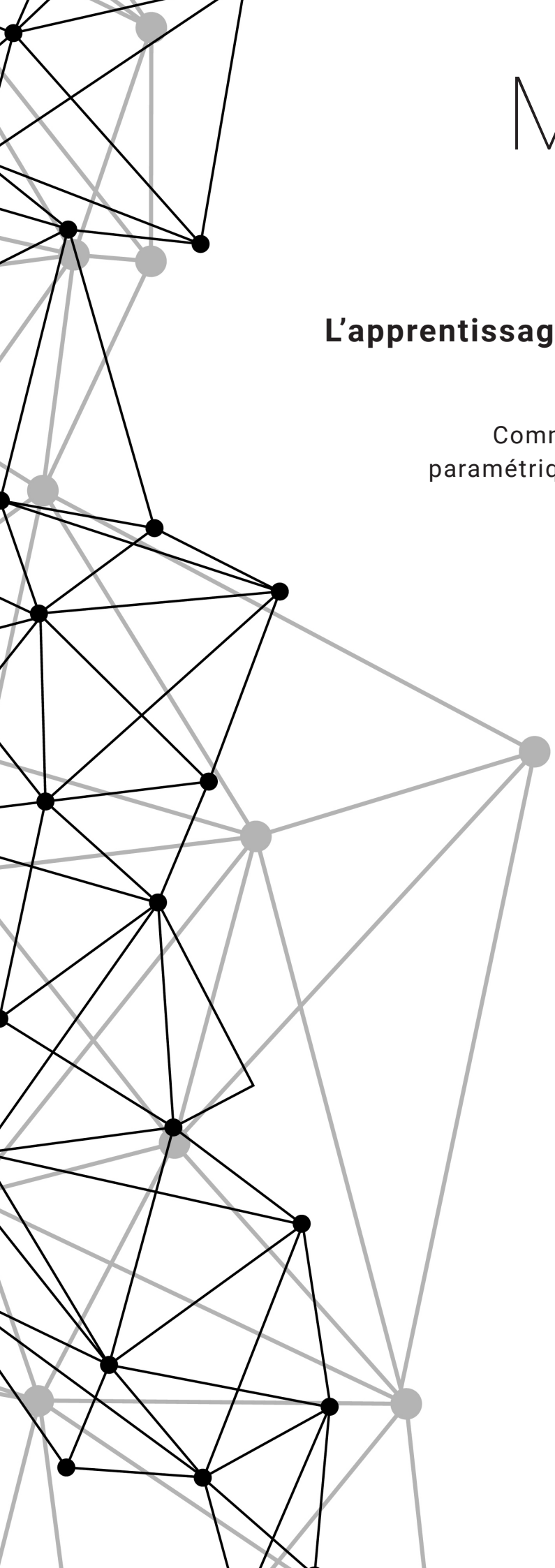


# MÉMOIRE

par Jean-Raphaël **Piquard**

## L'apprentissage machine au service de la conception architecturale

Comment enrichir un espace de solution paramétrique grâce aux réseaux antagonistes génératifs ?

Séminaire Activités et Instrumentation de la conception

Encadré par François Guéna, Joaquim Silvestre et Anne Tüscher

ENSA Paris la Villette, 2020

ensa paris la villette

# MÉMOIRE DE MASTER

par Jean-Raphaël Piquard

## L'apprentissage machine au service de la conception architecturale

Comment enrichir un espace de solution paramétrique grâce aux réseaux antagonistes génératifs ?

Séminaire Activités et Instrumentation de la conception

Encadré par François Guéna, Joaquim Silvestre et Anne Tüscher

•

ENSA Paris la Villette, 2020
Contact : jrpiquard@gmail.com

ensa paris la villette

# REMERCIEMENTS

Je souhaite avant tout remercier les professeurs qui m'ont accompagné durant ce travail, Joaquim Silvestre, François Guéna et Anne Tüscher. Leurs apports ont été indispensables, et générateur de tout ce qui a pu être réalisé ici.

Je voudrais également remercier ceux qui m'ont soutenu durant cette période difficile. Mes parents Jean-Pascal et Isabelle Piquard, mais également Alizée, Nicolas et Françoise Gauthier pour leur confiance et leurs encouragements.

