Mémoire de recherche

Étude de la génération procédurale par réseaux de neurones

Master « jeux et médias interactifs numériques » cohabilité par le Conservatoire national des arts et métiers, l'Université de Poitiers et l'Université de La Rochelle

Florian Dormont

August 9, 2016

Contents

Abstract

Ce document est un mémoire de recherche dans le cadre d'un Master JMIN 1 de l'année 2015-2016 à l'École Nationale des Jeux et Médias Intéractifs Numéiques (ENJMIN). Il concerne un étude réalisée sur l'utilisation des réseaux de neurones et l'apprentissage profond dans la génération procédurale destinée au développement vidéoludique. Le dossier est strcturé en plusieurs parties : l'historique, l'explication du fonctionnement d'un réseau de neurone aisni que des applications déjà faites de cette technique, l'objectif de la recherche et la réalisation des expériences.

Part I
Les réseaux de neurones

1 Principe d'un neurone

1.1 Biologie

1.1.1 Note

Le fonctionnement précis biologique n'est pas la priorité pour la compréhension du fonctionnement d'un réseau de neurones mais il est intéressant de s'intéresser à sa structure pour pouvoir faire les parallèles.

1.1.2 Un neurone?

Un neurone (du grec ancien νεῦρον, nerf) est une cellule qui est à la base du fonctionnement du système nerveux. Le neurone est principalement composé d'un cœur appelé soma (ou péricaryon) et d'extrémités telles que les axones qui transitent l'information ou les dendrites qui la reçoivent d'autres neurones. Un neurone n'a qu'un seul axone mais peut posséder plusieurs dendrites. La communication se fait au niveau des synapses qui agissent de manière chimique par secretion ou réception d'hormones telle que l'ocytocine ou la dopamine ou de manière électrique. Les neurones ont donc la possibilité de pouvoir communiquer des informations mais ils ont aussi la possiblité de pouvoir traiter des stimulis externes ainsi que de faire évoluer les connexions entres neurones pour s'adapter en fonction de ces stimulis, la plasticité synaptique. C'est ce qui fait la force et la flexbilité du système nerveux et succité l'intérêt des mathématiciens pour les formaliser en un modèle utilisable.

1.2 Mathématique

1.2.1 Un neurone formel?

Un neurone formel est un modèle mathématique calqué sur son équivalent bilogique. Composé d'une à plusieurs entrées (représentants les dendrites) et d'une sortie (représentant l'axone) qui ont généralement appliqués par des facteurs pondérants. Le neurone peut ainsi traiter ses entrées et communiquer le résultat qui sera soit traité par d'autres neurones, formant ainsi un réseau de neurones ou directement liés à la sortie du processus.
Un neurone formel possède aussi une phase d'apprentissage qui lui permet de s'adapter aux stimulis qui lui sont appliqués, reflétant la plasticité synpatique.

1.2.2 Sigmoïde et activation

La phase d'activation d'un neurone est gérée par sa fonction d'activation. Ainsi, au fil de l'histoire du neuron artificiel, différentes formules ont permi de générer des résultats différents à partir des entrées mais une est plus utilisée que les autres dans les réseaux modernes : la fonction sigmoïde. $f_{sig}(x) = \frac{1}{1+e^{-x}}$ Cette fonction est employée pour diverses raisons : infiniement dérivable, cela simplifie et accélère le calcul de la dérivée; elle renvoie des valeurs dans $[0; 1]$, d'où la non-nécéssité de normaliser les résultats.

2 Historique des réseaux de neurones

2.1 Historique du neurone formel

Le neurone formel a été réalisé en premier par Warren McCulloch et Walter Pitts en 1943[7] avec leur Threshold Logic Unit (Unité à seuil logique), utilisant des entrées et sorties binaires. Il était ainsi déjà observable que toute fonction binaire pouvait être ainsi réalisé par des réseaux de TLU (une version de ceux-ci sont les portes logiques si chères aux débutants programmeurs, ingénieurs ou joueurs de Minecraft).

2.1.1 Le perceptron, premier important réseau de neurones

L'un des premiers réseaux de neurones fut le perceptron, réalisé dans les années 1950 par Franck Rosenblatt, basé sur des fonctions de prédictabilité linéaires, utilisant un set de variables et de coefficients généralement utilisés dans la régression linéaire.[10]

$$f(x) = \begin{cases} 1 & \text{si } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}, w \cdot x = \sum_{i=0}^{m} w_i x_i, b \text{ étant un biais d'activation à dépasser.}$$

Le perceptron possède aussi un système d'apprentissage par pondération. Il permet au neurones d'être corrigés progressivement à mesure que le réseau est utilisé ou apprend. Il permet ainsi d'avoir un réseau évolutif et en fonction des champs d'applications, de le faire converger vers une solution globalemement efficace.

$$W_i' = W_i + \alpha(Y.X_i) \text{ où } W_i' \text{ est le poids corrigé et } \alpha \text{ le pas d'apprentissage.}$$

2.1.2 ADALINE, les biais comme seuils

ADALINE (*Adaptive Linerar Neuron*) fut une évolution du perceptron réalisé dans les années 1960 par Bernard Widrow[12] et Ted Hoff où la phase d'apprentissage était ajustée par la somme pondérées des entrées là où le perceptron corrige la sortie.

$$W_i' = W_i + \alpha(Y_t - Y)X$$

.

2.1.3 Réseaux de neurones convolutifs

Les CNN (*Convolutional Neural Networks* sont un type de réseaux inspiré du cortex visuel animal où l'image est traité par régions superposées pour un traitement local afin de détecter des motifs semblable à des bords. C'est un système dérivé du perceptron qui est aujourd'hui très employé dans la reconaissance, traitement ou triage d'images [11][9]. Ce type de réseau est notamment devenu populaire grâce à l'implémentation de *DeepDream* de Google qui offrait une visualisation des traitements réalisés par le réseau superposés à l'image originale, offrant un résultat psychédélique, parfois cauchemardesque.

3 Applications

Les réseaux de neurones ont été développés dnas l'optique de pouvoir déterminer un point commun, une constante dans un échantillon donné. La première implémentation d'un perceptron était conçu pour la reconnaissance d'image, bien qu'il n'ait pu apprendre plusieurs motifs. Sujet d'étude qui est encore d'actualité avec la reconnaissance d'images avec des réseaux plus complexes, sur plusieurs niveaux et ayant de meilleurs résultats (avec même des études pour étiquetter le contenu d'une image au pixel près[9]).

Figure 1: Avant et après un passage par DeepDream.[1]

Les réseaux de neurones trouvent aussi une application dans la biologie. Un réseau de type ADALINE fut employé dans la réduction de bruit dans des ECG (*Electrocardiography*) ou la supression du battement du cœur maternel pour n'entendre que celui de l'enfant sur une ECG d'une femme en grossesse.[4]
De par l'apprentissage continu, les réseaux peuvent ainsi progressivement donner un résutlat plus net à mesure qu'ils apprennent. Des recherches on été faites pour déterminer l'utilité des réseaux neuronaux dans l'économie, plus préciemment dans la prédiction de tendances de valeurs boursières[5].
Comme sus-cité, les réseaux de neurones ont été par exemple utilisés de manière récréative avec DeepDream[8], un projet de Alexander Mordvintsev, Christopher Olah et Mike Tyka qui permettait de visualiser ce que les couches de neurones du réseau apprenant percevait au fur et à mesure du traitement.

