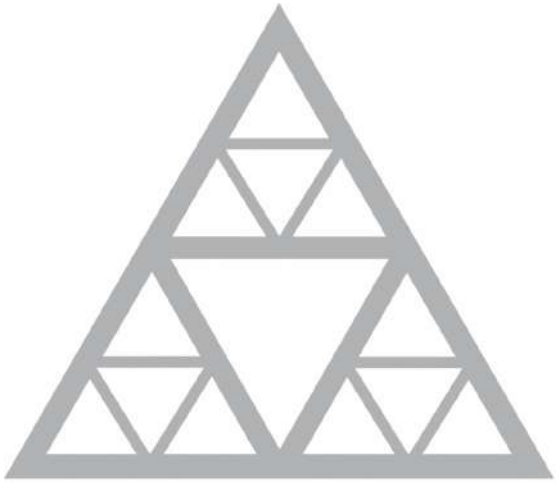
PROJET TDLOG

Génération de cartes d'élévation et interface utilisateur



École des Ponts

ParisTech

FAUDUET Alex
RIOU Auriane
VAN DEN BERGH Candice
VINCENT Théo

Tuteur : FOURRIER Clémentine
Professeur : CLERC Xavier

Novembre 2019 — Janvier 2020

Table des matières

Table des figures

Objectifs

Dans le cadre du projet de notre cours de techniques de développement logiciel, nous avons choisi de nous intéresser aux méthodes de génération d'images. N'ayant encore jamais fait de machine learning et curieux d'en savoir plus sur les réseaux de neurones, nous avons donc choisi d'utiliser un GAN, Generative Adversarial Network, une architecture de réseaux de neurones qui a la particularité de couvrir un champ étendu de techniques. En effet, il utilise deux types de réseaux de neurones qui vont être en "compétition" l'un contre l'autre, d'où le nom "Adversarial".

Ici ce sont des cartes d'élévation que l'on souhaite générer. Ces cartes de relief sont notamment utilisées dans l'industrie de l'animation pour constituer les futurs décors de jeux vidéos ou de dessin-animés. Dans ce cadre on peut souhaiter obtenir des "décors" obéissant à des caractéristiques précises : hauteur maximale du relief, hauteur minimale, nombre de sommets...

La génération de cartes d'élévations est encore aujourd'hui un problème compliqué, pour lequel il existe trois méthodes principales :

- Générer une carte entièrement à la main, ce qui représente un travail considérable.
- Utiliser des reliefs réels, ce qui limite potentiellement la diversité des décors créés.
- Utiliser un algorithme procédural, tel que le bruit de Perlin, dont le réalisme reste cependant discutable.

Les réseaux de neurones rassemblent les avantages de chacune de ses méthodes : la possibilité de choisir à quoi la carte va ressembler, le réalisme, et la possibilité de laisser un algorithme s'occuper de la génération.

Dans le cadre de notre projet, nous avons donc désiré offrir à l'utilisateur une plateforme où il puisse générer des cartes obéissant à des conditions qu'il souhaite imposer et observer les résultats. Cela sera possible grâce à une application web reliée à notre GAN codé en python.

1 Génération par machine learning

1.1 Generative Adversarial Network

L'idée principale du machine learning pour la génération d'images est, étant donnée une base de données initiale, de générer de nouvelles images ressemblant à notre base de données sans leur être identiques, et éventuellement d'imposer certains critères sur l'image générée.

Pour la génération par machine learning on s'appuie sur un réseau principal chargé de la création d'images, appelé réseau générateur. Cependant, étant donné un réseau générateur, il est difficile de juger de la qualité des images créées et de leur ressemblance à la base de données initiale. Pour cela il faudrait soit des critères très précis et vérifiables algorithmiquement de ce qu'est une "bonne" image, soit un travail très long "à la main".

L'idée intéressante du GAN est d'introduire un nouveau réseau, dit discriminateur, chargé de vérifier à notre place si les images générées sont réalistes ou non.

La force du GAN est d'entraîner conjointement le discriminateur à reconnaître les images "réelles" et à démasquer les "fausses" images comme celles du générateur, et le générateur à tromper le discriminateur. On obtient alors un générateur de plus en plus performant, dans le sens où les images qu'il crée sont de plus en plus proches des images réelles afin d'essayer de tromper le discriminateur. Les deux réseaux sont donc en opposition d'où le nom "Adversarial"

D'un point de vue plus théorique, on crée, au sens de la théorie des jeux, un jeu à deux joueurs (les 2 réseaux de neurones) à somme nulle. Le gain du discriminateur représente sa capacité à reconnaître les vraies images des fausses, et celui du générateur sa capacité à créer des images que le discriminateur ne parvient pas à démasquer.