# RÉSUMÉ

Dans un contexte de déploiement de technologies de plus en plus lourdes dans le milieu architectural, ce mémoire touche à une des plus énigmatiques et magiques d'entre elles, l'apprentissage machine. Reprenant les logiques d'apprentissage et d'adaptation du cerveau de l'Homme, cet outil présente un réel potentiel pour l'architecte comme pour l'architecture.

Malgré des avantages évidents, plusieurs limites justifient que son intégration soit encore difficile, voire même controversée. Tout d'abord d'un point de vue fonctionnel, car l'apprentissage machine requiert uneww grande quantité de données pour fonctionner correctement, et le contexte actuel de la donnée en architecture est loin du libre-échange.

La question de l'éthique se pose également. Plus un outil devient puissant, plus la place de celui qui l'utilise se voit transformée, voir fragilisée. Garder le contrôle est essentiel, afin de préserver l'essence de l'architecture et de ses apôtres.

Ce mémoire approche ainsi une manière de recourir à l'apprentissage machine dans le processus de conception, par le biais de la création de données et de leur enrichissement via des réseaux de neurones antagonistes génératifs. Le système avancé est ouvert, et prêt à se nourrir des évolutions impétueuses de la technologie, de la société, et surtout de l'architecture.

# SOMMAIRE

# 0
# AVANT-PROPOS

Le domaine de l'architecture, tout comme de nombreux domaines du monde que nous connaissons, est fortement affecté par les avancées technologiques et sociales de la société. Bien que l'on observe un décalage temporel entre ces changements et la pratique de l'architecture, celle-ci s'en voit irrémédiablement impactée.

Dans le cadre plus restreint de la conception architecturale, on observe également une évolution de plus en plus rapide des outils à disposition des concepteurs. Si l'arrivée du dessin assisté par ordinateur (**DAO**) a su révolutionner la façon de concevoir l'architecture, jusqu'alors sur papier depuis des centaines d'années, la révolution que nous vivons aujourd'hui se situe dans la multiplication des données tout au long du processus architectural, leur gestion et leur partage.

En parallèle de l'émergence de nouveaux procédés à forte influence sur la conception (architecture paramétrique, travail sur la maquette numérique dans le cadre du **BIM**…), l'explosion de la quantité d'informations et de données ouvre une nouvelle porte, celle de l'apprentissage machine, ou **Machine Learning**. C'est de cette nouvelle corrélation entre conception architecturale, automatisation des processus et apprentissage machine, qu'a été initié ce mémoire.

L'architecture paramétrique, dans l'Histoire comme dans mon cursus architectural, est un procédé nouveau et à très forte influence sur la manière de concevoir. Le dessin et/ou la modélisation sont remplacés par la réflexion et la description du processus. Une fois le modèle mis en place, il est possible d'en faire varier les paramètres et ainsi obtenir une grande quantité de solutions, toutes respectant le modèle établi. Cependant, ces solutions restent « sages », et elles ne vont pas produire des résultats surprenants ou alors difficilement. Or, la richesse de l'architecture consiste parfois à faire des erreurs, ou encore savoir s'affranchir de certaines limites : « tricher pour mieux réussir ». Cette démarche m'a mené vers l'étude d'une approche différente, qui en est encore à ses balbutiements dans le domaine architectural : l'apprentissage machine.

L'apprentissage machine, et plus précisément l'apprentissage dit « profond », est un procédé informatique issu des intelligences artificielles, qui va être capable d'apprendre par lui-même quelles que soient les données d'entrée. Procédé bien plus lourd à mettre en place et moins adapté, pour le moment, à la conception architecturale, il est cependant source d'une richesse de solutions presque infinie. La limite reste cependant le « tri » à faire dans ce trop-plein d'informations produites, mais également le besoin d'une gigantesque base de données initiale pour pouvoir apprendre correctement.

Les deux approches précédemment abordées, bien que très différentes, semblent pouvoir se compléter. Leur mise en relation évoque alors autant de nouvelles questions que de réponses, ce qui a donné lieu à ce mémoire.