Part II
Projet

4 Objectif

Le projet vise à reproduire des mécaniques de génération procédurale de contenu telle que la génération de terrains ou de donjons à l'aide de réseaux de neurones. L'étude concernera sur les differents types de contenu qu'un projet de jeu peut demander telle la demande de textures ou de mondes et donner des avis sur la viabilité d'une telle technologie dans ce domaine.

4.1 Génération procédurale

La génération procédurale est un moyen de générer du contenu à l'aide de règles préétablies et d'un algorithme (d'où *procédural*) de génération. Il permet ainsi de générer des textures, des sons, des modèles 3D, des entités ou même des mondes. Des genres de jeux comme les *roguelikes* (nom inspiré du jeu *Rogue*) se basent principalement sur la génération procédurale pour générer le mondes ou donjons dans lesquels vont naviguer le joueur, rafraichissant l'expérience à chaque partie.

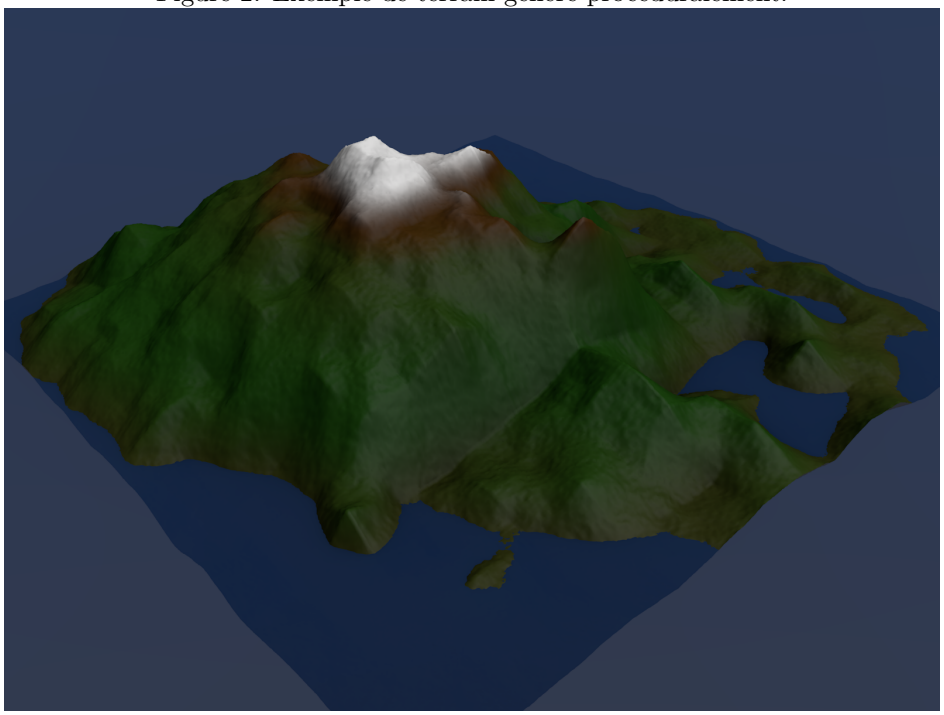
4.2 Outils et systèmes employés

Le projet se basant sur le calcul mathématique important, il a été important de trouver un système permettant de pouvoir faire un maximum de calculs sur des ressources limités. Ainsi, voici les caractéristiques de l'ordinateur qui aura servi à l'expérience:

- Ordinateur portable Asus N751JK
- Processeur : Intel i7-4710HQ (cadencé à 2.5GHz, Turbo Cache à 3.5GHz, 4 cœurs physiques pour 8 cœurs logiques)
- Mémoire vive : 8GB de RAM DDR3L cadencée à 1600MHz
- Carte Graphique : Nvidia GeForce GTX850M (utilisée avec CUDA)
- Système d'exploitation : Archlinux avec kernel ARCH 4.6.4.1 (ARCH étant la version Archlinux du kernel standard Linux.)
- Stockage : Disque dur 1TB 7200rpm

Pour le système de calcul, Torch 7 a été choisi comme base. Torch est un *framework* de calcul scientifique basé sur LuaJIT (interpréteur Lua avec compilateur à-la-volée pour optimiser les performances) offrant un environement de programmation en Lua et étant compatible avec le calcul GPGPU (*General Programming GPU*, programmation générale de carte graphique, utilisation de la carte graphique à des fins ne concernant pas le rendu graphique) avec la technologie CUDA de Nvidia pour augmenter de manière significative la vitesse d'exécution des calculs par l'architecture parallélisée des cartes graphiques. L'autre raison de l'utilisation de Torch est l'existence de librairies permettant de pouvoir lire/écrire des images ou d'indiquer la progression via une page dynamique web mise à jour accessible à distance.
Torch est accessible au moment d'écriture de ce mémoire à l'adresse suivante : `http://torch.cc`
Pour le système de réseaux neuronaux, un projet déjà existant a été choisi pour faciliter la mise en place de la recherche et la compréhension progressive du mécanisme dérrière les réseaux de neurones : dcgan.torch, dont les sources sont accessibles à l'adresse suivante : `https://github.com/soumith/dcgan.torch/`.

Figure 2: Exemple de terrain généré procéduralement.



Ce projet a été choisi car il fut le premier à fonctionner correctement avec un minimum de réglages et était fourni avec un exemple s'approchant de la demande du projet : apprentissage d'un échantillon de données et génération à partir des règles déterminées de ces échantillons.

4.2.1 Expérimentations résumés

Le projet concernera deux types de contenu généré : les niveaux de donjons dans un roguelike classique et une génération de terrain par surperposition de couches de bruit de Perlin/Simplex noise.
La génération de niveaux de roguelike est un domaine classique de la génération procédure et l'une des manifestations les plus communes de celle-ci. Datant même avant les jeux graphiques, elle fut employée dans des jeux comme *Rogue* pour pouvoir générer un grand nombre de parties différentes dans un espace mémoire et stockage limité.
La génération de terrains est un autre domaine populaire dans le développement vidéoludiques. Bien que quasiment aussi vieux que la génération de donjons, ce type de génération a surtout été rendue populaire par la montée des jeux bacs à sables tel que *Minecraft*, officiellement sorti le 11 novembre 2011 (précédemment disponible en versions *alpha* ou *beta*). Que ce soit pour un monde pseudo-infini ou sur une surface limitée, ce type de génération apporte d'autres problématiques et d'autres techniques de générations.
Ces deux types ont aussi été choisis car il fut assez rapide de mettre en place des générateurs d'échantillons pour entraîner les réseaux de neurones.

5 DCGAN

DCGAN[2] (*Deep Convolutional Generative Adversarial Network*) est une classe d'architecture de réseau de neurone recherché et conçu en janvier 2016 par Alec Radford, Luke Metz et Soumith Chintala afin d'obtenir de meilleurs résultats dans l'apprentissage de reconnaissance de motifs au travers d'échantillon pour étiquetter un contenu à partir des règles établies.

