 images

Train : 10000 images	
Utilisées : 700 images	Restantes : 9300 images

Ce sont donc dans ces 9300 images restantes qu'une prédiction préalable à été réalisée pour en tirer les images donnant les faux positifs recherchés. Il est à noter ici que tout à été mis en oeuvre pour ne pas toucher le jeu de validation et de prédiction de base de manière à avoir une comparaison objective et non biaisée.

En ce qui concerne les cratères , voici la progression observée :

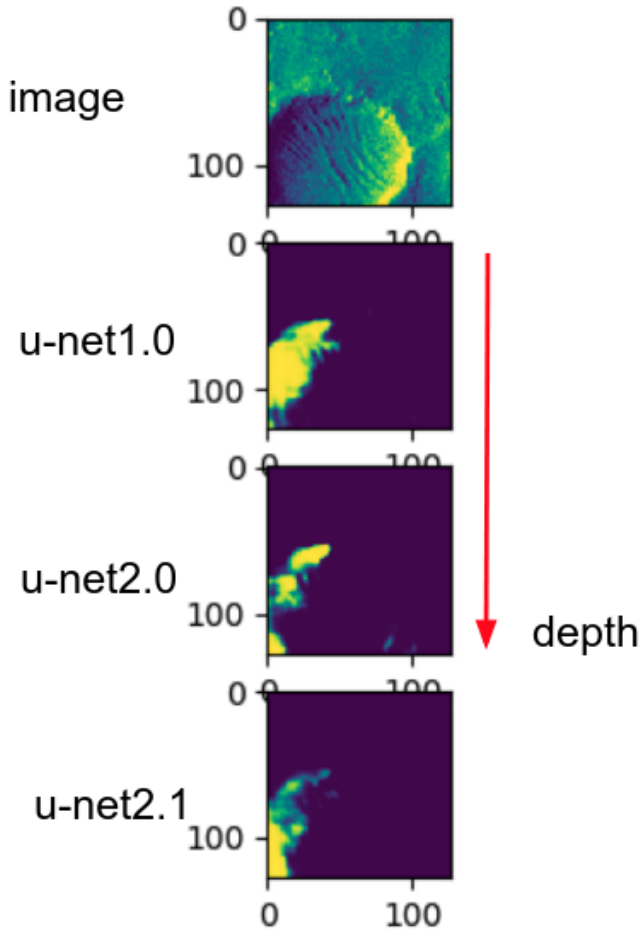

FIGURE 11 – Progression de la prédiction sur un cratère

On voit que le réseau s'améliore à mesure qu'il gagne en profondeur. Sauf qu'à la différence de la figure 10, les deux dernières lignes concernent le réseau le plus profond mais auxquels on a présenté des jeux de données différents donc.

Après cette dernière étape d'amélioration, il était intéressant de pouvoir quantifier la qualité de la prédiction et de savoir où on se situe en termes de validation loss afin d'avoir un repère plus précis pour d'eventuels améliorations/ajustements futurs. Ci dessous figurent les courbes de training loss (en bleu) et validation loss (en rouge) pour le réseau profond confrontés à ses propres faux positifs :

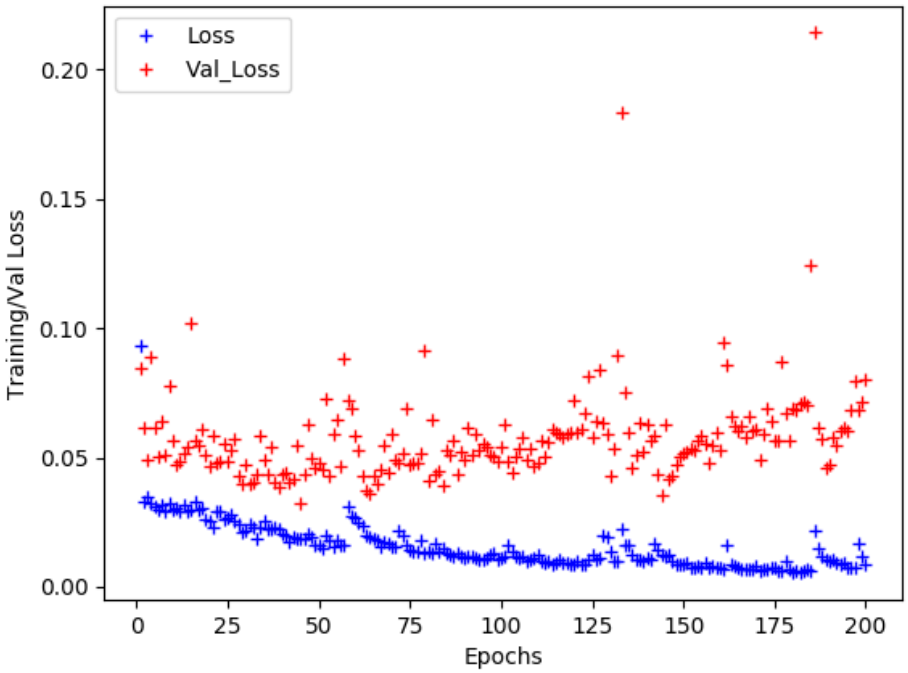

FIGURE 12 – Training/Validation loss

Nous voyons donc le phénomène décrit plus haut section 3.1, le réseau à overfitté vers l'epoch 40. Donc concrètement cela veut dire que les résultats précédent auraient pu être meilleurs si la phase d'entrainement avait été arrété vers l'epoch 40.