Ce travail étant orienté sur des sujets informatiques parfois assez poussés, un glossaire permet de renseigner et d'accompagner la lecture. Ainsi, tous les mots repérés en **gras** dans le texte y trouveront leur définition, traduction ou explications.

En version dématérialisée le mémoire se lit en double-page comme un livre, les pages paires à gauche et impaires à droite.

# 1
# INTRODUCTION

La conception architecturale est par nature très complexe, pouvant prendre autant de formes qu'il y a d'outils pour l'accompagner, ou qu'il y a d'acteurs pour concevoir. L'enjeu de ces outils est alors considérable, et la réflexion quant à leur élaboration ne doit pas oublier ce pour quoi ils sont faits : accompagner le concepteur sans s'approprier son contrôle, et sans limiter la qualité de l'architecture produite.

Dans ce mémoire, il sera question de réfléchir à la mise en œuvre d'un processus d'aide à la conception, et plus particulièrement dans la génération d'un espace de solutions. L'idée directrice est de produire des solutions « plus riches », qui ne se cantonnent pas à l'exécution simple de paramètres entrés au préalable, c'est à dire des solutions acceptant l'imprévu, l'inattendu. En effet, l'aide à la conception devenant monnaie courante dans le domaine architectural, elle peut parfois prendre trop d'importance jusqu'à en devenir invasive, et entraîner potentiellement les résultats vers une uniformisation fâcheuse.

Pour tenter d'y apporter une solution, tout en restant dans des domaines qui me sont abordables dans le temps d'un mémoire, une stratégie d'approche possible vise à associer les avantages non négligeables du processus de conception paramétrique, avec les outils plus complexes et exigeants de l'apprentissage machine, tels que les **réseaux de neurones antagonistes génératifs** (**GAN**). J'ai ainsi la possibilité de reprendre la problématique générale déjà soulevée par un mémoire de master au sein de ce séminaire par Sophie OROS en 2019 : « l'apprentissage machine au service de la conception architecturale ? »[1].

[1] **Sophie Oros**, « L'apprentissage machine au service de la conception architecturale ? Une application : extraire des informations à partir de plans », Mémoire de master dans le cadre du séminaire Activités et Instrumentation de la Conception, ENSA de Paris la Villette, 2019.

Nous verrons ainsi dans une première partie le contexte actuel dans lequel s'inscrivent les grandes notions de conception architecturale et son automatisation, ainsi que les débats et questions souvent posées. Il s'agira ensuite d'inscrire la problématique du mémoire au sein de ce contexte, et de proposer un premier protocole expérimental afin d'y apporter des réponses.

Dans une seconde partie, il conviendra d'aborder l'état de l'art, qui développera d'une part l'état des connaissances, soit une approche pédagogique des différents concepts et notions qui seront abordés par la suite dans le mémoire ; et reprendra d'autre part l'état de la recherche, c'est-à-dire les pistes déjà explorées au croisement des domaines de l'apprentissage machine et de la conception architecturale.

La troisième partie sera quant à elle dédiée à l'application des éléments exposés précédemment au travers d'expérimentations. Après une présentation des logiciels requis et du processus expérimental plus détaillé, je pourrai m'attaquer aux expériences en elles-mêmes, tout d'abord sur un « témoin » caractérisé par le thème des chiffres de 0 à 9, puis dans un second temps sur des formes et enfin sur un sujet plus architectural : des dispositions de bâtiments. Suite à ces expériences, une première analyse des résultats sera effectuée.

Enfin, il s'agira d'analyser de manière plus critique les résultats obtenus à partir de ces expériences, puis d'en extraire les différentes pistes d'améliorations possibles du processus proposé, à l'échelle du détail mais aussi du principe, et pour finir d'explorer les évolutions envisageables du mémoire sur d'autres sujets en lien avec la problématique générale.

# 2
# PROBLÉMATIQUE

## 2.1 Le contexte

L'architecture, comme la plupart des professions, est fortement liée au contexte de la société, et voit sa pratique évoluer au rythme des changements politiques, économiques, sociaux ou environnementaux. Ces changements sont souvent imbriqués, et analyser un critère unique nécessite un certain recul. Aussi, au sein de ce chapitre, c'est au travers du spectre technologique que je vais observer le domaine architectural. Il faut donc garder à l'esprit que malgré les secousses provoquées par les nouvelles avancées, la mouvance architecturale reste complexe et liée à bien d'autres phénomènes.