5.1 Fonctionnement

5.1.1 Falsificateurs et détectives

Un DCGAN fonctionne grâce à deux sous réseaux : un réseau qui génère des nouveaux échantillons dérivés de ceux fournis pour l'apprentissage (dit un réseau *générateur*) et un réseau qui va apprendre à déterminer et différencier un échantillon généré d'un échantillon original (un réseau *discriminant*). On peut faire un parallèle entre un falsificateur qui va créer de la fausse monnaie en observant des vrais billets et leurs détails (comme le numéro de série, les filigranes, etc...) et un détective qui va apprendre à détecter les faux billets des vrais.

5.1.2 Convolution et déconvolution

Ces deux réseaux sont composés de réseaux dits de convolutions et déconvolutions. Un réseau de convolution va, à partir de son entrée, générer un vecteur Z d'une certaine dimension censée résumer l'image en entrée. Fonctionnant part décomposition de l'image en sous-régions, chaque sous-réseau traite ainsi une partie de l'image, réagissant le plus à des motifs ressemblant à des bords. Les couches s'empilant les unes sur les autres, de telles réactions peuvent ainsi devenir de plus en plus globales. Chaque paramètre de filtre est stocké dans Z et l'apprentissage global se charge de faire évoluer progressivement le tout afin d'obtenir un résultat qui se raffine au fil des entraînements. Un réseau de déconvolution fait l'opération inverse, c'est-à-dire générer une image à partir de Z.

6 Exemple : génération de visages

Cet exemple est fourni avec la documenation de DCGAN.torch et il est intéressant de le montrer afin de voir ce à quoi l'on pourrait s'attendre.

6.1 Apprentissage

Une fois un certain nombre d'échantillons généré, ils ont été utilisés en tant que set de données pour dcgan.torch avec ces paramètres:

Listing 1: Paramètres d'apprentissage

```
opt = {
   -- Type de dataset (folder == images dans un dossier)
   dataset = 'folder',
   -- Nombres d'images à apprendre
   batchSize = 64,
   -- Inconnyu
   loadSize = 96,
   -- Taille des images produites
   fineSize = 64,
   -- Champs inconnus
   nz = 100,               -- #  of dim for Z
   ngf = 64,               -- #  of gen filters in first conv layer
   ndf = 64,               -- #  of discrim filters in first conv layer
   nThreads = 4,           -- #  of data loading threads to use
   -- Nombre d'epoch (cycles d'apprentissage) à réaliser.
   niter = 15,             -- #  of iter at starting learning rate
```

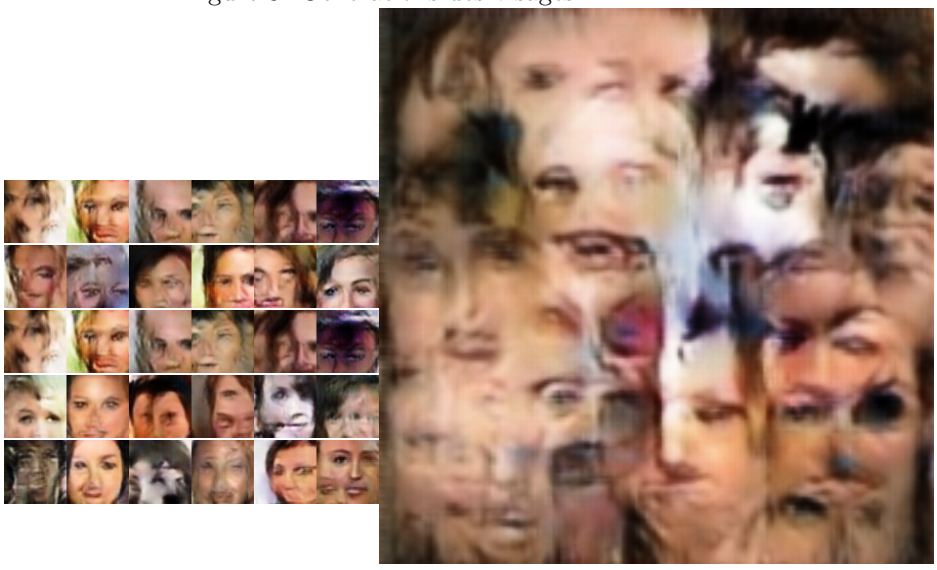
```
    -- Taux d'apprentissage (inconnu, variable potentiellement intéressante à exploiter)
    lr = 0.0002,            -- initial learning rate for adam
    -- Taux d'apprentissage (inconnu, variable potentiellement intéressante à exploiter)
    beta1 = 0.5,            -- momentum term of adam
    -- Tyoe de bruit utilisé (inconnu, variable potentiellement intéressante à exploiter)
    noise = 'normal',       -- uniform / normal

    -- Variable de paramètres pour le moteur et non l'apprentissage
    ntrain = math.huge,     -- #  of examples per epoch. math.huge for full dataset
    display = 1,            -- display samples while training. 0 = false
    display_id = 10,        -- display window id.
    -- CUDAAAAA
    gpu = 1,                -- gpu = 0 is CPU mode. gpu=X is GPU mode on GPU X
    name = 'experiment1',
}
```

6.1.1 Résultat

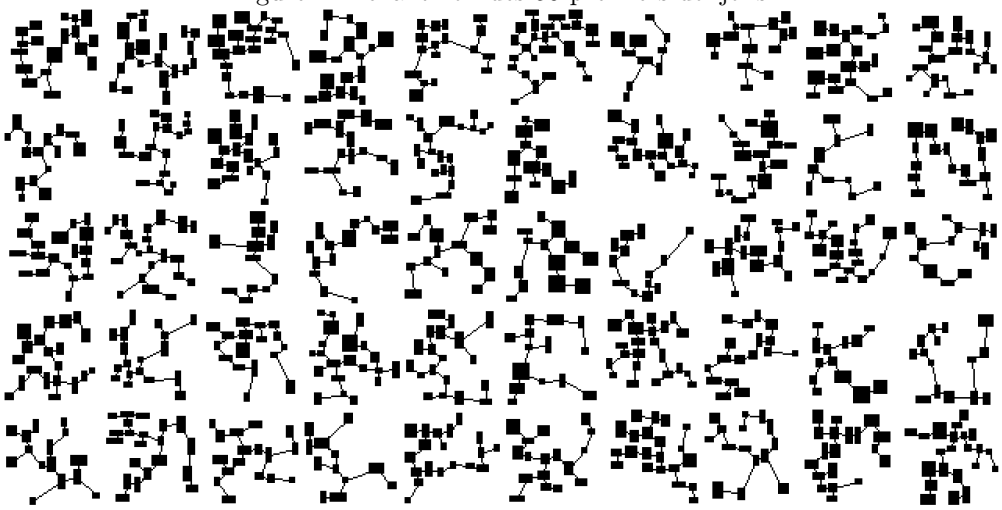
Chaque résulat représentera les mêmes epochs que l'échantillon avec le paramètres par défaut, c'est-à-dire epochs N°5, 10, 15, 20 et 25 à raison de 6 images par epoch. Cela montrera une partie de la variété des résultats possibles. Il a été rajouté aussi une génératon de l'époch 25 où la taille de l'image est plus grande. Le résultat se passe de commentaires.

Figure 3: Générations des visages



Génération avec une taille d'image plus grande ($imsize = 10$)

Figure 4: Échantillon des 50 premiers donjons.