3.2.5 Défauts et correction des réseaux précédents

Jusqu'ici seulement un détecteur d'ombre plus ou moins sophistiqué à été fait. Il serait souhaitable d'avoir un réseau qui apprenne un peu plus de la topologie spatiale d'un impact (forme ronde par exemple). les défauts pointés par le CNES sur nos entrainements sont :

— Random Crop : les impacts se trouvaient toujours en bordure de fenêtre ce qui pouvaient amener le réseau à apprendre que les impacts se trouvent majoritairement sur les bordures de l'image

— Batch : Des batch pas toujours équilibré entre images avec impact et images sans impact

Les solutions sont donc de faire un random crop plus intelligent en selectionnant une zone aléatoire autour de l'impact de manière à ce que l'impact se retrouve à plusieurs endroits sur l'image et pas juste en bordure comme avant. De la sorte on peut utiliser cette technique plusieurs fois autour du même impact et cela constituera plusieurs images à partir d'un seul impact. Cela est dans notre cas plus efficace que du flipping (effet miroir sur une image) ou des rotations. Par ailleurs, le réseau est toujours relativement peu profond. Un u-net plus profond ayant 5 millions de paramètres entrainables à été choisi. L'architecture finale se présente page suivante, où seule la partie gauche a été représentée. La partie droite non présente correspond à la remontée avec la phese d'upsampling comme dans l'architecture de la figure 1.

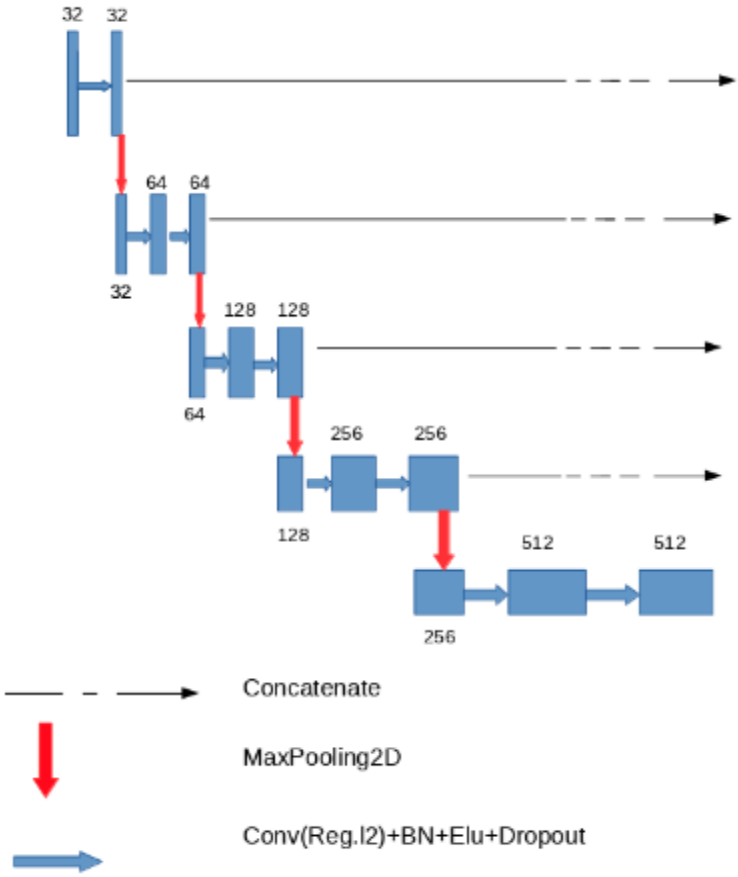

FIGURE 13 – Architecture U-net3.0

Du Dropout et de la régularisation l2 ont été rajouté. Lorsque un réseau de neuronne devient profond, les risques d'overfitting sont grands. Ces deux couches sont rajouttées pour limiter cet effet. Le learning rate a également été abaissé au fur et à mesure des épochs. Par ailleurs, nous avons constaté que redonner des faux positifs amène de grands progrès dans l'apprentissage Voici les résultats visuels :