1.2 Nos réseaux

Ainsi, nous utilisons deux réseaux de neurones : le discriminateur et le générateur. En entrée, un réseau de neurones prend une matrice en trois dimensions : la longueur, la largeur et le

nombre de champs. A la sortie, il y a une matrice de même type mais de taille différente. Le discriminateur et le générateur ont des structures similaires. En effet, un réseau de neurones est composé de couches qu'il exécute à la suite. Une couche contient deux étapes :

— on applique une fonction linéaire
— on applique une fonction non linéaire dite *fonction d'activation*

C'est dans ces deux étapes et dans la taille de l'entrée et de la sortie que le discriminateur et le générateur divergent.

1.2.1 Discriminateur

Le discriminateur prend en entrée une matrice de taille $1 \times 100 \times 100$ le "1" signifiant que l'image est en noir et blanc. Il y aurait eu un "3" si l'image avait été en couleur. Il retourne une matrice de taille 2 que l'on peut voir comme un vecteur. On pourra ensuite renormaliser ce vecteur pour obtenir la probabilité que l'image soit réelle ou fausse.

La fonction linéaire utilisée est une convolution 2D car nos images sont en deux dimensions. C'est-à-dire qu'elle applique des opérations sur les coefficients avec des poids prédéfinis. Ces poids sont ceux que l'on cherche à définir de manière précise pour qu'ils puissent mener les opérations à un résultat qui correspond à nos attentes.

La première fonction non linéaire que nous utilisons est la fonction **relu** qui ne garde que la partie positive. Puis, nous appliquons la fonction **maxpooling** qui ne conserve que certains coefficients de la matrice obtenue pour en réduire la dimension.

Nous avons choisi de prendre 3 couches. L'entraînement a fonctionné pour ce nombre et nous n'avons pas eu besoin de changer.

Ce réseau de neurone est appelé un CNN (Convolutional neural network).

1.2.2 Générateur

Le générateur prend en entrée une matrice de taille $100 \times 1 \times 1$ où 100 représente le nombre de champs que nous prenons. Il retourne une matrice de taille $1 \times 100 \times 100$. La sortie est donc au même format que l'entrée du discriminateur pour que les deux réseaux puissent s'échanger de l'information.

La fonction linéaire utilisée est une convolution transposée. Elle applique également des opérations sur les coefficients avec des poids. Ici, l'objectif est aussi de définir ces poids pour que nous puissions générer une carte d'élévation qui semble réelle. Ces opérations vont faire diminuer le nombre de champs et augmenter la taille de la matrice sur les deux premières composantes qui correspondent à la largeur et à la longueur de la carte.

La première fonction non linéaire utilisée normalise l'image obtenue. Nous ne rentrons volontairement pas dans les détails. Puis, nous utilisons la fonction relu déjà évoquée pour le discriminateur.

Nous avons imposé 5 couches et comme l'entraînement a convergé nous n'avons pas eu à changer le nombre de couches.

On appelle ce réseau de neurone un CNN inversé.

2 Structuration du code et outils

Ce projet étant notre première approche du machine learning, nous avons utilisé la librairie PyTorch sur les conseils de Clémentine Fourrier, facile de prise en main.[1]

Il fallait également un outil permettant à notre application web (HTML, CSS, JavaScript) d'interagir avec nos codes en Python. Pour cela nous avons choisi Flask, un micro-framework de développement web en Python. Celui-ci est flexible et apparemment plus facile d'utilisation pour nous, débutants en développement web, que d'autres frameworks plus élaborés. Flask est constitué d'un ensemble de modules permettant de développer plus rapidement en fournissant des fonctionnalités telles que la gestion des requêtes HTTP ou encore l'affichage de pages web dynamiques.

Nous avons fait de notre GAN un module pour pouvoir ensuite utiliser ses fonctions dans notre application web.

Notre application Flask se situera à la racine du projet étant donné qu'elle constitue le but final. C'est elle qui permettra à l'utilisateur d'utiliser facilement notre GAN et de visualiser les résultats.