7 Génération de donjons

Cette exépérience vise à déterminer si il est possible de générer des étages de donjons pour des jeux tel que des roguelikes. Il ne s'agit pas de générer un étage complet et peuplé d'objets et créatures mais de générer la géométrie et vérifier si elle reproduit les règles pré-déterminées

7.1 Génération des échantillons

Dans le cadre de l'expérience, j'ai crée un algorithme permettant de générer des dispositions d'étages de donjons selon un procédé assez simple:

1. Générer un ensemble de rectangles représentant les salles aléatoirement autour du centre de la région.
2. Tant que des salles se croisent, pour chaque salle S, appliquer une force de déplacement en fonction de la position des salles qui croisent S.
3. Générer un arbre couvrant minimum contenant les centres de chaque salle
4. Creuser les salles ainsi que les corridors en suivant chaque arête.

7.1.1 Génération des salles

La génération des salles se fait de manière aléatoire avec des contraintes : leur dimension est bornée et elles sont placées dans un cercle autout du centre.

Listing 2: Génération des salles

```lua
function roundm(n, m) return math.floor(((n + m - 1)/m))*m end
-- ...
function getRandomPointInCircle(radius)
  local t = 2*math.pi*math.random()
  local u = math.random() + math.random()
  local r = nil
  if u > 1 then r = 2 - u else r = u end
  return radius*r*math.cos(t), radius*r*math.sin(t)
```

```lua
end
-- ...
local rooms = {}
for i=1, Generator.NB_ROOMS do
        width,height = roundm(math.pow(math.random(), 2)*10 + 2, 1),
            roundm(math.pow(math.random(), 2)*10 + 2, 1)
        local x, y = getRandomPointInCircle(16 - math.max(width, height))
        x = x + Generator.DIMENSION/2
        y = y + Generator.DIMENSION/2
        local r = Room(x, y, width, height)
        table.insert(rooms, r)
end
-- ...
```

7.1.2 Placement des salles

Une fois les salles placées, on assigne à chaque salle des variables de velocité et on calcule pour chacune d'entre une somme de la différence de position entre la salle choisie et toutes les salles qui l'intersecte. C'est un comportement inspiré des mouvement de foules (telle que le *steering behaviour*).

Listing 3: Placement des salles

```lua
-- ...
-- Repulsion calculation
for a_i, a in pairs(rooms) do
  a.vx, a.vy = 0,0
  for b_i, b in pairs(rooms) do
    if (a_i ~= b_i) and a:intersects(b) then
      --Distance calculation.
      local dx = a.x - b.x
      local dy = a.y - b.y
      local l = math.sqrt(dx*dx + dy*dy)
      a.vx = a.vx + dx/l
      a.vy = a.vy + dy/l
      break
    end
  end
end

-- Repulsion applicaiton and room carving
for _, room in pairs(rooms) do
  local l = math.sqrt(room.vx*room.vx + room.vy*room.vy)
  -- Basic divide by zero check.
  if l >= 0.01 then
    room.x = room.x + room.vx/l
    room.y = room.y + room.vy/l
  end
end
-- ...
```

Notez aussi qu'il y a déjà un tri des salles qui se passent durant cette phase : toute salle qui se retrouve en à croiser les limites ou se retrouve dehors des dimensions est retirée de la génération. Cela permet d'enlever une bonne partie des calculs.

Listing 4: Supression des salles superflues

```
    -- triming out of bounds rooms
    for i=#rooms,1,-1 do
       local room = rooms[i]
        if room.x < 1 or room.y < 1 or room.x >= Generator.DIMENSION-room.
           width-1 or room.y >= Generator.DIMENSION-room.height-1 then
         table.remove(rooms, i)
       end
    end
```

Une amélioration possible serait d'ajouter un *quadtree* (arbre à quadrants) permettant de réduire de manière drastique le nombre de détections par salle à faire. Ici, le prototype fonctionnant avec une vitesse désirable, il n'a pas été jugé utile d'investir du temps dans un tel rajout.
Une dernière étape permet de ne conserver que les plus grandes salles (où largeur et hauteur sont tous deux supérieurs ou égaux à 4 cases). Il aurait pu être aussi intéressant de conserver toutes les salles mais le résultat aurait pu être très chargé.

7.1.3 Génération de l'arbre couvrant

Afin de générer l'arbre couvrant minimum des centres des salles, j'ai choisi d'abord de générer un graphe maillage avec l'algorithme de Delaunay afain d'obtenir un graphe avec des arêtes qui ne se croisent pas et qui ne relient que deux salles géométriquement relativement proches.

Listing 5: Génératon de la triangulation de Delaunay

```
local triangles = Delaunay.triangulate(unpack(centers))
local edges = {}
-- Triangle to edge conversion
for i, triangle in ipairs(triangles) do
  for _,edge in pairs({triangle.e1, triangle.e2, triangle.e3}) do
    local found = false
    for _,v in pairs(edges) do
      if v == edge then
        found = true
        break
      end
    end
    if not found then
      table.insert(edges, edge)
    end
  end
end
```

Note : La librairie ne générant que des triangles, il a fallu d'abord en extraire les côtés. Une fois ce graphe généré, on génère l'arbre couvrant minimum à l'aide de l'algorithme de Prim en se servant de la longueur

euclidienne d'une arête comme poids. Cela permet de ne pas avoir de cycles dans la disposition (bien qu'il puisse être intéressant dans un jeu d'en avoir pour éviter les aller-retours dans les mêmes salles).

Listing 6: Génératon de l'arbre couvrant minimum

```
local graph = Graph(centers, edges)
local res = graph
-- There was an error when the number of edges <= 3.
if #edges > 3 then
  res = MST.search(graph,
    function(edge)
      local dx = edge.p1.x - edge.p2.x
      local dy = edge.p1.y - edge.p2.y
      return math.sqrt(dx*dx+dy*dy)
    end
    )
end
```

7.1.4 Création du niveau

Pour finir, une fois cet arbre généré, les corridors reliant chaque salle sont creusés avec l'algorithme de Bresenham, classique mais efficace. Cela permet d'avoir plus facilement un résultat et d'avoir moins de mauvaises surprises car je n'avais pas d'autre solutions valables en tête; le simple *random backtracking* (combinaison entre marcheur aléatoire et retour-sur-trace permettant éventuellement de générer des corridors sineux) pouvait éventuellement bloquer la génération d'autres corridors dans certaines configurations.

Listing 7: Génération de la carte

```
-- Digging the rooms
local m = Map(Generator.DIMENSION, Generator.DIMENSION)
for _, r in pairs(biggest_rooms) do
  m:carve(r)
end

-- Drawing the edges
for _,edge in pairs(res.edges) do
  -- There were some edges linking to nothing, so there's a check to see
     if the edge links two room
  local vertices_ok = 0
  for _,c in pairs(centers) do
    if edge.p1.id ==c.id or edge.p2.id == c.id then
      vertices_ok = vertices_ok + 1
    end
  end
  if vertices_ok == 2 then
    -- Bresenham is probably one of my most used algorithms and one of my
       favorites.
    Bresenham.los(math.floor(edge.p1.x), math.floor(edge.p1.y), math.floor
       (edge.p2.x), math.floor(edge.p2.y),
```

```
19        function(x, y)
20          m.tiles[y][x] = 0
21          return true
22        end
23        )
24    end
25  end
26  return m
```

7.2 Procédure

7.2.1 Apprentissage

On va utiliser les mêmes paramètres et variations que l'expérience de la génération des visages. Même note que l'expérience précédente sur le paquet `display` ainsi que la lourdeur du processus sur la machine.