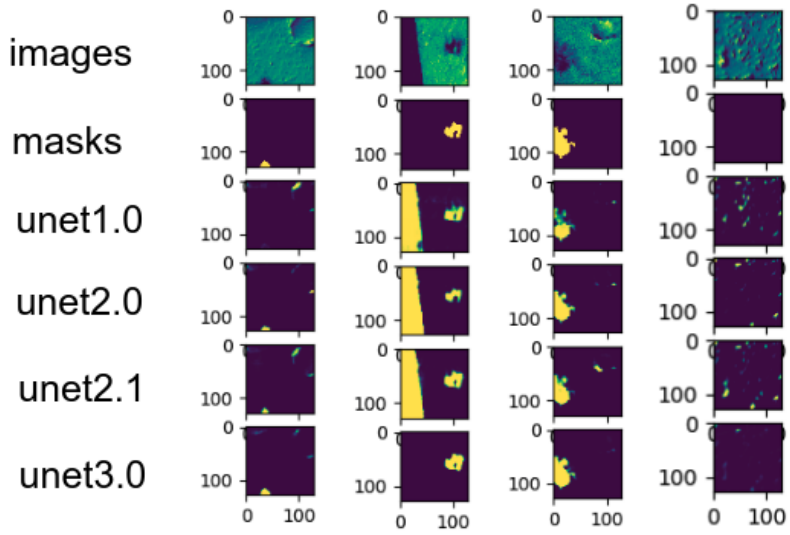

FIGURE 14 – Predictions des différents réseaux

La différence se voit très clairement sur ces quatre prédictions. Sur la première colonne on peut voir que le réseau détecte mieux l'impact tout en commettant moins d'erreur sur le cratère en haut à droite. La deuxième colonne est particulièrement satisfaisante. Là où comme dit précédemment les anciens réseaux n'avaient appris qu'à détecter de l'ombre, le dernier a appri la topologie spatiale d'un impact et sait donc qu'une bordure noire ne peut être un impact (comme on peut le voir sur la figure 5 les images sont tronquées et donc il y a des bordures noires qui ne correpondent pas à l'image en elle-même) Cela se vérifie alors sur la troisème prédiction (colonne 3) où le dernier réseau n'a pas du tout confondu le cratère avec un impact contrairement aux précédents réseaux. La dernière colonne montre qu'ainsi le nouveau Unet renvoit des prédictions moins bruitées en général et en particulier ici sur des surfaces rocailleuses, contrairements aux précédents.

Ces résultats qualitatifs satisfaisant sont appuyés par un test quantitatif plus large qui parcoure l'ensemble du jeu de test (et pas seulement quatre images comme sur la figure 14). Pour chacun de ces quatre réseaux, la moyenne des MSE (Mean Squared Error) entre les prédictions et les masks associés a été calculée. Voici les résultats en pourcentage :

Unet1.0	Unet2.0	Unet2.1	Unet3.0
13.716	13.706	13.63	0.524

FIGURE 15 – MSE(%) des différents réseaux étudiés

La nette progression visuelle se confirme donc par les tests quantitatifs. Toutefois, ce scoring là n'est pas à prendre comme une echelle d'évaluation parfaite. En effet, le but recherché par ce réseau est de pointer les impacts en ayant le moins de faux positifs possible. Cependant, une importance moindre est apportée à la qualitée du pointage. Si donc un réseau pointe un impact et à des contours flous voir manquants cela n'est pas dommageable. Le but étant que derrière un humain valide ou invalide la prédiction.

3.3 Conclusion sur les résultats du u-net

Cette partie du stage comportait comme principale difficulté la distinction cratère/impact tout en ne tombant pas dans le piège de ne rien détecter du tout. Ne rien détecter du tout donne en en moyenne un score qui peut paraitre très bon, mais en réalité très mauvais.

L'architecture du U-net a montré sa capacité à résoudre ces deux points à la fois avec un très bon résultat final. Ce n'est toutefois pas une boite noire magique. Cette étude montre en effet qu'il faut l'ajuster soigneusement au problème concerné avec sa profondeur entre autre mais aussi qu'il faut lui donner un jeu de donnée adéquat. La phase de data preprocessing joue un rôle centrale dans les résultats finaux notamment sur la réinjection de faux positifs.

Finalement le unet3.0 permet de détecter précisemment les impacts avec une MSE proche de 0.5% et arrive à faire la distinction entre cratère et impact.

3.4 Analyse des performances

Le CNES était curieux de savoir quels type de speed up étaient possibles sur du multi-gpu. Comme dit en amont dans l'introduction section 1.2.3, j'ai accès à plusieurs architectures matérielles dont Ouessant à l'Idris qui est une architecture externe au CERFACS. Nous allons analyser le speed-up que l'on a observé sur 1 - 2 - 3 et 4 GPUs au sein d'un même noeud sur Ouessant, puis au sein de 2 et 4 noeuds.

3.4.1 Présentation rapide de Ouessant

Ouessant se présente comme une plateforme de calculs accessible via ssh. En réalité, elle s'utilise de manière très similaire à PlaFRIM. Il faut charger des modules via la commande **$module load**. Ouessant s'utilise via la soumission de job sous la forme de script batch. Dans ces batch peuvent être inclus des lignes de directives comme par exemple

#BSUB -gpu "num=4 :mode=exclusive_process :mps=no :j_exclusive=yes".