Stanislas Chaillou propose à cet effet 4 grandes temporalités[1], du 20ème siècle à nos jours. La première, dans les années 30, ne relève pas de la technologie, mais plutôt d'une réflexion qui tend vers plus de systématisation dans la conception architecturale.

La seconde est cette fois plus technique et liée à l'avènement de la puissance informatique jusque dans les années 80. *« En 1959, le professeur Patrick Hanratty déploie PRONTO, le premier prototype de logiciel de **DAO** (Dessin Assisté par Ordinateur), conçu pour la conception de pièces d'ingénierie. Les possibilités offertes par ce logiciel, associées à l'évolution rapide de la puissance de calcul, déclenchent alors une réflexion de fond au sein de la communauté des architectes. »*[1]

La troisième temporalité est beaucoup plus actuelle : le paramétrisme. Là où les évolutions précédentes sont majoritairement intégrées au processus de pratique et d'enseignement de l'architecture, le paramétrisme reste encore quelque chose de nouveau, bien que présent dans les esprits. Cette nouvelle approche faisant appel aux paramètres des formes architecturales plutôt qu'aux formes en elles-mêmes change la façon de concevoir, et en est devenue la stratégie d'approche principale de certains architectes.

Au-delà de l'aspect formel, la paramétrisation est également à l'origine d'une révolution encore une fois très actuelle, dont les enjeux font beaucoup débat : le **BIM** (Building Information Modeling).

---

[1] **Stanislas Chaillou,** « L'IA en Architecture, Séquence d'une Alliance », Article du site Medium, 5 octobre 2019, consulté le 30/12/2020. Disponible sur : https://medium.com/@sfjchaillou/lia-en-architecture-c805651d7f42.

Cette nouvelle manière de travailler, en coopération entre acteurs par le biais d'une maquette numérique comprenant tous les paramètres du projet, est toute nouvelle dans la pratique et encore plus dans l'enseignement. En tant qu'élève, j'ai pu observer que cette « révolution » technique n'est pas toujours bien reçue, preuve que tout changement se fait parfois aux dépens de certains.

Alors que le paramétrisme est en cours d'adoption, ou d'intégration plus ou moins voulue, une dernière temporalité se prépare : l'intelligence artificielle. Cette « révolution » est d'autant plus primordiale qu'elle a déjà eu lieu dans d'autres secteurs, et les transformations qui en ont résulté sont difficiles à ignorer. Le moteur de recherche le plus puissant est incontestablement Google, qui utilise pleinement les intelligences artificielles depuis maintenant des années. Tesla est devenu en l'espace de quelques années le principal constructeur de véhicules électriques devant des marques présentes depuis plus d'un siècle, mais n'exploitant pas cette technologie.

Là où le paramétrisme peut être considéré comme un « plus », un choix que l'on peut faire pour être assisté dans son travail, les technologies d'intelligence artificielle ont la "méchante" habitude de tout raser sur leur passage, de transformer le domaine impacté. Que nous la voyions comme un danger ou une révolution nécessaire, cette quatrième temporalité de l'intelligence de la machine est à comprendre et à anticiper.

Même si j'essaye de garder un certain recul, certains considèrent déjà cette révolution comme indéniable, et presque acquise :

*« L'intelligence artificielle va bouleverser les métiers du bâtiment et de l'immobilier. Dans le domaine de l'architecture, elle ouvre de nouvelles frontières avec la conception de formes jusqu'alors inexplorées. Les tâches répétitives où la valeur ajoutée de l'architecte est la plus faible seront traitées par des machines, laissant le professionnel se concentrer sur le cœur de sa mission : l'arbitrage des contraintes et la synthèse des solutions. Des garde-fous seront toutefois nécessaires pour protéger les droits d'auteur et la propriété intellectuelle des créateurs.* »[2]

[2] **Stéphane Lutard,** « Architecture et intelligence artificielle », Cahiers de la profession n°70, 2e trimestre 2020, page 21.