7.2.2 Résultat

Chaque résulat représentera les mêmes epochs que l'échantillon avec le paramètres par défaut, c'est-à-dire epochs N°5, 10, 15, 20 et 25 à raison de 6 images par epoch. Cela montrera une partie de la variété des résultats possibles.
Note : L'apprentissage, se basant sur des paramètres aléatoires non surveillés a bien de cas où elle tombe sur un système générant un résultat incorrect. Ceci est probablement dû à l'utilisation d'un réseau de neurone conçu pour la génération à partir de photos (aux différences de valeurs entre pixels petites à moyennes) et non à partir de ces échantillons (différence de valeur nette).

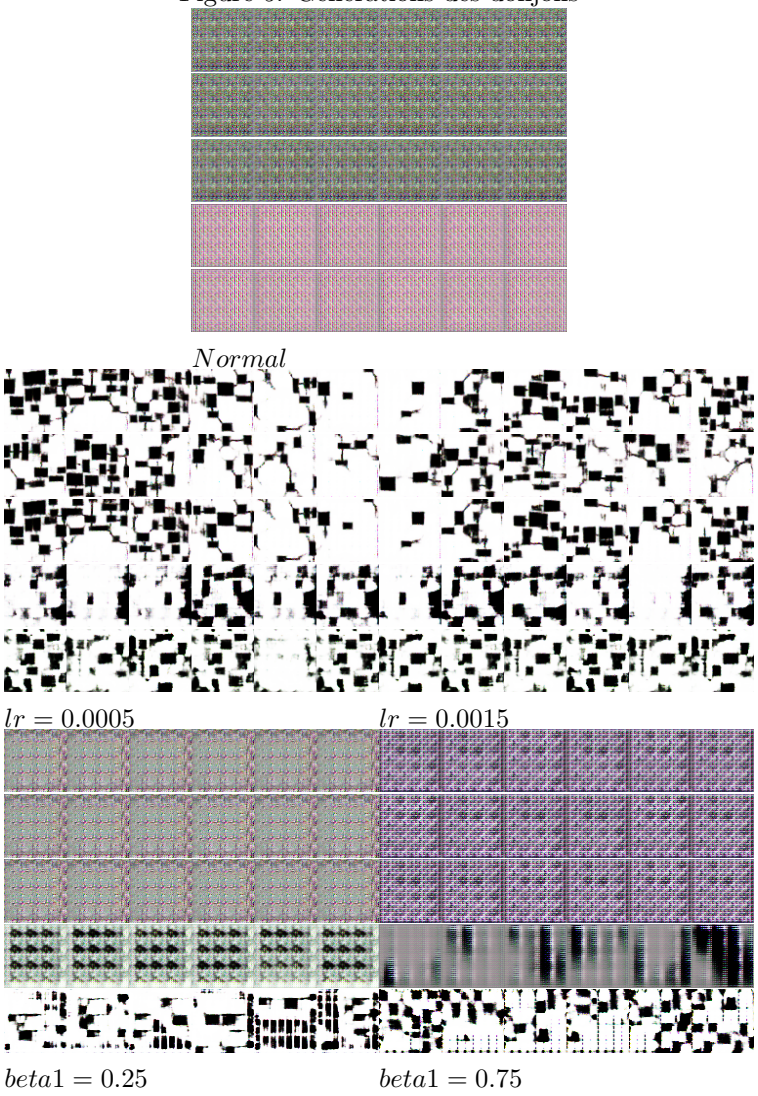
7.2.3 Interprétation

Il semblerait que par l'utilisation du sigmoïde, nécessaire pour tirer pleinement profit de la flexibilité des réseaux de neurones et étant base de la logique floue et de l'apprentissage par filtres convolutifs, DCGAN soit maladapté pour la génération procédurale à valeurs nettes. Dans certaines conditions, les produits générés après un long apprentissage ressemblent visuellement à des éléments venant des échantillions mais en aucun cas ressemblent à un matériel exploitable, les salles ne sont pas liées de manière cohérente et leurs limites sont floues. Il y a aussi un souci lors de l'apprentissage : il y a un fort taux d'échec où les productions générées ne sont que du bruit coloré, ce qui n'est absolument pas le résultat désiré. Une autre observation montre que il a été bien plus difficile de générer un réseau capable de générer un résultat convenable quand le coefficient *beta* a été manipulé. Une dernière observation aura été faite durant l'apprentissage de DCGAN : quand le réseau convergeait vers un état où $Err_G = 27.6310 Err_D = 0.0006$, il était très probable que le réseau n'évolue plus durant tout le reste de la session, la raison étant inconnue (cela pourrait être venant d'un bug dans le script, un souci avec la librarie de Cuda, l'alignement des planètes, etc...) mais des retournements de situations ont été rencontrés avec les variations de $beta1$ où le réseau est finalement sorti de ce souci et a commencé à générer de vrais résultats tards dans l'apprentissage.
En résumé, c'est un résultat semi-attendu que l'on observe : une génération chaotique de salles et corridors déformés auquel il faudrait un second traitement, automatique ou humain pour corriger ces productions.

7.2.4 Extra

DCGAN.torch m'offre la possiblité de générer des images plus grandes que la dimension des échantillons, regardons donc à quoi ressemble le produit de la génération avec un *imsize* (taille de l'image) à 10 pour toutes les générations à l'epoch 15.

Figure 5: Générations des donjons



8 Génération de terrains

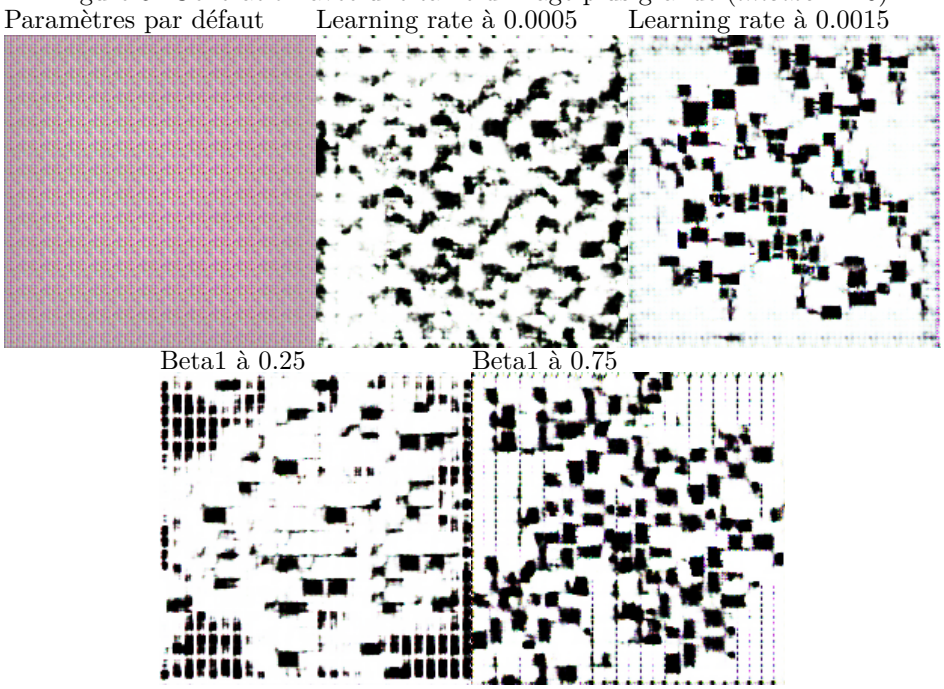
Cette exépérience vise à déterminer si il est possible de générer des textures terrains pour des jeux tel que des STR (*jeux de Stratégie en Temps Réel*) ou des jeux bac à sable.