Ces directives permettent de faciliter la selection des GPUs et noeuds sur lesquels on souhaite travailler. Voici un exemple de script batch :

```
# Nom du job
#BSUB -J monJob_test
# Ficher output et error
#BSUB -e %J.err
#BSUB -o %J.out
# Nombre de tache MPI
#BSUB -n 4
# Binding
#BSUB -a p8aff(1,8,1,balance)
# Nombre de gpu
#BSUB -gpu "num=4:mode=exclusive_process:mps=no:j_exclusive=yes"
# Nombre de tache MPI par noeud
#BSUB -R "span[ptile=4]"
# Duree maximum du travail
#BSUB -W 01:00

$PYTHON model.py > u-net.log
```

La gestion des jobs sur ouessant se fait au travers de queues : lorsqu'un job est lancé, il faut qu'il attende qu'un noeud soit libre.

3.4.2 Entrainer le réseau sur plusieurs GPUs sur un noeud

Keras facilite l'utilisation du parallélisme pour pouvoir utiliser plusieurs GPUs. Il suffit de rajouter/modifier quelques lignes de codes pour passer au modèle multi-gpu, keras fournissant une implémentation haut niveau. Lorsqu'on lance un entrainement sur un GPU, chaque epoch se fait au travers de plusieurs batch (à ne pas confondre avec le batch de soumission pour ouessant). Le batch dans une epoch est un échantillon de données qui va permettre au réseau de faire sa descente de gradient stochastique (adapter les poids). Par exemple, si la taille du batch est de 32 et qu'un jeu d'entrainement comporte 1600 données, le réseau devra faire 50 descentes de gradient stochastiques par epoch. Si maintenant on passe sur deux GPUs, et qu'on met la taille du batch à 64, celui ci sera divisé par deux et chaque GPUs fera une descente de gradient stochastique sur son batch de taille 32. En parallèle, chacun des deux gradient effectuera donc 25 descentes de gradient, deux fois moins donc ! TensorFlow, qui est le backend sur lequel s'appuie Keras, fournit une explication plus détaillée sur les détails de ce parallélisme.

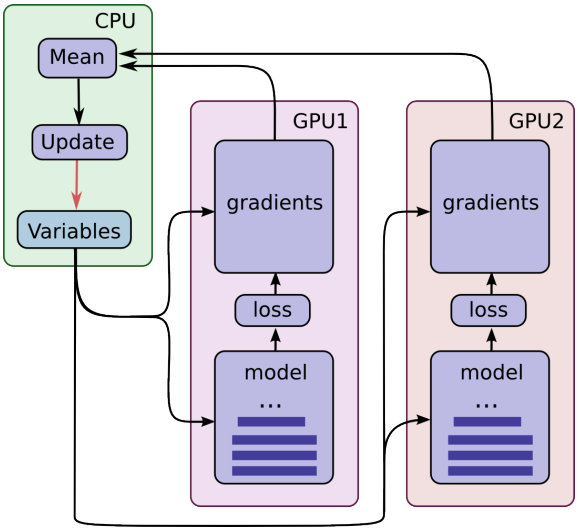

FIGURE 16 – Parallelisme Mutli-GPUs

La grande force de ce parallélisme repose sur l'association CPU-GPUs. En effet, cette association va ainsi permettre de supprimer les communications inter-GPUs. Chaque GPU communiquera le résultat de sa descente de gradient stochastique au CPU du même noeud, qui va ensuite calculer rapidement la moyenne pour la retransmettre ensuite aux GPUs pour qu'ils puissent continuer l'apprentissage sur le batch suivant. La partie de communication est donc vraiment réduite si on passe de 2 GPUs à 4, 6, 10 etc.. TensorFlow promet un speed up linéaire , c'est-à-dire le speed-up idéal. C'est une chose rare qu'on a bien évidemment voulu vérifié. Voici le résultat de performance :