TDLOG
- `app.py` ..... notre application web
- `generation.py` ..... le module permettant de générer des maps en chargeant les réseaux de neurones avec les derniers entraînements
- `templates/` le fichier contenant les templates de nos différentes pages web .html
- `static/`
  - `style.css`
  - `script.js`
  - `images/` .. contient les images générées par l'utilisateur et affichées sur la page web
- `src/` ..... Code source du GAN
  - `networks/` ..... contient les fichiers des classes définissant les 2 réseaux de neurones
    - `discriminator/`
    - `generator/`
  - `pre_processing/` ..... outils de traitement de la base de données
  - `utils/` ..... fonctions mathématiques appelées par les autres modules
    - `noise/` ..... contient les scripts utilisés pour générer des "fausses" cartes d'élévation pour l'entraînement
    - `.....`
  - `.....`
- `networks_data/` ..... paramètres des réseaux sauvegardés après entraînement
- `README.md`

3 Bases de données utilisées et entraînement

3.1 Données

3.1.1 Obtention des données

Pour pouvoir entraîner le discriminateur, nous avons eu besoin d'images réelles et de fausses images.

Obtention d'une base de données composée d'images réelles :

Notre base de données est issue du site CGIAR-CSI [2]. Il s'agit d'une base de données regroupant des cartes d'élévation prises par des satellites. Le site internet explique que les données proviennent de la Nasa et que certaines images ont été complétées car il manquait de l'information par endroit. Par exemple dans les endroits où il y a de l'eau (rivière, lac et surtout montagne), les satellites ont du mal à déterminer correctement la topographie du lieu.

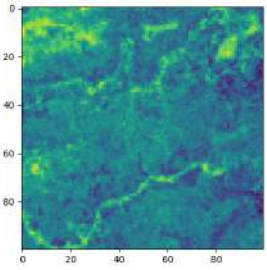
Les données sont obtenues en téléchargeant un par un les fichiers zip contenant les images. Grâce au module : src/pre_processing/unzip.py nous avons pu extraire les .tif qui étaient dans les zip précédemment téléchargés. La taille de ces images étant trop grande (6000 x 6000 pixels) pour les entraînements, nous avons décidé de les couper en sous-images. Cela nous a permis d'obtenir une base de données plus importante également. Le problème avec cette décision est que les cartes d'élévation du désert ou de la côte contiennent beaucoup de sous-images uniformes. Il a donc fallu écrire une fonction python (interesting_image(image) dans TDLOG/Projet/TDLOG/src/pre_processing.py) qui refuse l'image si un niveau est présent à plus de x pour cent dans la sous-image. Nous créons alors un dossier où se trouvent des cartes d'élévation réelles de taille 100 x 100 pixels (TDLOG/SRTMdata/Real).

Obtention d'une base de données composée de fausses images :

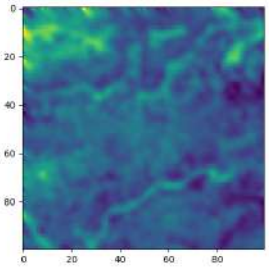
Dans un premier temps, nous avons généré les fausses images en utilisant le *bruit de Perlin*. Nous avons donc un dossier contenant des cartes d'élévation fausses de taille 100 x 100 pixels (TDLOG/SRTM_data/Fake). Dans un deuxième temps et afin que le discriminateur s'améliore, nous avons arrêté de l'entraîner sur ces fausses images et les avons remplacées par les "fausses" images générées par le générateur.

3.1.2 Traitement post-génération

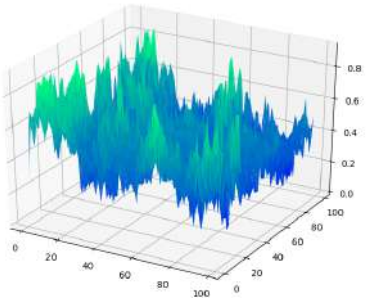
Une fois l'image générée par le générateur entraîné (Figure 1a), on applique une fonction qui permet de réaliser un rendu en 3 dimensions (Figure 1c). Nous avons décidé de flouter l'image en 2 dimensions obtenue (Figure 1b) puis de réaliser le rendu en 3 dimensions (Figure 1d) pour que le dénivelé paraisse réaliste.



