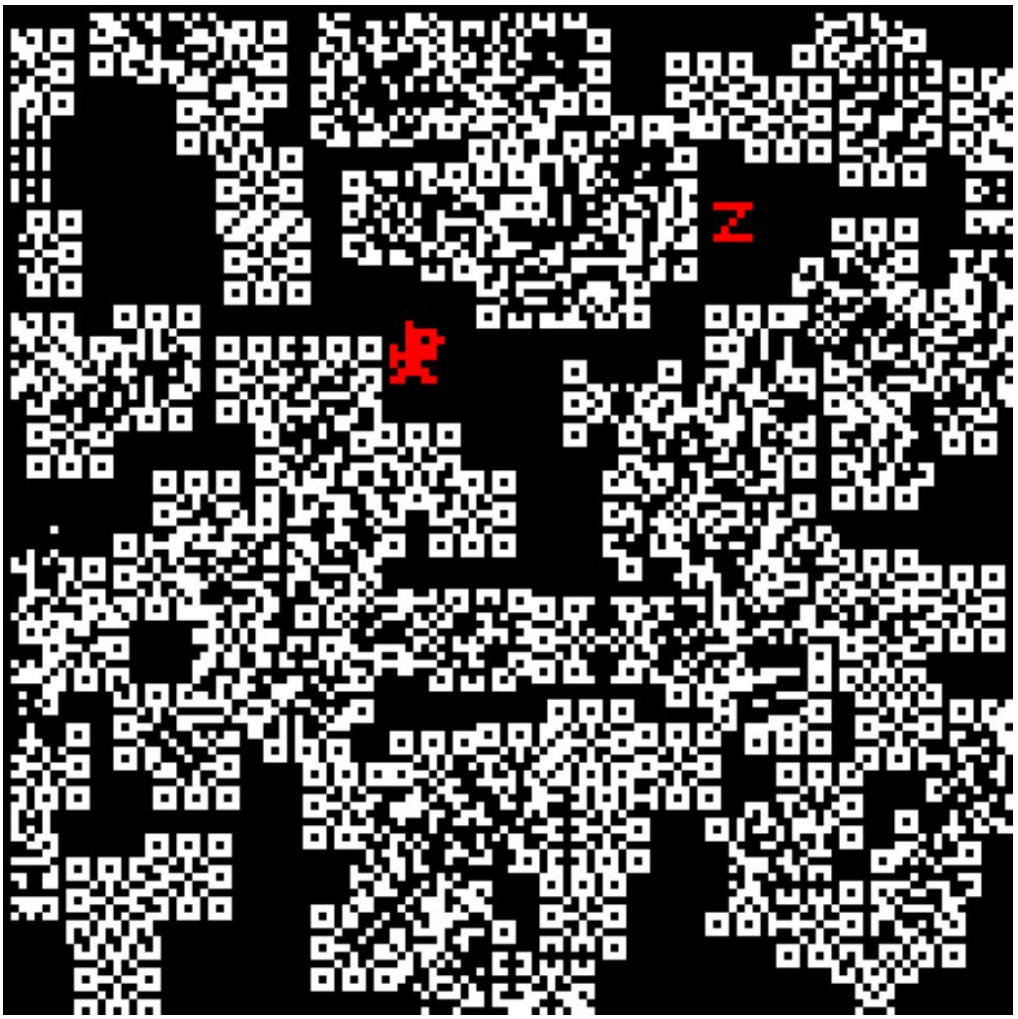
Rapport final

Projet Drowsy



Rabii Younès

Projet Informatique Individuel

Semestre 8 - 2018/2019

Tuteur : M. Baptiste Pesquet

Contexte

Bitsy est un moteur de jeu qui permet à des novices de créer des petites expériences interactives de manière très accessible. Très populaire, on compte à ce jour plus de 1500 jeux Bitsy jouables directement dans le navigateur et que l'on peut retrouver sur Itch.io .

À leur coeur, ils ont tous le même principe : le joueur se balade de case en case dans un petit monde pixelisé et peut interagir avec certains éléments.

Fig 1 : Un écran de jeu Bitsy typique

D'après les auteurs, le charme des jeux Bitsy vient des contraintes qu'impose la plateforme : résolution avec peu de pixels, palette à seulement 2 couleurs, interactions simples avec les objets, etc.
Ces limitations stimulent la créativité des auteurs et leur permettent de composer simplement des petits poèmes interactifs.

Les jeux Bitsy n'ont besoin que de très peu pour plonger le joueur dans l'exploration d'un petit monde.
Le but de ce projet est de mettre au point un système capable, à partir des jeux Bitsys conçus par des créateurs humains, d'apprendre à en créer un lui-même.

Les projets s'appuyant sur des réseaux de neurones pour créer des tableaux donnent souvent l'impression que leur productions ont été tirées d'un rêve un peu flou. Je m'attend ici à des résultats similaires, d'où le nom du projet : ***Drowsy*** qui signifie "endormi" ou "somnolent".

Technologies utilisées

Ce projet est centré sur la mise en place d'une structure particulière de réseau de neurones artificiel : les réseaux antagonistes génératifs (GAN).
Les outils associés sont les bibliothèques Python dédiées au Machine Learning et de l'environnement associé :

- TensorFlow, pour entraîner et faire performer le réseau
- Keras, comme interface de haut niveau avec TensorFlow
- Google Collaboratory, pour effectuer des calculs sur une machine dédiée
- Anaconda, comme environnement de travail

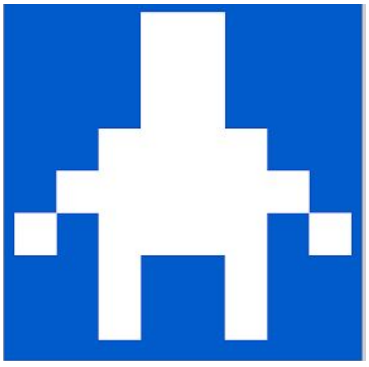
Pour les divers scripts d'extractions ou de formattage des données, j'ai été amené à utiliser l'environnement NodeJS et sa capacité à manipuler les fichiers et faire des requêtes HTTP.