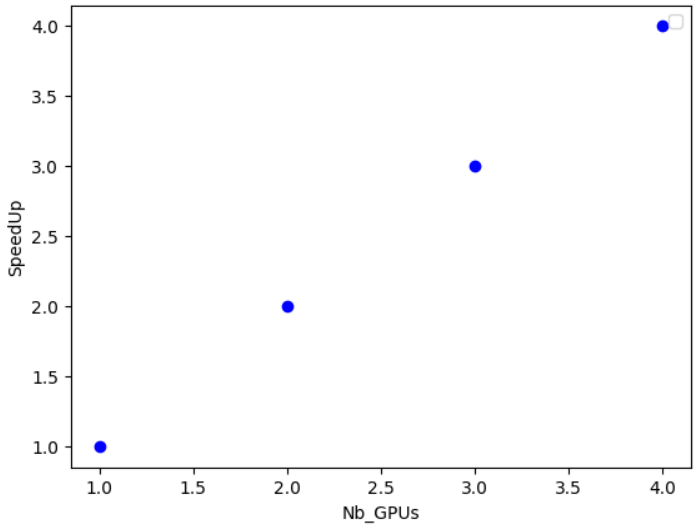

FIGURE 17 – Speed Up multi GPUs

Le speed-up est bien rigoureusement linéaire en nombre de GPUs où les temps d'exécutions se divisent par le nombre de GPUs sélectionnés. Ceci est vrai dans le cas où les GPUs sont identiques. Car lors de la communication du n-ième batch, si un GPU termine avant , le CPU devra attendre que les autres terminent pour pouvoir ensuite faire la moyenne des poids recus. Nous n'avons pu tester cela "que" sur 4 GPUs car Ouessant est composé de 12 noeud possédant chacun 4 GPUs (Nvidia P100), ce qui est déjà appréciable.

3.4.3 Entrainer le réseau sur plusieurs GPUs multi-noeuds

Keras ne suffit plus pour faire du Deep Learning sur plusieurs noeuds de calculs. La librairie Horovod développée récemment (2017-2018) par Uber permet de le faire à une echelle encore assez réduite. En effet, cette technologie récente présente un défaut qui se retrouve de manière généralisée lorsqu'on souhaite dépasser un certain nombre de GPUs (3-4 GPUs). Horovod passe par la bibliothèque MPI et on doit lancer autant de processus MPI que de GPUs souhaités. Voici un exemple de commande dans le script bash pour sélectionner 4GPUs :

```
mpirun -np 4 -x LD_LIBRARY_PATH -bind-to none -map-by slot -x
    ↪ NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH --mca orte_tmpdir_base
    ↪ "/tmp" python u-net_multi_gpu.py > unet4gpu1node.log
```

Il y a plusieurs moyens d'entrainer le réseau en parallèle. Un premier moyen était donc de faire du parallélisme sur les données (data parallelism) en divisant le batch par le nombre de GPUs sélectionnés comme précédemment. Un autre moyen peut être de lancer le modèle avec toutes les données sur plusieurs GPUs et à la fin de chaque batch, les poids sont partagés, une moyenne est faite entre tous les différents poids. Le réseau converge donc plus vite car il exécute plusieurs descente de gradient en parallèle et donc plusieurs epochs en parallèle. Les epochs diffèrent d'un GPU à l'autre du fait que chaque GPU exécute les batch dans un ordre différent. Voici un schéma explicatif de ce que fait Horovod pour accélerer l'entrainement :

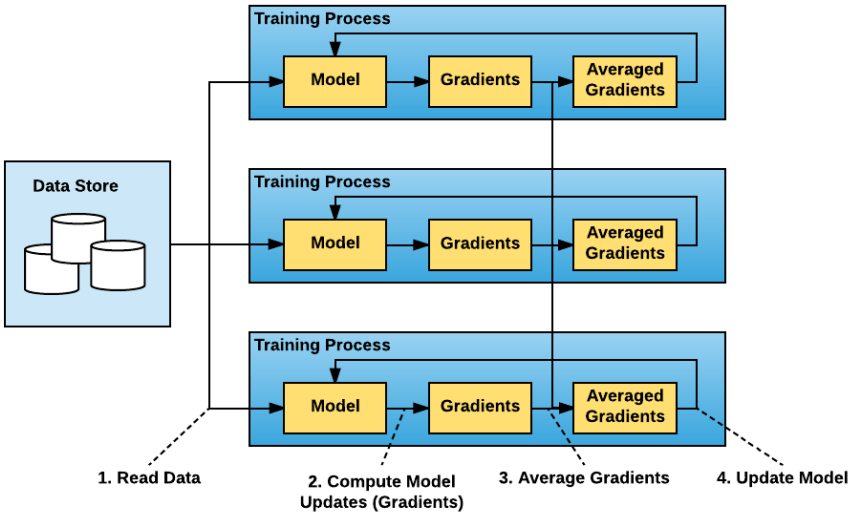

FIGURE 18 – Fonctionnement d'Horovod

Chaque GPU va donc recevoir l'ensemble des données lors de la phase 1 'Read Data' et exécuter le modèle dessus. Chaque mise à jour des poids après passage sur les batch sera faite d'abord localement lors de la phase 2 'Compute Model Updates' puis distribué lors de la phase 3 'Average Gradients'. Et chaque GPU récupère la mise à jour des poids faite par la moyenne des poids calculés localement lors de la phase 4 'Update Model' Il y a donc des coûts de communication non négligeables à prévoir contrairement à précédemment.