Cet article n'est pas anodin : il fait partie du magazine *« Cahiers de la profession »*, conçu par et pour des architectes. Ce n'est donc pas l'avis d'un ingénieur ou d'un informaticien baignant dans l'informatique, c'est bel et bien un constat émanant du monde de l'architecture.

Ce point de vue fataliste est cependant bien fondé. Le monde actuel s'oriente de plus en plus autour des « données ». Tout ce que l'on fait produit de jour en jour toujours plus d'informations. L'architecture est de plus en plus renseignée, informatisée et paramétrisée : elle devient un terrain plus que propice à la révolution de l'intelligence artificielle. Au-delà de se demander s'il s'agit d'une bonne chose ou non, ou encore si on y sera confronté de force, il faut surtout se préparer à l'appréhender, la contrôler et potentiellement l'intégrer de la meilleure manière qu'il soit dans le respect de la production architecturale et des concepteurs.

Les pires tremblements de terre ne sont pas les plus puissants, mais les moins anticipés.

## 2.2 La problématique

Dans un contexte d'appréhension d'une nouvelle technologie aux potentiels aussi riches que dangereux, j'ai cherché un angle d'attaque alliant contrôle et apport au concepteur.

Il est certain que les intelligences artificielles et le **Machine Learning** sont, ou finiront par être, une source de solutions architecturales préétablies. La proximité de fonctionnement entre le cerveau humain et les réseaux de neurones est très prometteuse en termes de complexité de réponse, mais c'est ailleurs que j'aimerais trouver une richesse, dans les erreurs. Tout comme le cerveau, les réseaux de neurones génératifs produisent des résultats inattendus, parfois tellement hors des clous qu'ils sont inexploitables, parfois intéressants par le biais de leur manquement aux règles.

A ce titre, Caroline Quentin évoquait dans la série The World's Most Extraordinary Homes : *« J'aime l'idée que la créativité vienne d'erreurs ou de choses que l'on n'avait pas prévues »*[3]

---

[3]**Caroline Quentin,** « The World's Most Extraordinary Homes, Sous-terre », Saison 1 Episode 4, BBC Two, 2017.

Ces paroles prennent d'autant plus de sens qu'elles sont prononcées par une personne qui n'est pas concepteur, mais actrice. Son regard est extérieur, et exempt de toute pratique.

Si les réseaux de neurones sont sources d'erreurs productives, comment les exploiter ? Comment orienter le système pour contrôler sa production ?

J'ai trouvé réponse à ces questionnements dans le fonctionnement même de ces réseaux de neurones. Le type d'**algorithme** qui m'intéressait (les fameux réseaux de neurones antagonistes génératifs, que je développerai dans l'état de l'art) avait besoin d'une base de données pour fonctionner et apprendre, pour ensuite créer à son tour des résultats. La meilleure manière d'observer des erreurs étant la comparaison, la solution devient claire : créer soi-même une base de données, la fournir au réseau de neurones, et analyser les résultats en ayant en tête la base de données initiale. Plus que des erreurs, je pourrais alors analyser finement la production de l'**algorithme**, ses réussites, ses échecs, et ses surprises.

Les nouveaux résultats produits seraient alors différents, plus nombreux, et peut-être plus riches ? Ainsi mon processus de réflexion aboutit à cette problématique :

*« Comment enrichir un espace de solution paramétrique grâce aux réseaux antagonistes génératifs ? »*.

Bien que le processus global soit établi, à savoir créer des données puis les donner à un réseau de neurones, la manière de le faire reste floue et justifie l'emploi du « comment ». Quels logiciels ? Quelles données ? En quel format et quel type ? Combien d'essais ? De quoi est fait un essai ? La moindre étape va-t-elle aboutir à quoi que ce soit ?

Ce mémoire a pour but de résoudre toutes ces zones d'ombre, et proposer une réponse aussi complète que possible à la problématique soulevée.

Pour les plus impatients, l'entièreté des réponses se trouve en page 142. Mais le plaisir résidant dans l'attente, je vous propose d'apprécier les 141 pages précédentes.

2.3 Logique expérimentale

Enseignée dès le collège, la **démarche scientifique** est une méthode permettant de garantir la qualité de la production de connaissances scientifiques, mais également d'appréhender le monde. Son principe se veut universel, étant abordable par les scientifiques tout comme le grand public, et s'adressant à toute discipline.