Création du Dataset

Forme des données

Avatar

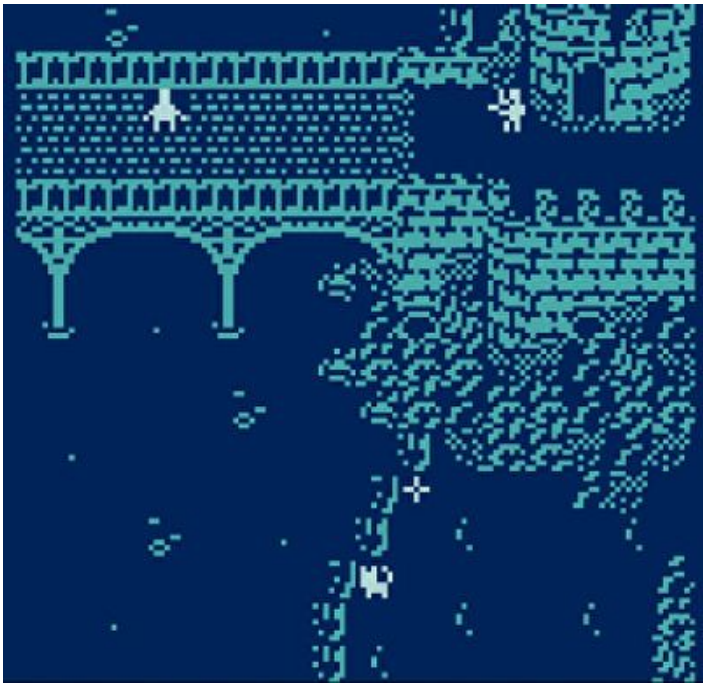
Un *avatar* est une image qui représente le joueur dans un jeu Bitsy.
Il s'agit d'une image de 8x8 pixels, où chaque pixel est indexé dans une palette à deux couleurs uniquement.



Dans notre modélisation, on ignore les couleurs renseignées par la palette et on considère un avatar comme étant une matrice 8x8 contenant soit des 0, soit des 1.

Fig 2: Un avatar représentant une silhouette humaine

Room



Le joueur se déplace d'une *room* à l'autre. Ces rooms sont des images qui représentent l'environnement statique. Elles sont elles mêmes composées de 16x16 *tiles*, qui sont des blocs de 8x8 pixels qui peuvent être réutilisés plusieurs fois dans la même *room*. Une room est donc une image de 128x128 pixels bichromatique.
On la modélise ici comme une matrice 128x128 contenant des 0 ou des 1.

Fig 3 : Une room tirée d'un jeu Bitsy, représentant un pont au-dessus d'un fleuve

Acquisition et formatage des données

Puisqu'il n'existe pas de dataset qui rassemble les données de tous les jeux Bitsy, j'ai dû en composer un moi-même. Afin d'obtenir suffisamment de données et les formater d'une manière exploitable, j'ai conçu un outil dédié à cette tâche.

La plupart des jeux Bitsy étant hébergés sur la plate-forme en ligne itch.io, j'ai mis en place un script NodeJS qui :

- prend l'URL d'un jeu hébergé sur itch.io
- opère des requêtes HTTP pour passer de lien en lien et accéder au code source
- repère les parties qui correspondent aux *avatars* et aux *rooms*
- télécharge et formate ces données en JSON

Tous les jeux n'étant pas exploitables de manière automatique, j'ai pu extraire les données de 22 jeux Bitsy. En les complétant avec une compilation officielle de tous les *avatars* existants, le *dataset* final contient **420 *avatars* et 595 *rooms***.

Architecture pour la génération d'Avatars

Mise en place du GAN

Les GAN (Réseaux Antagonistes Génératifs) sont des algorithmes d'apprentissage capables, à partir d'un set de données initial, de s'entraîner à produire de nouvelles données qui possèdent les mêmes caractéristiques.

Ils sont composés de deux briques fondamentales :

- un réseau Discriminateur, qui apprend à distinguer les vraies et fausses données

- un réseau Générateur, qui transforme des données aléatoires en données qui ont une structure

  On peut voir la concaténation de ces deux réseaux (Générateur, puis Discriminateur) comme un seul réseau appelé Antagoniste.

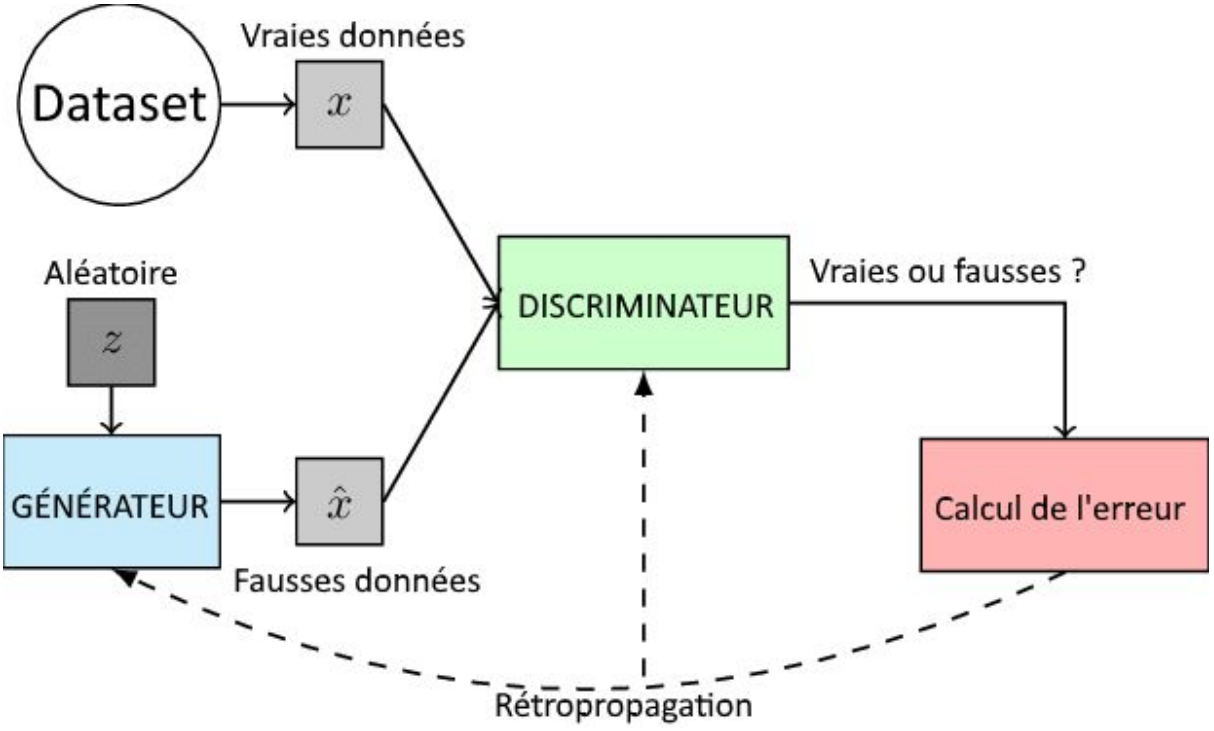


Fig 4 : Schéma de fonctionnement du GAN

Les réseaux Discriminateur et Antagoniste sont entraînés l'un après l'autre.

À terme, le Discriminateur devient de plus en plus performant à distinguer les données fabriquées, et la partie Générateur de l'Antagoniste à produire des données indiscernables des vraies. On obtient donc des données fabriquées, mais qui semble être des vraies.

Implémentation

La structure des réseaux Discriminateurs et Générateurs doit être adaptée aux données. Dans notre cas, nous manipulons des matrices qui représentent des images, nous utilisons donc des réseaux convolutifs comme il est conseillé dans la littérature.