Voici les résultats obtenus sur un noeud de calcul avec 1, 2 et 4 GPUs :

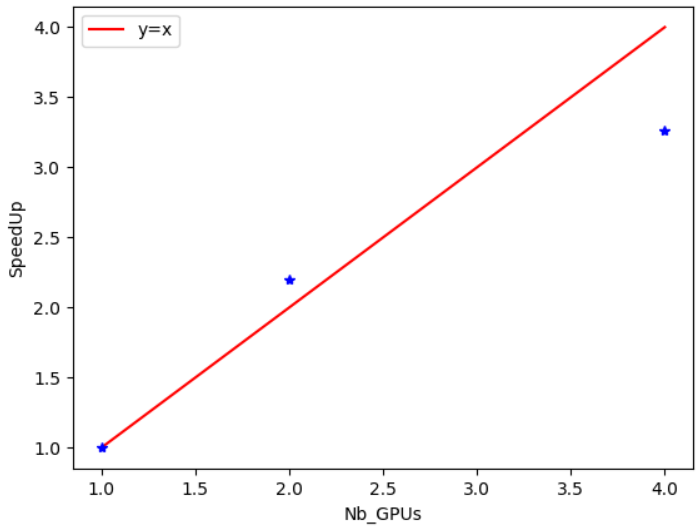

FIGURE 19 – Speed Up Horovod

Pour obtenir ces résultats, il faut se fixer une valeure de convergence en dessous de laquelle on stoppe l'entrainement. Car de base l'entrainement ne termine que lorsqu'il a effectué toutes les epochs qu'on lui a demandé de faire. Ainsi les temps d'exécutions ne varieront pas que ce soit sur 2, 4 ou 8 GPUs et plus. Ici, l'entrainement est stoppé lorsque la validation loss descend au dessous de 0.6. Pour information, il faut plus de 40 minutes à un seul GPU (Tesla P100 - 16Go RAM) pour y arriver. C'est une valeur choisie de manière à être certains que jusqu'à cette valeur la validation loss soit strictement décroissante. Cela permet de mieux mesurer l'effet du parallélisme induit par Horovod.

Il peut paraitre surprenant d'avoir un Speed Up supérieur à 2 pour 2 GPUs mais en réalité, cela correspond au fait que l'exécution d'un entrainement n'est pas déterministe d'où le terme stochastique dans le calcul des poids avec la 'descente de gradient stochastique'. Ainsi, d'une exécution à l'autre les résultats d'exécution et la convergence peuvent varier légèrement. Pour mieux faire cette étude, il aurait été préférable de répéter plusieurs fois les calculs afin d'obtenir une courbe moyenne. Le manque de temps (fin de stage) et la durée des exécutions des modèles (plusieurs dizaines de minutes) n'ont pas permis de le faire.

On confirme par ailleurs que des coûts de communications se font ressentir et ce dès l'utilisation de 4 GPUs avec un speed up qui semble concave. Comme dit en amont, il a été impossible de voir le comportement pour plus de 4 GPUs avec les limites d'Horovod levant une exception MPI après 5 minutes d'exécution normale (de plus sur un noeud on ne peut pas avoir plus de 4 GPUs)

On s'est intéressé aussi aux éventuels sur-coûts de communication en sélectionnant plusieurs GPUs sur plusieurs noeuds. L'étude à donc été limitée à 4 GPUs seulement. Voici les résultats des temps d'exécutions où on sélectionné 4 GPUs sur 1 - 2 et 4 noeuds donc 4 GPUs sur un noeud, 2GPU par noeud sur 2 noeuds et 1 GPU par noeud sur 4 noeuds.

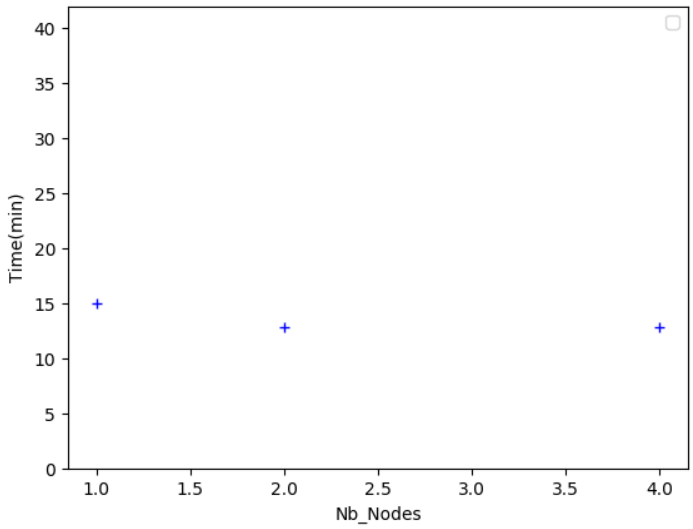

FIGURE 20 – Temps d'exécution de 4 GPUs sur 1,2 et 4 noeuds

Il est difficile de statuer sur les véritables surcoûts de communications puisqu'il n'a pas été possible d'aller plus loin que 4 GPUs mais ce graphe laisse des raisons de penser qu'ils paraissent négligeables à côtés des coûts de communications inter-GPUs. De même il peut paraitre surprenant que sur un noeud l'entrainement soit plus lent, mais c'est encore une fois du à l'aspect stochastique de l'entrainement.