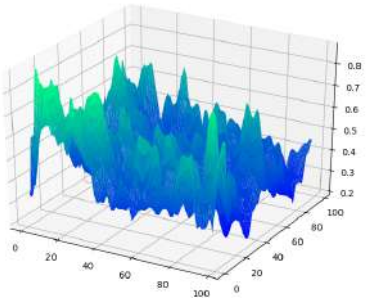
(a) Carte en deux dimensions



(b) Carte en deux dimensions floutée



(c) Carte en trois dimensions



(d) Carte en trois dimensions floutée

FIGURE 1 – Visuel en 3 dimensions et intérêt de flouter l'image

3.2 Entraînements

Dans la suite, on appellera "batch" une liste de taille fixée d'images sur lesquelles on fera les descentes de gradient visant à optimiser les paramètres des réseaux. Nous entraînons nos réseaux de neurones sur plusieurs images et non sur une seule pour éviter d'entraîner sur des cas non-représentatifs. Nous n'entraînons pas non plus nos réseaux sur l'ensemble complet des images pour que les réseaux s'améliorent au fur et à mesure de l'entraînement et convergent donc plus vite. Pour entraîner le générateur et le discriminateur, nous avons besoin que le générateur crée un batch de fausses images, et de le juger par le discriminateur afin : pour le générateur, qu'il trompe de plus en plus efficacement le discriminateur ; et pour le discriminateur, qu'il soit de plus en plus efficace pour repérer les images du générateur.

Il est donc intéressant d'entraîner les deux réseaux ensemble, en premier lieu pour ne pas répéter les inconvénients décrits précédemment, et en second lieu pour obtenir des améliorations progressives de chacun, poussant à chaque étape l'autre à évoluer encore un peu plus, et donc plus rapidement que si nous enchaînions les entraînements unilatéraux de chacun.

L'entraînement se fait donc en 4 étapes :

- Créer un batch de vraies images (issues de la base de données) et un batch de fausses images (créées par le générateur).
- Faire juger ces deux batchs par le discriminateur.
- Évaluer les pertes, c'est à dire l'écart entre ce qu'on attend de chaque réseau et le résultat effectivement obtenu.
  Pour le générateur, il s'agit de la distance entre la probabilité estimée par le discriminateur que les images soient vraies et 1 (le générateur veut que ses images soient confondues avec de vraies images), à laquelle on ajoute également la distance entre les conditions imposées et celles obtenues. Les conditions imposées sont une hauteur

maximale du relief et une hauteur moyenne. [3]
Pour le discriminateur, il s'agit de la distance entre la probabilité estimée pour le vrai batch que les images soient vraies et 1 (il doit être capable de reconnaître que les vraies images sont effectivement issues de notre base de données) à laquelle on ajoute la distance entre la probabilité estimée pour le faux batch que les images soient fausses et 0 (il veut pouvoir reconnaître les fausses images comme telles).

— Optimiser les paramètres des deux réseaux en fonction des pertes trouvées.

En pratique, le discriminateur ne renvoie pas des probabilités mais deux réels, qui peuvent être interprétés comme une "énergie" que le réseau associe à chaque classe (image réelle ou générée) : plus elle est élevée, plus le réseau est convaincu que l'image jugée est dans cette classe. On peut estimer la probabilité voulue en calculant :

$$p_{classe} = \frac{e^{E_{classe}}}{e^{E_{vrai}} + e^{E_{faux}}}$$

La fonction de perte qu'on utilise alors est l'entropie croisée, qui renvoie la quantité

$$d_{classe} = -\log\left(\frac{e^{E_{classe}}}{e^{E_{vrai}} + e^{E_{faux}}}\right)$$

où $classe$ est la classe cible du réseau (toujours $vrai$ pour le générateur, $vrai$ pour les vraies images et $faux$ pour les images générées pour le discriminateur).

4 Interface graphique

4.1 Création du site

Nous avons décidé de présenter nos résultats via une interface web. Nous avons pour cela utilisé le langage de balisage HTML afin de créer la structure de base de notre site web. Nous avons également utilisé les langages CSS et JavaScript afin de gérer respectivement la mise en forme et les éléments dynamiques du site.

Notre site est composé de trois pages distinctes. La structure interne de chaque page se divise en un header, un contenu principal ainsi qu'un footer. Comme toutes les pages partagent le même squelette, nous avons utilisé un héritage. Ainsi, nous définissons une page template de base contenant le header, le footer ainsi que des zones personnalisables et faisons appel à celles-ci lors de la construction des autres pages.

Toutes nos pages partagent également une même feuille de style et un même fichier javascript.

4.2 Utilisation de Flask

Nous avions d'abord créé un site web indépendant pour afficher nos résultats. Mais il nous fallait en réalité un outil pour rendre notre page web dynamique et permettre à l'utilisateur d'interagir. Nous avons donc utilisé Flask et redéfini la manière de charger nos pages. Flask définit en effet un moteur de templates ainsi qu'un système de routes personnalisables.