Le réseau Générateur part d'une couche de la taille des images d'entrées, et modifie la dimension de l'information couche par couche, jusqu'à obtenir une probabilité.

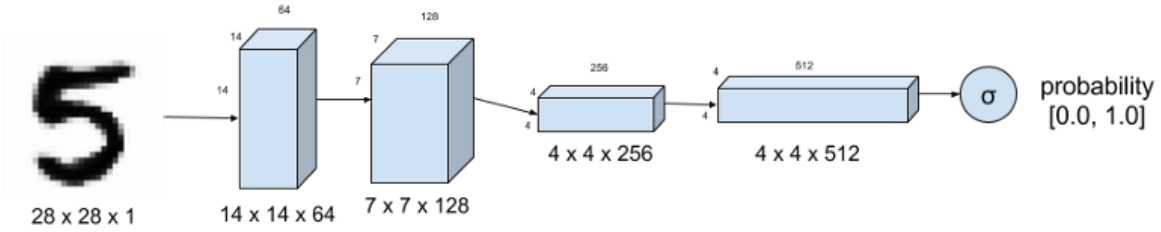


Fig 5: Visualisation de la structure du réseau Discriminateur
(les dimensions ne sont pas les mêmes que celle de l'implémentation)

Le réseau Générateur fait l'inverse : il part d'un vecteur aléatoire et le transforme couche par couche pour obtenir une image.

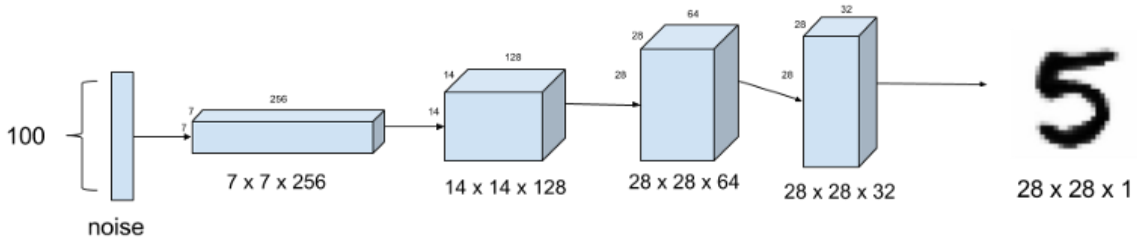


Fig 6: Visualisation de la structure du réseau Générateur
(les dimensions ne sont pas les mêmes que celle de l'implémentation)

Certains paramètres très spécifiques comme le nombre d'exemples présentés à chaque session d'entraînement (*batch size)* ou le taux d'entraînement (*learning rate*) ont été configurés d'abord en partant des valeurs trouvées dans la littérature, puis en les adaptant par tâtonnement à notre cas précis.

Résultats

Fig 7: Panel de résultats obtenus après 60.000 itération d'entraînement (1h de calcul)

Format

Le premier constat est que les images produites ne sont pas exploitables directement. En effet, le réseau Générateur produit par construction des matrices de réels et non des entiers. Ici, un pixel blanc correspond à 1.0, un noir à 0.0, et un gris à un nombre compris dans [0.0, 1.0]

Au fil de l'entraînement, les valeurs associées aux pixels se polarisent autour de 0 et 1, mais il en reste toujours qui sont autour de 0.5 par exemple.

Pour obtenir des images complètement exploitables par un jeu Bitsy, nous pouvons par exemple arrondir les valeurs à l'entier le plus proche, impliquant un traitement post-génération.

Esthétique

Les *avatars* obtenus possèdent l'esthétique désirée : il s'agit de formes à la frontière entre parfaitement reconnaissables et évanescentes. Elles possèdent une forte capacité d'évocation, exploitant l'imagination du joueur pour lui donner un sens.

Diversité

Lorsque on arrête l'entraînement du Générateur, une bonne partie des images produites semblent similaires. Il s'agit d'un phénomène très courant sur les GAN : le Mode Collapse. Alors que les données fournies pourraient être réparties en plusieurs classes distinctes, les données produites ont tendance à se retrouver dans uniquement une ou deux de ses classes. Il s'agit d'une problématique importante dans le domaine, qui est encore étudiée à ce jour et qui n'est pas trivialement résolvable.

Dans notre cas précis au bout d'un nombre fixé d'entraînement, les *avatars* produits ont souvent tendance à soit avoir une silhouette humaine, soit être un mélange de deux ou trois exemples du dataset. Cela convient parfaitement pour la création d'un seul jeu Bitsy, mais dans l'optique de faire un générateur de jeux, il y aura à terme un manque de diversité.

Pour éviter cela, une solution serait de réentraîner périodiquement les réseaux pour que le Mode Collapse se place sur des classes différentes. En gardant en mémoire les derniers poids synaptiques des réseaux, il est possible de n'effectuer que quelques entraînements à chaque fois, sans avoir à le réentrainer complètement. Cette pratique ne serait pas coûteuse en temps de calcul , et assurerait une diversité des avatars générés.

Architecture pour la génération de Rooms

Obstacles à l'utilisation d'un GAN

Pour la génération de rooms, l'approche initiale était, tout comme pour la génération d'*avatars*, de mettre en place un GAN. Plusieurs obstacles ont entravé cette démarche.

Performances

Générer des avatars impliquait de travailler avec un réseau capable d'analyser des matrices binaires 8x8. Les rooms quant à elles ont une dimension de 128x128. Les données à manipuler étant 256 fois plus grandes, les différentes couches doivent elles aussi être mises à l'échelle.

Gérer un réseau d'une taille aussi massive n'étant pas possible pour une machine qui n'est pas dédiée à ce travail, il a fallu se tourner vers d'autres alternatives qu'un simple ordinateur portable.

N'ayant pas un accès physique à une machine de calcul ni le budget pour en louer une en ligne, je me suis tourné vers la plateforme de calcul Google Collab.
J'ai pu y faire tourner un GAN aux bonnes dimensions, moyennant un temps d'entraînement très conséquent. Il faudrait au moins 1 journée de calcul entière pour un entraînement adéquat, chose que Google Collab ne permet pas facilement.