Une conclusion qui peut être faite sur Horovod est qu'a priori la solution de parallélisme offerte par Keras parait plus performante car elle suit un Speed Up rigoureusement linéaire en nombre de GPU. Toutefois celle-ci s'applique au cas d'un seul noeud. Horovod peut donc s'avérer plus performant en selectionnant plus de GPUs qu'il n'y en a de disponible sur un noeud. Toutefois Horovod présente encore un défaut majeur (limitation en nombre de GPU) qui pourrait cependant être corrigé rapidement.

3.4.4 Travail sur les Tensor Cores

Le CERFACS est équipé d'une carte graphique Nvidia Tesla V100. Celle-ci à la particularité de posséder des tensor cores, spécialement dédié au Deep Learning. Ces Tensor Cores permettent de faire des multiplications de matrices de tailles 4×4 plus rapidement. Pour cela, chacun des 640 tensor cores disponibles sur la V100 fait le calcul suivant : $D = A \times B + C$ avec C une matrice en simple précision (float32) ou demi précision et A et B deux matrices en demi précision. Voici un schéma explicatif :

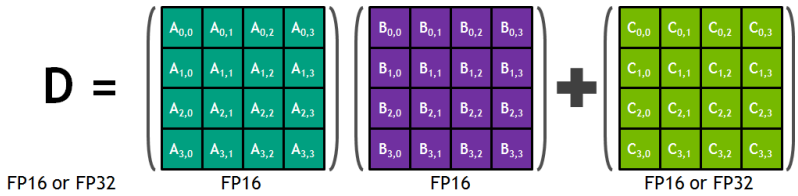

FIGURE 21 – calcul sur des matrices 4×4 avec les tensor cores

D'où l'anglicisme 'mixed precision' pour cette méthode de calcul dans un apprentissage.

Nvidia annonce sur sa documentation [4] un speed up de 8 (maximal) par rapport à un entrainement normal en simple précision. Cette partie du stage a été la plus fastidieuse. Sur cette documentation Nvidia donne des étapes à suivre sur la partie Frameworks. Beaucoup d'effort ont été fait pour essayer d'activer les Tensor Cores avec Keras, en vain. Grâce aux relations d'Isabelle D'Ast j'ai pu entrer en contact avec un ingénieur d'Nvidia pour avoir plus d'information. Tout cela m'a mené à changer de Framework en passant par TensorFlow et aussi PyTorch. PyTorch semble en effet plus simple d'utilisation concernant la demi précision.

Sur la page d'Nvidia, dans la section 5.5 avec PyTorch, une ligne concernant la batch normalization suggère de changer la fonction classique par BatchNormalization2dFP16 qui serait spécialement dédiée à l'entrainement en demi précision. Après avoir récupéré le container que suggère Nvidia, il a été constaté avec des ingénieurs d'Nvidia que finalement ce tuto est dépassé concernant le point sur la BatchNormalization. Nous avons récupéré des tests simples proposés par Nvidia pour tester les tensor cores, au final le speed up n'est que de 1.4 ce qui correspond seulement au passage de la simple à la demi précision, mais malheureusement pas à l'activation des tensor cores.

Un certain temps non négligeable a été accordé au sujet des tensor cores. Après avoir essayé de modifier les sources de Keras, d'implémenter la totalité du réseau 'from scratch' en TensorFlow, et de même en PyTorch, il a été impossible d'activer les tensor cores. Même un test venant d'Nvidia n'a rien changé. Ceci dit, ce travail n'est pas vain de tout résultat. En effet, comme le mentionne Nvidia au début de sa documentation, le passage à la demi précision permet de réduire la mémoire nécéssaire pour faire tourner l'entrainement. Ceci à été par contre vérifié comme le montre les deux figures qui suivent :