8.1 Génération des échantillons

Dans le cadre de l'expérience, j'ai crée un script permettant de générer des terrains inspirés des genres de jeux précédents:
Un terrain ne sera défini ici que par la hauteur, un pixel noir représentant la hauteur minimale et blanc la hauteur maximale. Ce choix a été considéré car étant la base de bon nombre d'algorithmes utilisés dans différents logiciels ou jeux qui vont aussi implémenter par exemple la corrosion des sols par les vents, l'humiditié, les biomes, les différentes structures géographiques ou même les marques faites par des civilisations.

Listing 8: Paramètres des couches de bruit

Figure 6: Génération avec une taille d'image plus grande ($imsize = 10$)



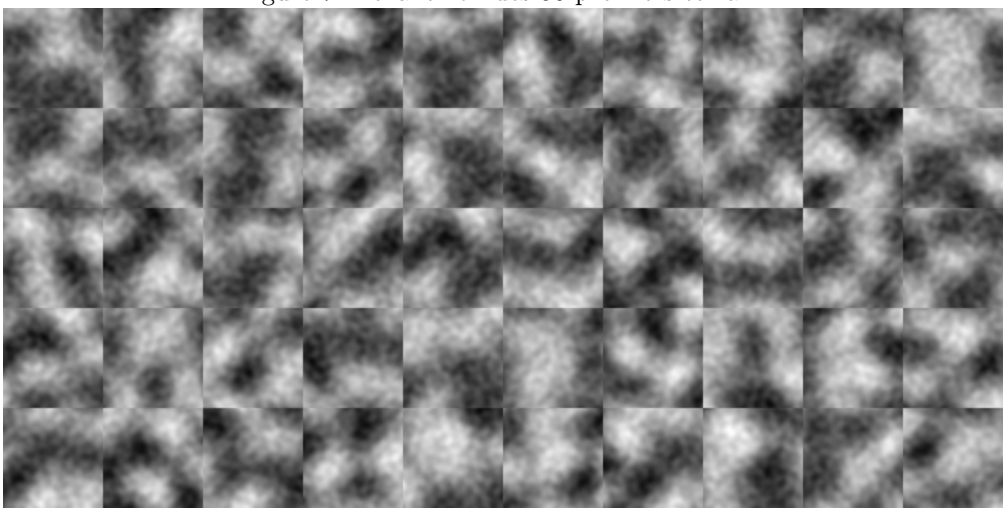

```
        passes = {
                -- Taille du bruit
                {scale=v2d(0.02, 0.02),
                -- Taille du décalage des positions
                offset_scale=v2d(100, 100),
                -- Coefficient de pondération (sur la somme de tous les
                    coefficients)
                factor = 1},
                {scale = v2d(0.1, 0.1),
                offset_scale = v2d(1000, 1000),
                factor = 0.2},
        }
```

Pour ce faire, j'ai repris une implémentation du Simplex Noise, remplacement du Noise Perlin pour une génération de bruit cohérent plus efficace et avec moins d'artéfacts directionnels.[6] J'ai additionné de manière pondérée deux couches de bruit afin de reproduire de façon simpliste une géographie.

Listing 9: Composition des couches de bruit

```
for i=0,Generator.DIMENSION-1 do
        for j=0,Generator.DIMENSION-1 do
                local pos = {x=j,y=i}
                map[i][j] = 0
                for _,pass in pairs(Generator.passes) do
                        map[i][j] =  map[i][j] + noise(pos, pass.scale,
                            offsets[_]) * pass.factor
                end
```

Figure 7: Échantillon des 50 premiers terrain.



```
8                   map[i][j] = map[i][j] / sum_of_factors
9               end
10          end
```

Note : Ici, par manque de temps, les échantillons sont très basiques et manquent de spécificités comme ceux listés précédemment. Il aurait pu être intéressant d'étudier l'implication de ces rajouts mais le sujet de ce mémoire ne concernait au départ que la génération procédurale de donjon. En fait, la génération de terrain était une issue de secours au cas où l'apprentissage ne produisait que du bruit... Désolé.

8.2 Procédure

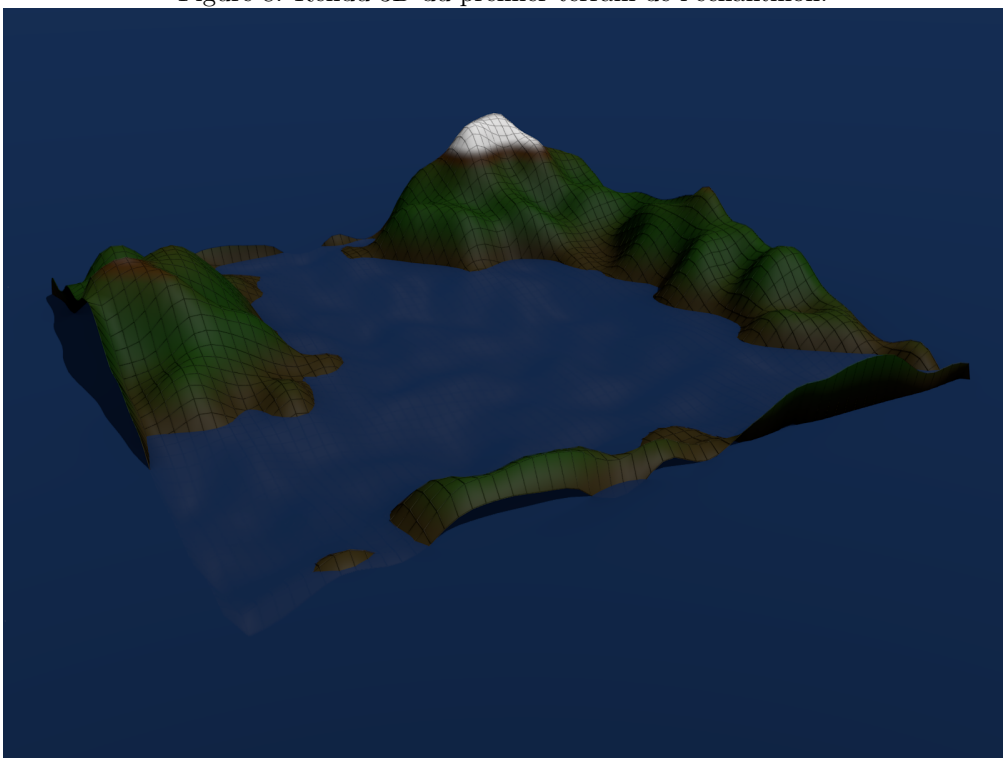
8.2.1 Apprentissage

On va utiliser les mêmes paramètres et variations que l'expérience de la génération des visages. Torch offre la possiblité de pouvoir monitorer les résultats de l'apprentissage via le paquet `display` qui ouvre un serveur HTTP qui permet d'observer en temps réel les échantillons traités et le résultat temporaire de l'apprentissage. Il est extrêmement pratique d'utiliser ce paquet car l'utilisation de la carte graphique a tendance à faire ralentique le serveur graphique au point où je me suis retrouvé à écrire des phrases sans retour visuel avant des secondes après la fin de la saisie.