Structures des résultats

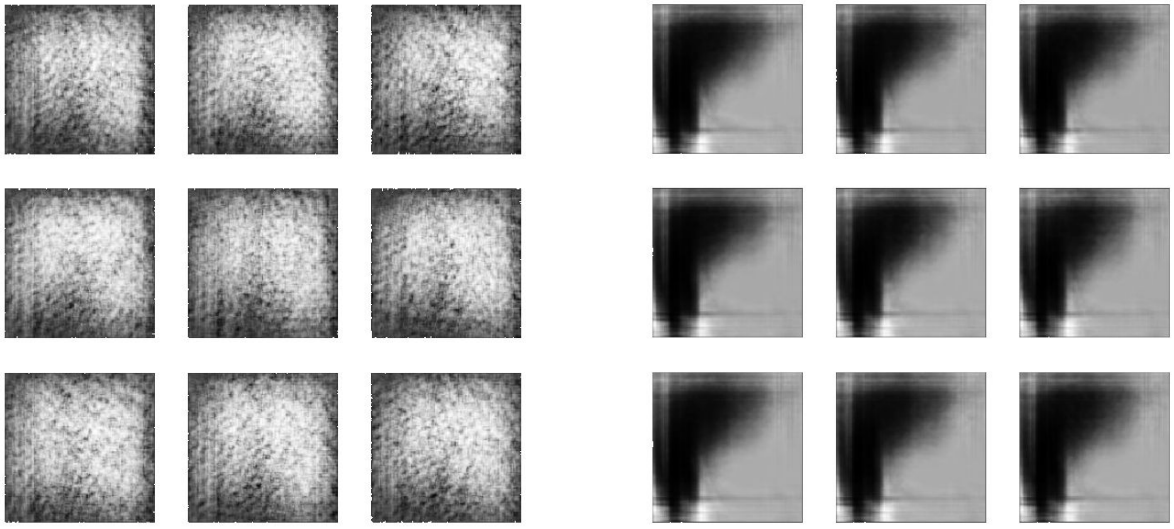


Fig 8 : Résultats obtenus après 2 sessions différentes de 100 itérations d'entraînement (4h de calcul)

Tout comme le GAN des *avatars*, les sorties de ce Générateur sont des matrices contenant des réels. Cette fois-ci les valeurs ne convergent que très lentement vers 0 ou 1. Les structures présentes sur les images produites sont celles d'un gradient presque continu de valeur, avec une vague silhouette sur toute l'échelle de l'image, et aucune structure particulière à petite échelle. Ces caractéristiques sont très éloignées de celles que l'on retrouve dans les *rooms*.

Fig 9: Exemple d'une room typique du dataset représentant un camping en forêt

Les *rooms* du dataset présentent des distinctions très nettes entre les zones qui correspondent à des parties tangibles de l'environnement (blanc) et celles qui sont traversables (noires). Elle sont composées de petites structures symétriques telle un arbre, qui sont répétées pour en former des plus complexes, telle une forêt. On y trouve aussi des éléments plus disparates, représentant des objets placés dans l'environnement.

Ce que produit le GAN au bout de plusieurs heures de calculs est très éloigné de ce qui est attendu. Nous pouvons éventuellement penser qu'avec un entraînement suffisamment long, le réseau serait capable de générer ses caractéristiques. Cependant, ces résultats particulièrement pauvres indiquent une profonde inadéquation entre la structure de notre traitement et celle de nos données, nous invitant à la repenser depuis le début.

Conception d'une structure plus adaptée

Transposition de l'espace des données

Les caractéristiques recherchées dans la génération de rooms (présence de structures complexes composées d'éléments symétriques) sont assez communes sur d'autres types de systèmes : les **automates cellulaires**.

Il s'agit d'un modèle de calculs qui, à partir d'une grille de départ, applique des règles de passages locales sur ses cellules pour obtenir une nouvelle grille. Itération après itération, on obtient des grilles qui présentent des caractéristiques proches de celles désirées.

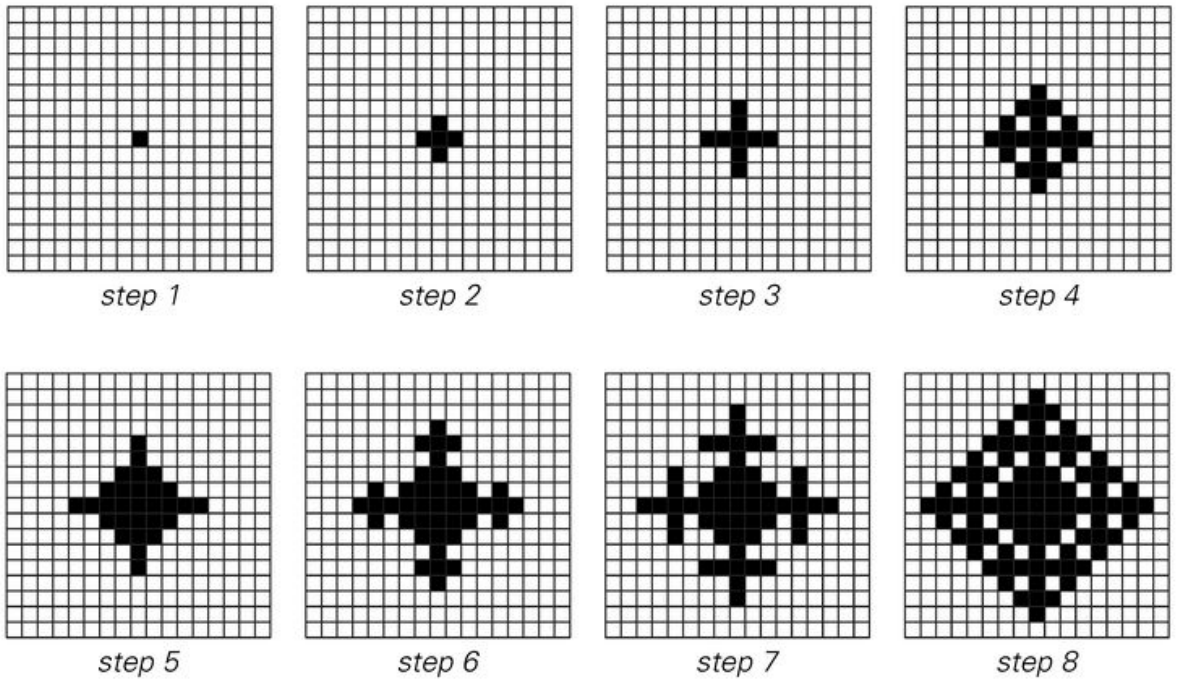
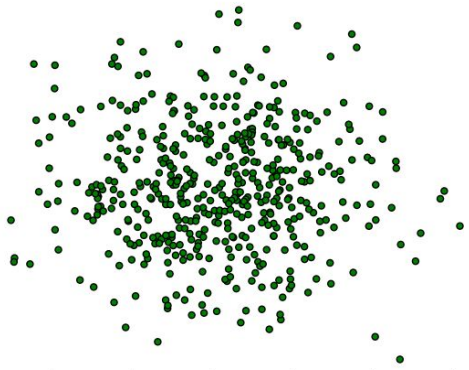


Fig 10 : Automate cellulaire appliquant itérativement ses règles sur une grille

Nous décidons donc dans la suite de la modélisation de penser non pas les *rooms* comme une simple matrice 128x128, mais comme une grille qui a été transformée par un automate cellulaire. Pour décrire parfaitement une telle grille, il suffit d'avoir ses règles de passages, sa grille de départ et le nombre d'itérations qu'elle a subit.