*« La science est un processus autocorrectif : elle contient en elle-même tous les procédés qui permettent d'affirmer ou de réfuter une théorie. En effet, une théorie ne respecte la méthode scientifique que si celle-ci est testable par l'expérience. Après la formulation d'une théorie vient nécessairement une partie de confirmation (ou infirmation) expérimentale. »*[4]

Comme l'introduit ce support éducatif du département de physique de l'ENS, la **démarche scientifique** se scinde en plusieurs étapes, qui régissent notamment la structure de mon expérimentation, mais également du mémoire dans sa globalité. Je distingue donc la problématique, l'hypothèse, l'expérience et l'étude des résultats. Ces éléments sont ou seront abordés dans le détail, mais il convient de préciser quelques éléments clés d'une étape qui régira en grande partie ce travail de recherche : l'expérience. Bien que le protocole sera abordé dans la partie prévue à cet effet, sa construction dépendra de ces quelques principes :

1. Mettre en place une expérience témoin afin d'y comparer les résultats.

2. Ne tester qu'un paramètre à la fois : j'isole ces paramètres en plusieurs expériences ou variations d'expériences, en rendant constant les autres paramètres.

3. Répéter l'expérience dans les mêmes conditions pour s'assurer de la qualité des résultats.

[4] **Ecole Normale Supérieure**, Département de physique, Guide « La Démarche scientifique » Page 1.

Pour concrétiser un peu ces propos, il est possible de mettre ces principes en application avec une expérience très simple : la détection de la présence d'eau dans une solution avec du sulfate de cuivre anhydre.

1. Pour l'expérience témoin j'essaye le sulfate de cuivre anhydre sur de l'eau, s'il se colore bien en bleu, tout fonctionne correctement et je peux passer à la suite.

2. J'essaye une solution à la fois, et pas de mélange (du vinaigre, puis de l'huile, pas les deux en même temps).

3. Je réalise plusieurs fois l'expérience sur la même solution.

Les principes 1, 2 et 3 seront donc respectés dans l'expérience de ce mémoire. En effet, la réponse à l'hypothèse ayant des caractères subjectifs et les résultats de l'expérience étant principalement graphiques, une rigueur s'impose.

J'établirai plusieurs parties devant valider certains critères, que j'aborderai/fixerai dans le détail dans la partie expérimentation. Si tous les objectifs sont atteints, l'expérience validera l'hypothèse, et inversement. Si les résultats sont mitigés, je peux rejeter la faute sur les aléas de l'informatique et assumer que cela valide quand même l'hypothèse, mais que des poursuites sont nécessaires. Enfin, comme le pointe l'ENS, aucune conclusion n'est définitive.

*« Les réponses que peuvent parfois apporter les scientifiques ne sont pas nécessairement fermes. C'est un principe de prudence car il n'y a pas de raison de dire que les connaissances sont fermes et définitives. »*[5]

[5] **Ecole Normale Supérieure**, Département de physique, Guide « La Démarche scientifique » Page 2.

# 3
# ÉTAT DE L'ART

## 3.1 Etat des connaissances

L'objectif de cette partie est de permettre à chacun de comprendre la suite du mémoire, tant dans les généralités que dans les détails si besoin. Bien que la conception paramétrique ait été employée pour ce travail, son utilisation n'a été que sommaire et ne mérite pas nécessairement un détour sur tout son principe. Les éléments nécessaires à la correcte compréhension seront donc apportés quand cela sera utile, et une présentation brève du paramétrique accompagnera la section dédiée aux logiciels employés.

En revanche, il convient de s'intéresser de près au monde des intelligences artificielles, pour ensuite recadrer ce vaste sujet vers le **Machine Learning**, et enfin les réseaux de neurones, dont celui qui occupe la problématique du mémoire : le fameux **réseau de neurones antagonistes génératifs**.

Il s'agit d'employer une approche complète et pédagogique, pour s'assurer que les éléments complexes mais aussi fondateurs du mémoire soient bien compris, au moins dans leur concept, le tout accompagné d'un glossaire mis à disposition.