8.2.2 Résultat

Bien que la quantité de donjons générés soit importants (100000 donjons), le réseau n'aura pu s'adapter aux échantillons et le résultat ne converge pas vers des résultats convenables. Les générations convergent très vite vers un résultat bruité rouge et totalement incohérent avec les produits d'origine. Chaque échantillon représentera les mêmes epochs que l'échantillon avec le paramètres par défaut, c'est-à-dire epochs N°5, 10, 15, 20 et 25.
Note : L'apprentissage, se basant sur des paramètres aléatoires a bien de cas où elle tombe sur un système générant un résultat incorrect. Ceci est probablement dû à l'utilisation d'un réseau de neurone conçu pour la génération à partir de photos (aux différences de valeurs entre pixels petites à moyennes) et non à partir de ces échantillons (différence de valeur nette).

Figure 8: Rendu 3D du premier terrain de l'échantillon.



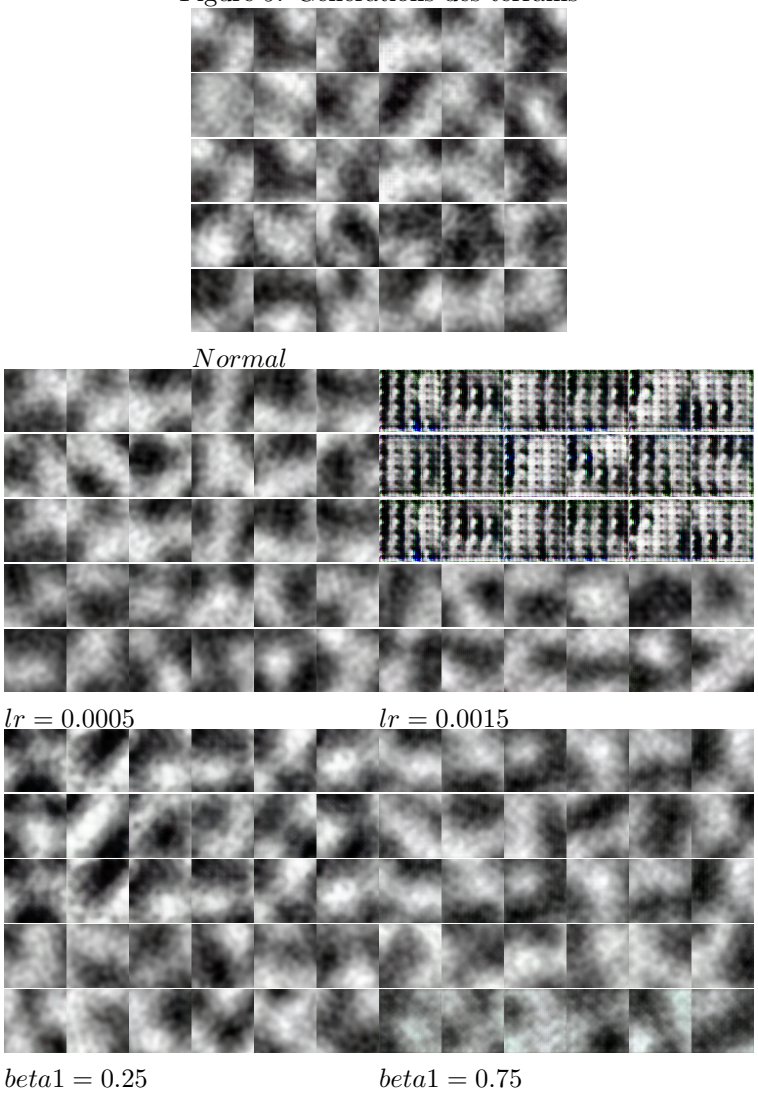
8.2.3 Interprétation

Dans les cas où l'on ne converge pas vers un résultat composé de bruit coloré, les réseaux forment assez vite un résultat assez proche des échantillons et se complexifie au fur et à mesure que les epochs se passent. De manière curieuse, il aura été plus difficile d'obtenir un résultat en modifiant le *learning rate* qu'en modifiant le *beta1* : en effet, aux cas où le taux d'apprentissage a été altéré, il y a dû avoir plus tentatives avant d'avoir un résultat qui ne convergera pas vers un résultat incorrect. L'exemple avec un *learning rate* de 0.0015 est un bon exemple de ce qui s'est passé dans une petite partie des cas : le réseau a très vite, directement avec les premiers échantillons du premier epoch, convergé vers une bouillie de pixels qui a subitement commencé à varier jusqu'à durant l'epoch 9 pour presque reconverger à l'epoch 10. Hélas, je n'ai pas pris de photos durant cet instant plein de rebondissements.
Pour résumer, on a un taux de réussite bien plus important qu'avec la génération de donjons car il y n'y aura pas ou peu de traitement post-production à réaliser pour avoir un résultat exploitable. Les réseaux, quand ne convergeant pas vers l'enfer, génèrent un résultat très vite intéressant et il serait aussi intéressant de de creuser de manière plus approfondie dans cette catégorie de génération, voir quels résultats l'on pourrait avoir avec des échantillons variés, colorés ou avec des échantillons de différents générateurs.

8.2.4 Extra

DCGAN.torch m'offre la possiblité de générer des images plus grandes que la dimension des échantillons, regardons donc à quoi ressemble le produit de la génération avec un *imsize* (taille de l'image) à 10 pour toutes les générations à l'epoch 15.
Il semblerait que les réseaux, une fois leur développement à terme, soient capables de générer un résultat convaincant. On ne discerne aucune rupture visuelle, la texture ne se répète pas, sauf pour le dernier resultat où surprenamment un motif s'est formé avec les plus petits détails de la génération. Il y a donc une possiblité pour qu'un tel système avec ce type d'échantillon et production soit utilisable.