En effectuant les bons choix de modélisation, on peut à la fois réduire le nombre de données à manipuler pour générer une *room*, et obtenir une génération qui présentera des caractéristiques attendues.

Grille de départ



Au lieu de stocker les 128x128 nombres d'une grille de départ, nous décidons que la population de cellules initiales suivra une loi de distribution comme une loi normale ou une loi uniforme. Il nous suffit donc de stocker les paramètres de ces lois de distribution, comme sa moyenne et sa variance.

Fig 11 : Population de points suivant une distribution normale

Règles de passage

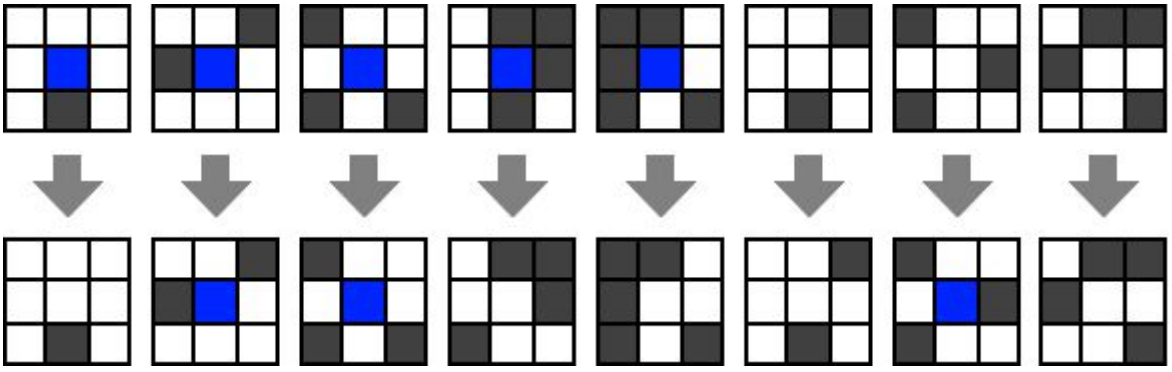


Fig 12 : Visualisation des règles de passage d'un automate cellulaire en fonction des voisins d'une cellule

On se limite à une classe d'automate cellulaire qui ne prend en compte que le nombre de voisins immédiats d'une cellule dans le même état qu'elle pour décider de son état à la prochaine génération. Puis qu'il y a 8 cellules voisines à une cellule, et 2 états possibles, il nous suffit de 18 nombres pour décrire les règles de l'automate cellulaire : chaque nombre représente une configuration différente, et sa valeur (0 ou 1) est le futur état de la cellule.

Nombre d'itérations

Nous faisons le choix de garder un nombre d'itérations constant pour chaque génération, et après quelques essais avec différentes valeurs, de garder ce nombre relativement bas : 5 itérations.

Avec cette modélisation, on se retrouve à manipuler des vecteurs à 20 nombres pour la génération d'une *room*, plutôt que de travailler sur les 128x128 = 784 pixels qui la

compose : on a changé d'espace de données. On appelle ces 20 nombres qui décrivent comment générer une *room*, ses "règles de génération".

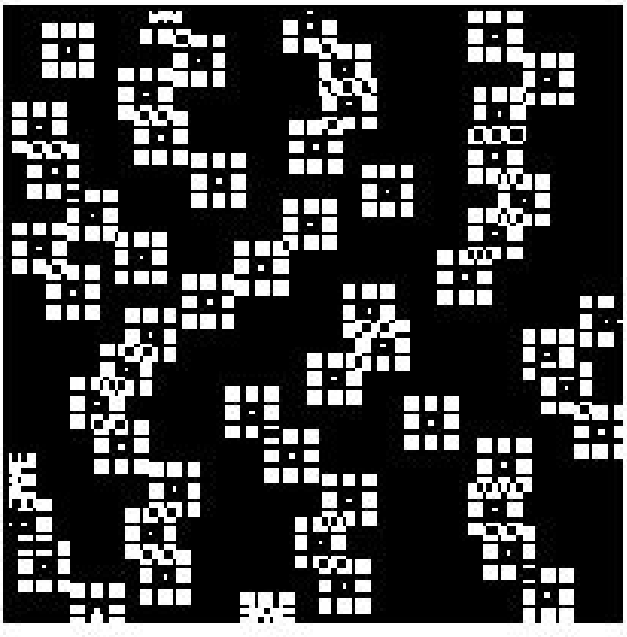


Fig 13 : Exemple de room créée en appliquant des règles de génération

Adaptation de la composante Antagoniste

Inadéquation avec un réseau Générateur

L'endroit de la structure du GAN à modifier pour prendre en compte notre nouvelle façon de décrire les données serait à priori au niveau du réseau Générateur. Or, cela ne correspond pas vraiment à la façon dont on souhaite les manipuler : la création d'une map à partir de ses règles de génération est un algorithme simple et exact, alors qu'un réseau de neurones doit apprendre à l'effectuer, et cela ne sera que statistiquement correct.

Entraîner un réseau pour qu'il apprenne à exécuter correctement toutes les règles de génération serait très coûteux en termes de ressources et incertain en termes de résultats alors que nous avons déjà une méthode définie pour le faire. Il n'est pas non plus trivialement possible d'avoir un Générateur qui travaille sur des règles de génération qui seraient post-traitées par des automates cellulaires avant d'être redonnées au Discriminateur : la façon dont la rétropropagation de l'erreur est

calculée empêche d'intercaler un algorithme entre les deux moitiés du réseau Antagoniste et espérer qu'il soit entrainé. Pourtant le Générateur est une composante importante du réseau Antagoniste, dont la capacité d'apprentissage est cruciale pour le fonctionnement du GAN.