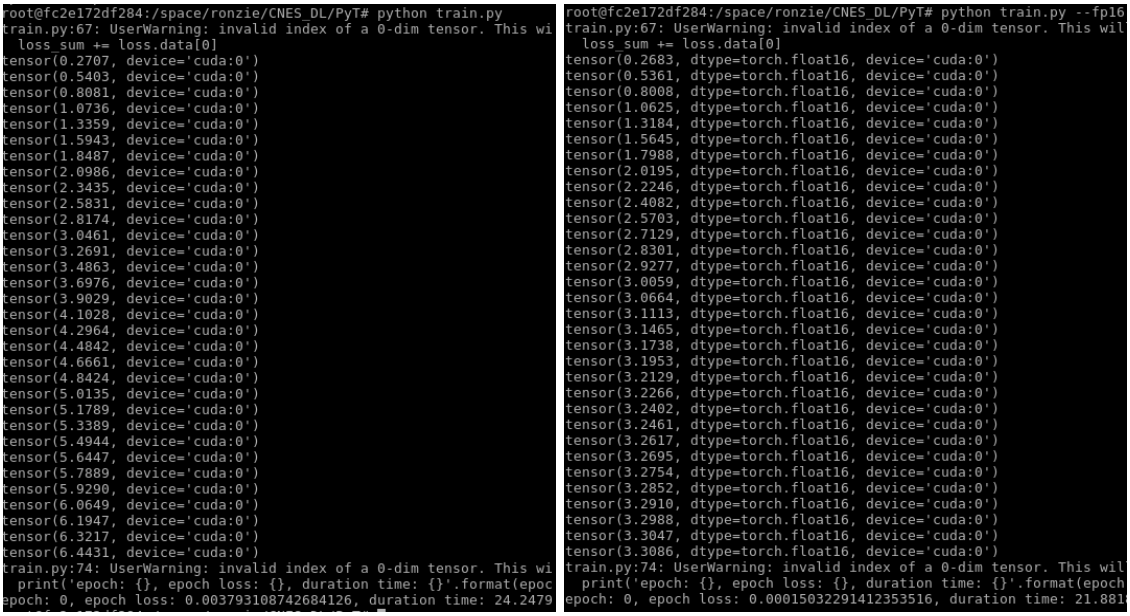

FIGURE 22 – Exécution d'une epoch en FP32 et FP16

4. https ://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html

Chaque ligne correspond à un passage sur un batch de taille 32 ici. Il y a 32 lignes, et 1000 images pour l'entrainements donc on a bien 32 passages de batch de 32 images pour couvrir le jeu d'entrainement sur une epoch. (31 passages avec des batch de 32 images et une dernière de 8 images). Sont affichés à chaque ligne les tenseurs de sortie avec comme premier paramètre la mémoire utilisée en GigaOctet. On voit qu'on divise par plus d'un facteur 2 la mémoire utilisée. Ceci peut s'avérer précieux en deep learning où l'on a pas souvent besoin d'une grande précision. En effet, cela permet d'entrainer des réseaux plus gros. Par exemple, sur la carte graphique Tesla V100, il est possible d'entrainer le réseau en demi précision pour des tailles d'images de 256 par 256 avec une taille de batch de 64 ce qui n'est pas possible en simple précision où l'on excède les 16GO de disponible.

Conclusion

Les réseaux convolutifs ont montré leur performance pour la segmentation d'image et c'est d'autant plus le cas avec les réseaux de types U-net. Malgré leur aspect boite noire, il reste essentiel d'en comprendre les principes. Cela permet en effet d'adapter la phase très imortante de data preprocessing en fonction. Il a été ainsi montré que plus la qualité de ce travail sur les données en amont est grande, meilleure est l'apprentissage et donc la prédiction d'un réseau. Dans le cas présent, étant donné la rareté d'un impact sur une image, il fallait essayer de respecter cette proportion tout en évitant au réseau de tomber dans le piège de ne pas reconnaitre les impact tout en ayant une très bonne MSE. L'ajout de faux positifs a aussi beaucoup joué sur la qualité de cet apprentissage. Une fois cela compris, jouer sur la profondeur du réseau et les hyperparamètres (learning rate, Dropout etc..) permet d'optimiser le réseau. A l'arrivée, un réseau capable de reconnaitre un impact avec une grande précision tout en distinguant impact et cratère (vieil impact) a été réalisé.

Néanmoins, le Deep Learning dont tout particulièrement les réseaux convolutifs sont très gourmands en ressources de calculs. Il est impensable d'essayer de faire tourner le même type de réseau sur une machine personnelle dépourvue d'une carte graphique récente. Même étant équipé d'un matériel puissant comme au CERFACS, ces réseaux peuvent prendre des heures, des jours voir des semaines pour tourner. Ce qui suscite un intérêt concernant la performance de ces réseaux notamment au CNES. Keras permet de paralléliser l'apprentissage sur un noeud de calcul de manière simple et optimale avec un speed up linéaire en nombre de GPUs. Horovod parait être une solution intéressante pour le cas du multi noeud. Cette bibliothèque s'appuie sur MPI. Bien que prometteuse, elle comporte un défaut majeur empéchant l'utilisateur de dépasser 4 GPUs quand celui-ci à la chance de pouvoir accéder à plus, ce qui fut mon cas. Enfin, Nvidia propose depuis peu des cartes graphiques équipées de tensor cores censés accélérer l'entrainement d'un réseau en demi précision avec un speed up allant jusqu'à 8, mais il n'a pas pu être possible d'activer ces tensor cores pour la Tesla V100. Le code de ce stage se trouve à cette adresse [5]

5. https ://github.com/RONRON2904/DataScience/tree/master/DeepLearningOnMars

Références

[1] Ronneberger Olaf, Fischer Philipp et Brox Thomas. “U-net : Convolutional networks for biomedical image segmentation”. In : *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, p. 234–241.