Figure 9: Générations des terrains



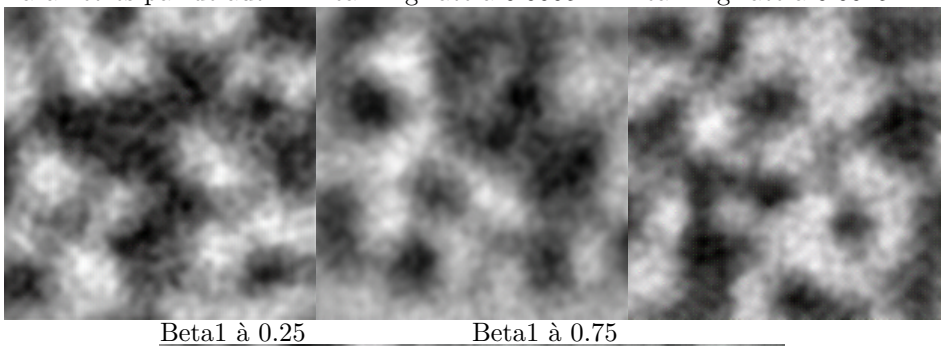
9 Mot de la fin

9.1 Conclusion

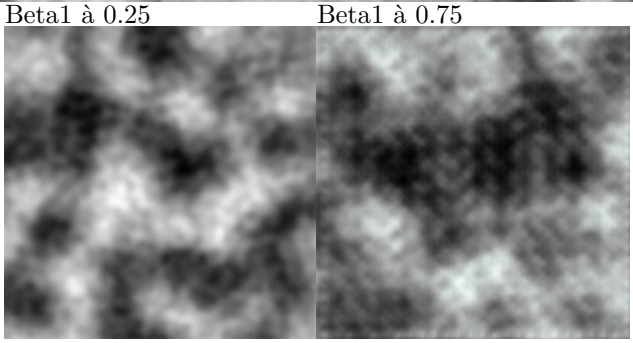
Les réseaux de neurones sont une technologie intéressante, ils offrent une flexibilité aux types d'échantillons et dans certains cas, offrir une solution viable sans avoir à développer et rechercher une solution mathématique/physique exacte. Malhereusement, l'apprentissage étant basé en partie sur l'aléatoire, il n'existe pas de moyen pour s'assurer des résultats comme peuvent montrer le compte-rendu des expérimentations. Ainsi, la génération de donjons à partir de résultats générateur aux règles bien définies amènent à des résultats n'appliquant pas les règles établies mais la génération de terrain basée sur le bruit reproduit assee bien le modèle original et est même utilisable dans des dimensions différentes.
Un autre problème qui survient de l'expérimentation est le temps investi pour une tentative : 5 heures pour 15 epochs avec un laptop mi-haut-de-gamme de janvier 2015. Le temps de calcul ne dépendant pas de la progression, c'est un temps assez linéaire contre lequel on fait face. L'utilisation d'un grand nombre d'échantillons impique l'existence de ces images ou de la présence d'un générateur déjà existant pour ce set, rendant la génération par un réseau bien moins utile (sauf eventuellement pour la reproduction). Dernière-

Figure 10: Génération avec une taille d'image plus grande ($imsize = 10$)

Paramètres par défaut Learning rate à 0.0005 Learning rate à 0.0015



Beta1 à 0.25 Beta1 à 0.75



ment, le temps de génération via le réseau est plus long dans les deux cas que par les générateurs déjà existants dans le cadre des expérieces réalisées.

9.2 Ouvertures

Bien entendu, ces deux expériences ne suffisent pas pour couvrir le large champ d'application qu'offre la génération procédurale. Malheuresement, par manque de temps et de matériel puissant, d'autres expérimentations intéressantes sont passées à la trappe comme eventuellement la fusion de textures totalement différentes, la génération d'îles à géométrie globale variable ou fixe (c'est-à-dire mettre en place une géométrie et construire le relief à partir de ce canevas) dérivé de la génération de terrain ou la génération de terrain avec éléments de géographie composite (comme biomes, villes ou rivières). D'autres types de réseaux de neurones auraient pû être employés dans le cadre de l'expérimentation. DCGAN a été un choix pragmatique car c'est un type de réseau relativement moderne à l'heure de l'écriture de ce rapport (l'article datant de janvier 2016, ce rapport d'août de la même année) et offrant une mise en place *relativement* aisée et un exemple s'approchant du sujet de ce rapport.

9.3 Quelques liens

Afin de fournir preuves du travail titanesque ma carte graphique (et optionnellement mon ordinateur) a réalisé, j'héberge des archives contenant les *checkpoints*, c'est-à-dire l'état du réseau de neurones de 1 à 15 pour chaque expérimentation sur mon serveur, ainsi que les générateurs programmés pour ce projet ainsi que les échantillons générés

9.3.1 Checkpoints

Note : La variable altérée est dans le nom du fichier
Exemple Visages (basés sur l'exemple fourni avec DCGAN) : `http://retroactive.me/media/research/neural_network/checkpoints/000_faces.tar.gz`

Donjons

http://retroactive.me/media/research/neural_network/checkpoints/001_dungeon.tar.gz
http://retroactive.me/media/research/neural_network/checkpoints/003_dungeon_rate_0.0005.tar.gz
http://retroactive.me/media/research/neural_network/checkpoints/004_dungeon_rate_0.0015.tar.gz
http://retroactive.me/media/research/neural_network/checkpoints/005_dungeon_beta1_0.25.tar.gz
http://retroactive.me/media/research/neural_network/checkpoints/006_dungeon_beta1_0.75.tar.gz

Terrains

http://retroactive.me/media/research/neural_network/checkpoints/007_terrain.tar.gz
http://retroactive.me/media/research/neural_network/checkpoints/008_terrain_rate_0.0005.tar.gz
http://retroactive.me/media/research/neural_network/checkpoints/009_terrain_rate_0.0015.tar.gz
http://retroactive.me/media/research/neural_network/checkpoints/010_terrain_beta1_0.25.tar.gz
http://retroactive.me/media/research/neural_network/checkpoints/011_terrain_beta1_0.75.tar.gz

Vrai mot de la fin

Banane.

References

[1] Toi aussi prends-toi une dose d'harold. http://www.shutterstock.com/cat.mhtml?models=595771, The year is 20XX.

[2] Soumith Chintala Alec Radford, Luke Metz. Unsupervised representation learning with deep convolutional generative adversarial networks. https://arxiv.org/abs/1511.06434, 2016.

[3] J. A. Anderson and E. Rosenfeld, editors. *Neurocomputing: Foundations of Research*. The MIT Press, Cambridge, MA, 1988.

[4] Judith Dayhoff. Adaline/madaline:applications. http://www.csee.wvu.edu/classes/cpe520/presentations/ADAMADAL2.pdf.

[5] Sabatino Costanzo Loren Trigo Luis Jiménez Juan González. A neural networks model of the venezuelan economy. https://arxiv.org/pdf/0708.3463.pdf, 2007.

[6] Stefan Gustavson. Simplex noise demystified. http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf, 2005.

[7] W. S. McCulloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943. [Reprinted in [3]].

[8] Alexander Mordvintsev Christopher Olah and Mike Tyka. Deepdream - a code example for visualizing neural networks. https://web.archive.org/web/20150708233542/http://googleresearch.blogspot.co.uk/2015/07/deepdream-code-example-for-visualizing.html, 2015.

[9] Pedro O. Pinheiro and Ronan Collobert. From image-level to pixel-level labeling with convolutional networks. http://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Pinheiro_From_Image-Level_to_2015_CVPR_paper.pdf, 2015.

[10] F. Rosenblatt. The perceptron—a perceiving and recognizing automaton. Report 85-460-1, Cornell Aeronautical Laboratory, 1957.

[11] Alexey Dosovitskiy Jost Tobias Springenberg Maxim Tatarchenko and Thomas Brox. Learning to generate chairs tables and cars with convolutional networks. https://arxiv.org/abs/1411.5928, 2014.

[12] Bernard Widrow. An adaptive "adaline" neuron using chemical "memistors". http://www-isl.stanford.edu/~widrow/papers/t1960anadaptive.pdf, 1960.