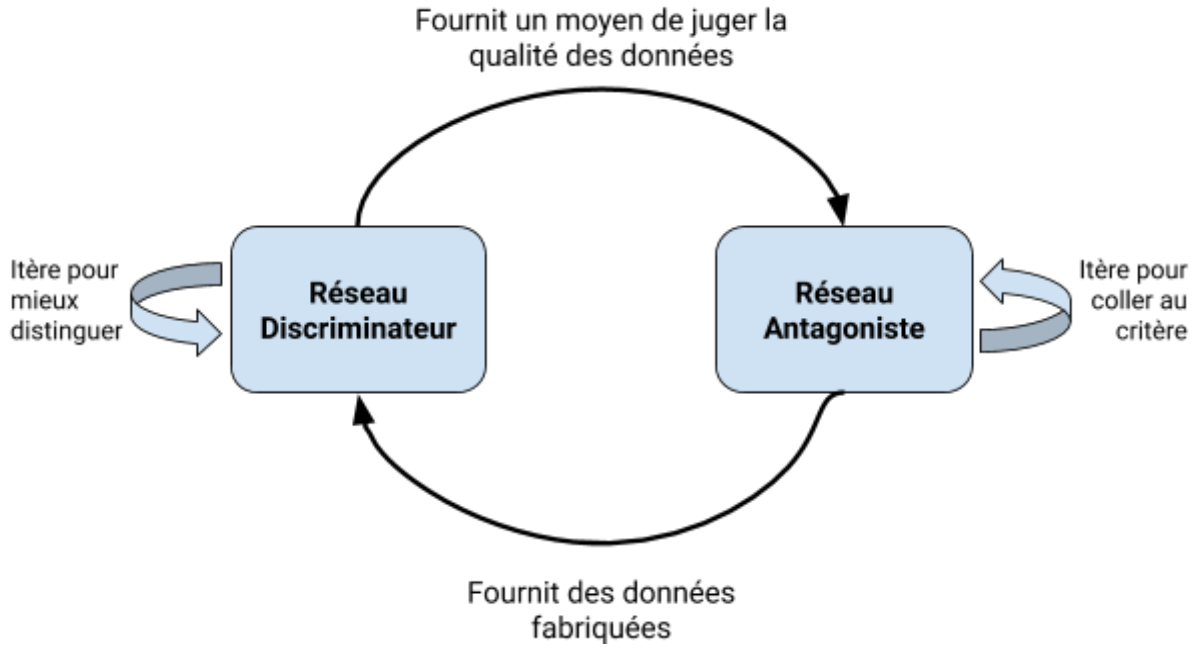


Fig 14 : Relations entre les réseaux Discriminateur et Antagonistes du GAN

L'enjeu est désormais d'adapter la structure du GAN pour que la partie Antagoniste soit toujours capable de :

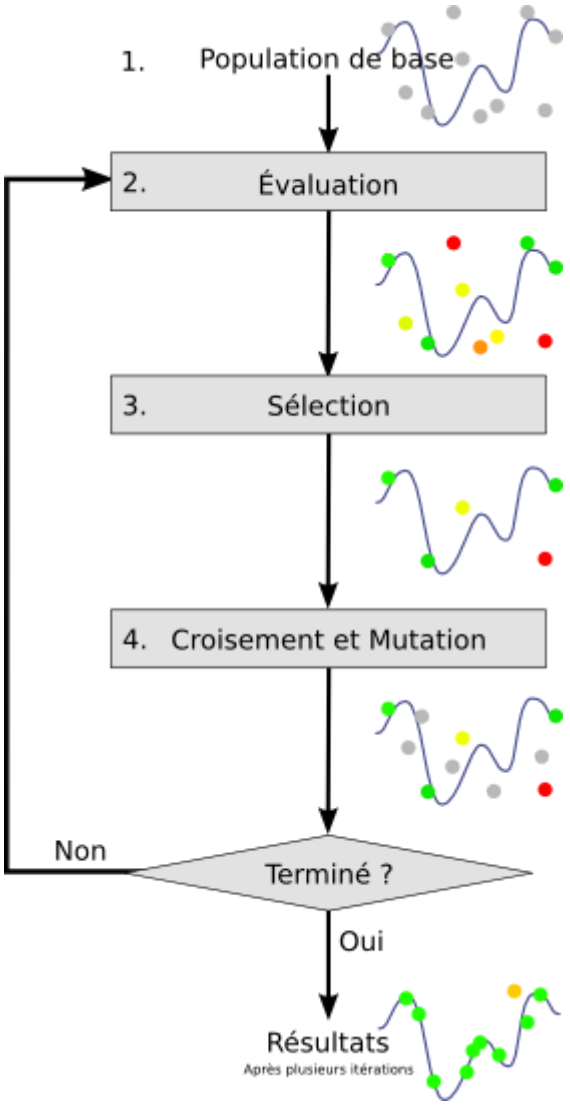
- Fournir des données fabriquées au bon format
- Capable de coller de mieux en mieux à un critère (fourni par le Discriminateur)

mais aussi de :

- Générer les grilles de manière exacte, selon les règles de génération

Utilisation d'un algorithme génétique

La solution proposée dans ce projet est de non pas avoir une partie Antagoniste sous la forme d'un réseau de neurones, mais d'un **algorithme génétique**.



Il s'agit d'un algorithme itératif capable de faire évoluer une population de données pour qu'elle convienne de plus en plus à un critère défini.

Ce type d'algorithme a les prérequis suivants :

- Une population initiale d'individus avec la même structure.
Ici, il s'agira d'un ensemble de règles de génération, dont les valeurs seront prises au hasard.

- Des gènes caractérisant un individu : il s'agira ici des 20 nombres décrivant les règles de générations.

- Une capacité à faire muter les gènes : ici, faire varier les valeurs des règles de génération. Soit inverser la valeur binaire des règles de l'automate cellulaire, soit modifier légèrement les paramètres de distribution de la population initiale.

Fig 15 : Schéma de fonctionnement d'un algorithme génétique

- Une fonction d'évaluation d'un individu : Ici, il s'agira de la prédiction fournie par le réseau Discriminateur sur une room créée en suivant les règles de génération de l'individu considéré. En effet, le réseau discriminateur renvoie une valeur proche de 1 si la room ressemble à une vraie room, et proche de 0 si elle est reconnue comme étant fausse. Cette fonction d'évaluation est donc à valeur dans [0 ; 1] et permettra de classer les règles de génération des plus propices à créer de bonnes rooms aux plus mauvaises.

On obtient donc le couplage Réseau Discriminateur / Algorithme Génétique Antagoniste (RD/AGA) suivant :