### *3.1.a Les intelligences artificielles*

De nos jours, l'intelligence artificielle (abrégée **IA**) devient presque omniprésente, et on commence à entendre que le **Machine Learning** se cache derrière de nombreux systèmes, sans que l'on ne s'en rende compte. Les confusions sont de plus en plus courantes, et il faut définir clairement qui est quoi, et dans quel ordre.

Il est intéressant de parler d'ordre, car comme on peut le voir dans le schéma en figure 3.1.1, chaque terme est une sous-catégorie du terme précédent.

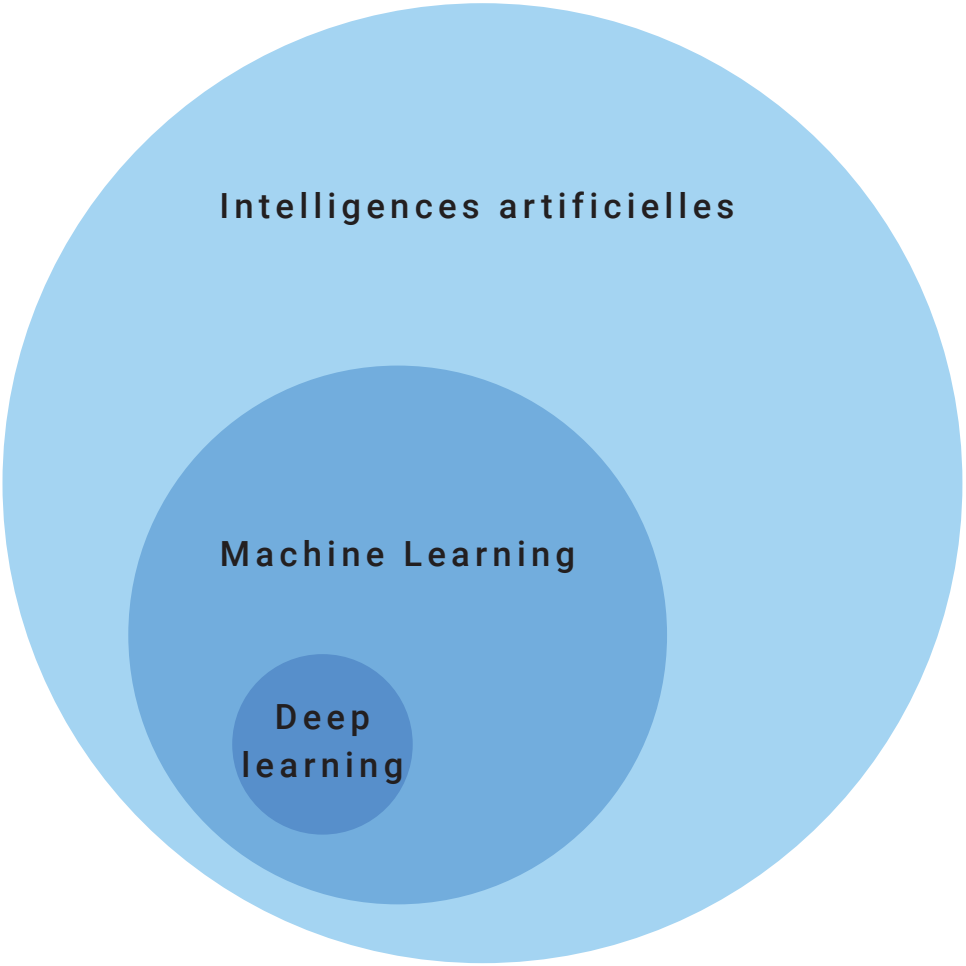



Fig. 3.1.1 - Schéma d'imbrication des familles et sous-familles des intelligences artificielles.

Ainsi, on se situe dans le très vaste domaine des intelligences artificielles. Par définition, cela signifie un ensemble de techniques visant à imiter l'intelligence humaine. Rien de très poussé comme on pourrait le croire : un simple **algorithme** calculant mon Indice de Masse Corporelle (IMC) en fonction de ma taille et de mon poids peut être considéré comme une **IA**, car il reproduit une suite de calculs comme un humain le ferait.