Flask n'impose pas de structure de fichier mais l'arborescence couramment utilisée consiste à créer un dossier templates/ contenant les fichiers HTML, ainsi qu'un dossier static/ contenant toutes les ressources (feuille de style, images utilisées sur le site, fichier JavaScript).

Comme dans tout framework web, Flask possède un système de gestion de routes. Chaque route possède des vues. Le décorateur **route()** définit l'url à laquelle ces vues doivent être utilisées, et les vues définissent les réponses à renvoyer à l'utilisateur.

Les routes permettent donc de traiter les requêtes utilisateurs. Les deux types de requêtes couramment utilisées sont la requête POST (qui est utilisée lorsque l'on valide un formulaire sur une page web), et la méthode GET utilisée le reste du temps. La méthode GET est par exemple utilisée lorsque l'on se déplace sur une nouvelle page web. En effet, des routes sont définies pour les adresses de nos différentes pages web. De cette manière lorsqu'un utilisateur choisit de se déplacer sur une nouvelle page, nous utilisons la méthode render_template() du module flask afin de retourner le template de la page web correspondante. La méthode POST est utilisée dans notre cas pour récupérer les données rentrées par l'utilisateur lorsque celui-ci soumet un formulaire. Nous utilisons pour cela une balise *form* dans notre code HTML, qui désigne une section contenant des contrôles interactifs. Cette balise possède des attributs, notamment l'attribut *method* qui permet d'indiquer la méthode HTTP qui sera utilisée pour envoyer les données au serveur. Ainsi, la création d'une balise *form* possédant l'attribut POST nous permet de récupérer les données fournies par l'utilisateur. Ces données sont ici les valeurs des sliders qui correspondent aux conditions que l'utilisateur souhaite imposer pour la génération.

4.3 Structure du site

La version finale de notre application comprend donc 3 pages :

- Une page **RESULTATS** où l'utilisateur peut demander à générer une carte d'élévation selon des conditions qu'il impose. Une fois ce formulaire soumis, une page apparaît où les images générées sont affichées.
- Une page **THEORIE** où l'on affiche ce rapport afin que l'utilisateur puisse comprendre notre démarche et le fonctionnement des outils utilisés.
- Une page **RESSOURCES** où l'on peut trouver notre bibliographie et des liens intéressants vers des sites internet qui nous ont été particulièrement utiles pendant notre projet.

5 Limites

Nos approches et implémentations ont cependant, dans une certaine mesure, limité la qualité de nos résultats. Le côut d'entrée pour aborder le GAN et les réseaux de neurones était de plus important et nous a demandé beaucoup de temps. Plusieurs améliorations auraient été possibles avec plus de temps. En particulier,

- Nous n'avons fait que très peu varier les hyper-paramètres (nombre de couches, de neurones par couche, forme des convolutions...) des deux réseaux. Cela a eu pour effet, entre autres, de bloquer la distance entre l'image générée et les conditions imposées par l'utilisateur à un seuil fixe qu'aucun entraînement ne parvient à dépasser.
- Nous n'avons pas testé et comparé les différentes méthodes possibles pour entraîner les réseaux séparément telles que entraîner un réseau puis l'autre ou conjointement,

faire un entraînement initial du discriminateur ou pas, ou encore faire varier la taille des batchs d'entraînement.

— Au début du projet, nous avions eu l'idée ambitieuse d'utiliser un algorithme neural de transfert artistique. L'idée est de prendre une image et un fond et d'entraîner un réseau de neurone qui réalise le mixte des deux. L'image donne les formes principales et le fond donne le style. Dans le cadre de notre projet, le client aurait alors pu dessiner une carte d'élévation grossière à la main sur le site. L'algorithme de transfert aurait rendu ce dessin plus plausible.[4]

— Du point de vue graphique, nous aurions voulu que le rendu en 3 dimensions de la carte d'élévation, qui est opérationnel sur Python, soit également interactif sur le site web et que l'utilisateur puisse le faire évoluer avec des curseurs. Nous avions envisagé d'utiliser l'interface proposée par Python pour la manipulation de la carte 3D.

Références

[1] Tutoriel pour comprendre et utiliser un GAN : `https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.htm`

[2] Adresse où nous avons téléchargé la base de données : `http://srtm.csi.cgiar.org/srtmdata/`

[3] Attribute-Guided Face Generation Using Conditional CycleGAN Yongyi Lu, Yu-Wing Tai, Chi-Keung Tang

[4] Tutoriel pour comprendre et utiliser un algorithme neural de transfert artistique : `https://pytorch.org/tutorials/advanced/neural_style_tutorial.html`