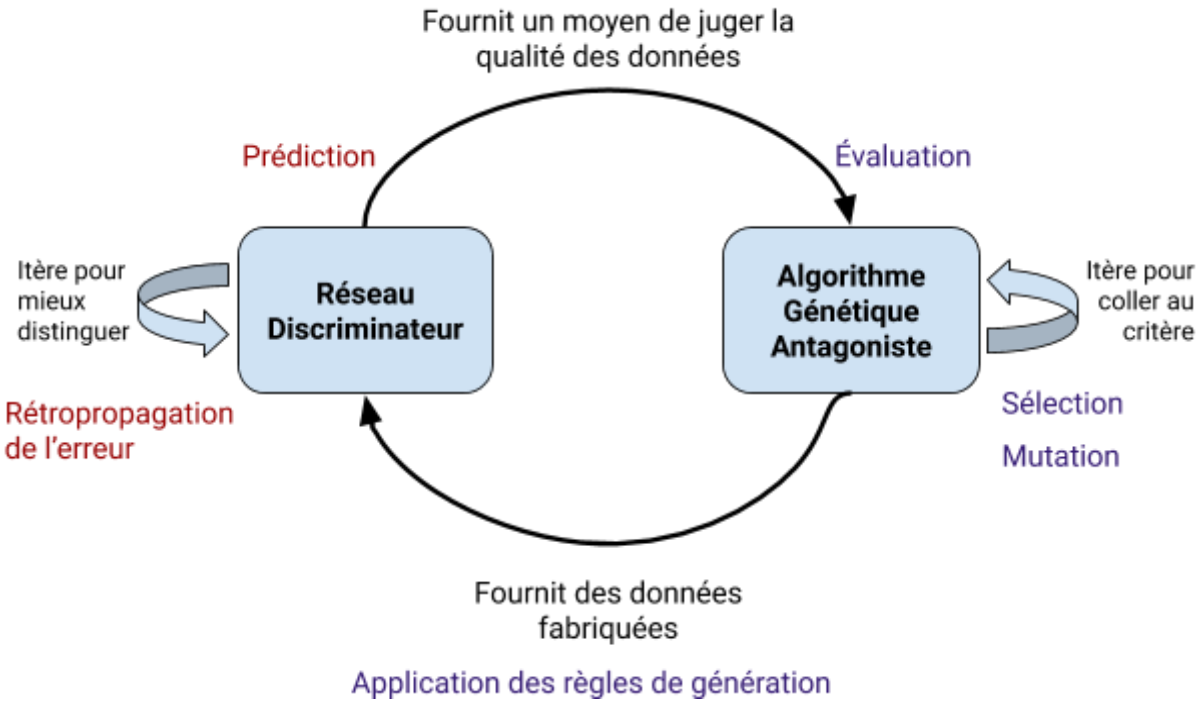


Fig 16 : Relation entre le Réseau Discriminateur (rouge) et l'Algorithme Génétique Antagoniste (violet) et leurs intéractions

Après recherches, ce n'est pas une structure qui a déjà conçue et été étudiée dans la littérature du domaine. Il n'y a donc pas d'étude particulière sur une façon de la paramétrer, et il s'agira avant tout ici d'une première utilisation expérimentale, où les paramètres seront pris d'abord sur des valeurs classiques puis adaptées par tâtonnement.

Adaptation de la méthode d'entraînement

L'entraînement du GAN était alterné : on entraînait séquentiellement le Discriminateur, puis le Générateur, une session à la fois. Dans le cas de notre couplage RD/AGA cette méthode était infructueuse. En effet, l'algorithme génétique a besoin de plusieurs itérations pour faire émerger des règles de générations avec des caractéristiques intéressantes, qui sont nécessaires au discriminateur pour apprendre efficacement.

L'approche choisie a été d'entraîner chaque moitié du couple en plusieurs sessions avant d'entraîner l'autre. Si la moitié en cours d'entraînement avait des résultats trop

pauvres, comme une précision trop basse pour le Discriminateur ou des prédictions trop faibles pour l'Antagoniste, elle était forcée à s'entraîner encore.

Cette adaptation permet de concentrer le temps de calcul sur la partie du couple qui a besoin de progresser, et ainsi rattraper le retard qu'elle avait accumulé sur l'autre.

Résultats

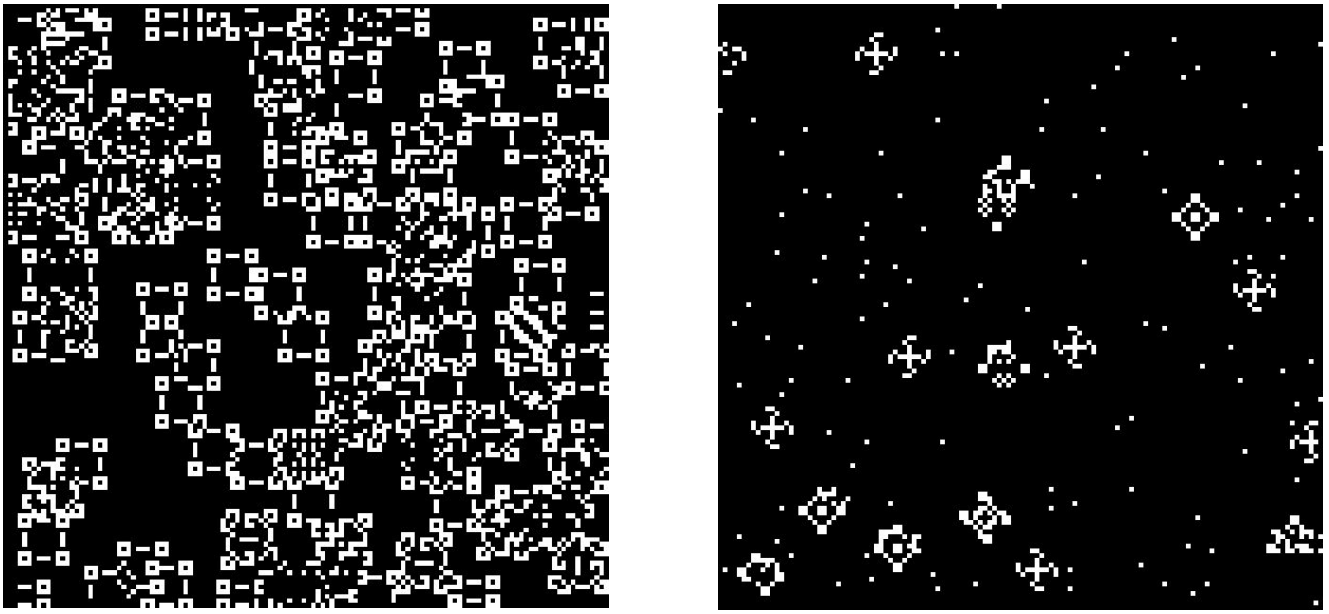


Fig 17 : Exemples de rooms générées par le couple RD/AGA (1h d'entrainement)

Performances

La structure du couple RD/AGA étant plus légère que celle d'un GAN, elle est beaucoup moins coûteuse en terme de performances et elle a pu être mise en place sur un ordinateur portable équipé d'une carte graphique.

Dès le lancement initial on obtient des *rooms* interprétables directement par le moteur Bitsy et qui présentent des structures marquées.

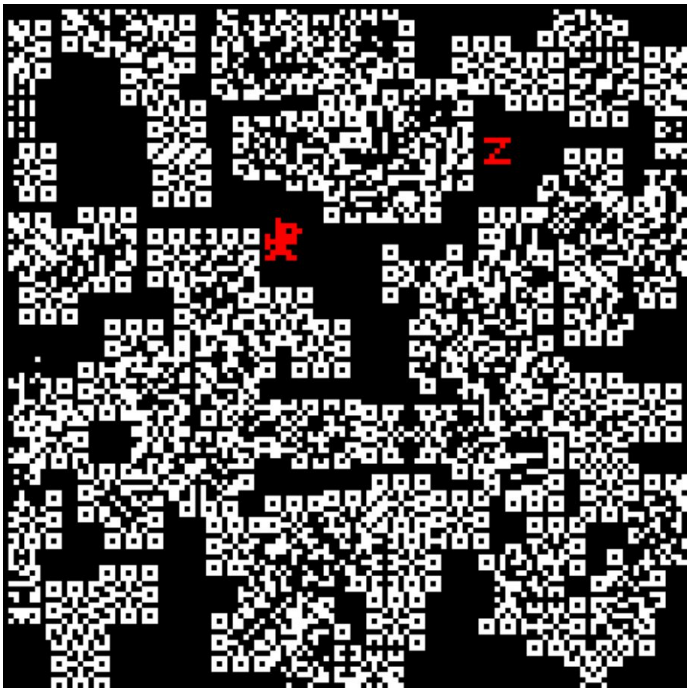
Esthétique

L'esthétique résultante correspond beaucoup plus aux attentes initiales : on note plusieurs petites structures répétées pour en former des plus complexes. La nature des structures élémentaires est propre à chaque règles d'automate cellulaire, garantissant une grande diversité dans les résultats. Enfin, la force d'évocation des *rooms* générées est très bonne, donnant une sensation de mystère autour de l'environnement généré et laissant libre cours à l'imagination du joueur.

Assemblage du jeu final

Pour composer le jeu généré, la démarche suivante a été suivie :

- Entraîner le GAN des avatars durant 1 heure
- Choisir une des images produites par le Générateur
- Appliquer un traitement pour arrondir les valeurs à des entiers
- Entraîner le RD/AGA des avatars durant 1 heure
- Choisir 10 images produites qui seront les rooms du jeu
- Importer toutes ces données dans le moteur de jeu Bitsy



Un texte introductif de mise en ambiance a été ajouté pour inviter le joueur à considérer l'environnement exploré comme un monde mystérieux et flou, semblable à un rêve.

Pour distinguer le joueur de l'environnement, la couleur rouge lui a été appliquée.

La sortie de chaque room a été indiquée par les lettres "ZZZZZ" associées au sommeil.

Fig 18 : Capture d'écran du jeu final

Excepté ces éléments, tout le contenu du jeu a été généré par les structures mises en place dans le projet. Il est jouable sur navigateur, à l'adresse suivante :

https://pyrofoux.github.io/Drowsy/

Bilan du projet

Gestion de projet

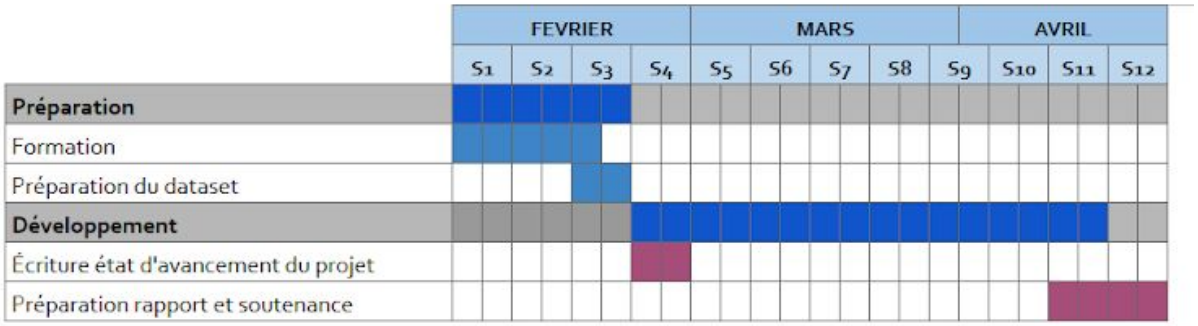


Fig 19: Planning initial et réel du projet

Le choix a été fait dès le début de séparer le projet en deux phases :

1) Préparation et documentation
- Formation
    - Comprendre la théorie derrière les GAN
    - Prendre en main les librairies Python
    - Mettre en place un environnement de travail
- Préparation du dataset
    - Analyser l'architecture des données d'un jeu Bitsy
    - Créer un script JS pour récolter automatiquement et créer un corpus de codes sources des jeux Bitsy

Cette phase était budgétée en terme de temps pour être sûr de laisser de grande marges pour la suivante.

2) Développement

En accord avec le tuteur, la phase de développement a été suivie selon un processus agile : effectuer rapidement un premier prototype pour valider les choix techniques, puis l'enrichir petit à petit pour atteindre les niveaux de fonctionnalités décrit dans le cahier des charges. Cette manière d'aborder le projet a porté ses fruits : le fait de s'assigner les tâches de façon souple a permis de passer plus de temps sur des concepts peu maîtrisés pour en approfondir les connaissances sous-jacentes, ainsi que de prendre de nouvelles directions lorsque confronté à des obstacles imprévisibles. La recherche et la conception du couplage RD/AGA pour rebondir sur

les problématiques rencontrées dans le dernier quart du projet n'aurait pas été possible sans cette philosophie de conduite.

Que cela soit dans la phase planifiée de préparation ou celle de développement agile, il n'y a pas eu de retard sur le planning initial.

Respect des exigences

Le projet Drowsy est sur le plan fonctionnel évalué selon des niveaux de fonctionnalités :

1. Générer l'image du joueur (un avatar)
2. Générer un environnement visuel (une room)
3. Générer un environnement explorable (une room avec des cases vides ou pleines)
4. Générer un monde composé de plusieurs environnements (plusieurs rooms liées)

Il était prévu à minima d'atteindre les niveaux 1 et 2, et d'envisager d'aller jusqu'au 3ème. La conception du GAN du niveau 1 était censée être réexploitable pour accélérer le développement du niveau 2. Cette opportunité s'est révélée dans la pratique infaisable, et a nécessité d'innover une nouvelle structure pour réaliser le second niveau.

En somme, les niveaux 1 et 2 étant complètement fonctionnels, on considère que le projet est satisfaisant par rapport aux exigences fonctionnelles initiales.

Les exigences annexes que sont :

- la capacité à faire tourner les structures de Drowsy sur une machine courante avec des performances raisonnables
- une esthétique qui est propice à faire marcher l'imagination du joueur

sont elles aussi respectées.

Apport personnel et suites du projet

Ce projet a été l'occasion pour moi de mettre un pied dans le domaine de la *Computational Creativity* et de prendre en main des techniques d'IA toutes récentes comme les réseaux de neurones antagonistes, et les technologies associées.

Ces compétences seront un atout crucial pour le déroulement de mes prochains projets ainsi que pour obtenir des stages dans le domaine.

J'ai également été amené à exploiter toutes mes connaissances dans le domaine de l'IA pour mettre en place une nouvelle structure hybride, capable d'étendre le concept des GAN à d'autres types de génération. Ce couplage est expérimental, et ne semble pas avoir déjà été étudiée dans les domaines de l'IA. Il s'agit peut-être là d'un nouvel objet d'étude, sur lequel j'aimerai travailler plus longtemps.

Le jeu généré par Drowsy sera présenté aux communautés dédiée à la génération procédurale dans les jeux vidéos, et elles seront invités à donner leurs retours et à collaborer à ce travail de manière Open Source.