Les petits **algorithmes** très simples font partie intégrante de notre quotidien : de la barrière de parking qui s'actionne quand le ticket est valide, au sèche-mains qui détecte la présence de nos doigts mouillés et invoque le souffle du désert.

Cependant, on va s'intéresser à une catégorie plus précise des **IA**, faisant appel à des **algorithmes** bien plus complexes : le **Machine Learning**.

Choix de la langue de certains termes :

En informatique il devient de plus en plus courant d'utiliser les termes originaux en anglais, y compris dans d'autres langues. En effet, même si des traductions existent (**Machine Learning** peut être traduit par apprentissage machine), elles sont souvent moins pertinentes que le terme original. Dans le cadre de ce mémoire je favoriserai les expressions anglaises, ce qui permettra de les différencier clairement des explications et d'identifier un terme technique. Les traductions et explications nécessaires à leur compréhension se trouveront également dans le glossaire.

### *3.1.b Le* **Machine Learning**

Le **Machine Learning** (en français, apprentissage machine) est un système qui va, au travers d'une approche mathématique et statistique, « apprendre » à partir de données. Cela lui permet de résoudre certaines tâches sans être spécifiquement programmée pour les résoudre. Il s'agit donc d'**algorithmes** plus « intelligents » et plus proches du comportement humain, qui savent aller au-delà des situations pour lesquelles ils n'ont pas été spécifiquement programmés.

Les applications sont souvent du domaine de la statistique. En effet même si l'apprentissage peut se faire sur n'importe quel type de données, les données les plus complexes seront plutôt destinées à la sous-famille du **Machine Learning**, le **deep learning**, ou apprentissage profond. La majorité des données traitées par les **algorithmes** de base du **Machine Learning** sont donc mathématiques.

Il est possible de trouver ce type d'**algorithme** dans la détection de SPAM dans les boîtes mail par exemple. En se basant sur ce que les utilisateurs désignent eux-mêmes comme SPAM ou non, l'**algorithme** apprendra à envoyer ce qu'il faut directement dans le dossier SPAM et ce qu'il faut conserver. C'est le même procédé que l'on retrouve dans les suggestions de publicités sur les sites Internet, dans les systèmes de reconnaissance vocale ou bien dans la détection des fraudes.

Toutes ces explications restent cependant très conceptuelles. Je préfère donc prendre le temps de détailler un type très basique d'**algorithme** de **Machine Learning**. Il ne faut pas oublier qu'il ne s'agit à aucun moment de magie noire, mais de mathématiques pures. Personnellement, plonger dans les abysses du fonctionnement de ces **algorithmes** m'a permis d'éclaircir un peu la boîte noire qu'était jusqu'alors le **Machine Learning**. Il n'est pas question de code, mais de fonctionnement mathématique et statistique.

Je vais donc parler ici de la **descente de gradient**, un **algorithme** encore assez simple pour le comprendre, mais assez avancé pour proposer un avant-goût du fonctionnement du **Machine Learning**.

La **descente de gradient** est un **algorithme** d'optimisation : il va chercher une solution optimale à un problème. Son principe est simple : corriger petit à petit les paramètres de son équation afin de trouver la solution la plus proche du résultat escompté.

La **fonction de coût** est l'outil qui va permettre de dire si le résultat est proche ou non. L'objectif est de minimiser cette **fonction de coût**, afin d'avoir le meilleur résultat possible.

*« Supposons que vous soyez perdu en montagne dans un épais brouillard et que vous puissiez uniquement sentir la pente du terrain sous vos pieds. Pour redescendre rapidement dans la vallée, une bonne stratégie consiste à avancer vers le bas dans la direction de la plus grande pente. C'est exactement ce que fait la **descente de gradient** : elle calcule le **gradient** de la **fonction de coût** au point θ, puis progresse en direction du **gradient** descendant. Lorsque le **gradient** est nul, vous avez atteint un minimum ! »*[1].

[1]**Aurélien Géron,** « Deep Learning avec TensorFlow, mise en oeuvre et cas concrets », Dunod, 2017, p.14.

La courbe ci-dessous proposée par Aurélien Géron cristallise cette analogie de la montagne : le minimum de la **fonction de coût** se situe bien au niveau le plus bas de la vallée.

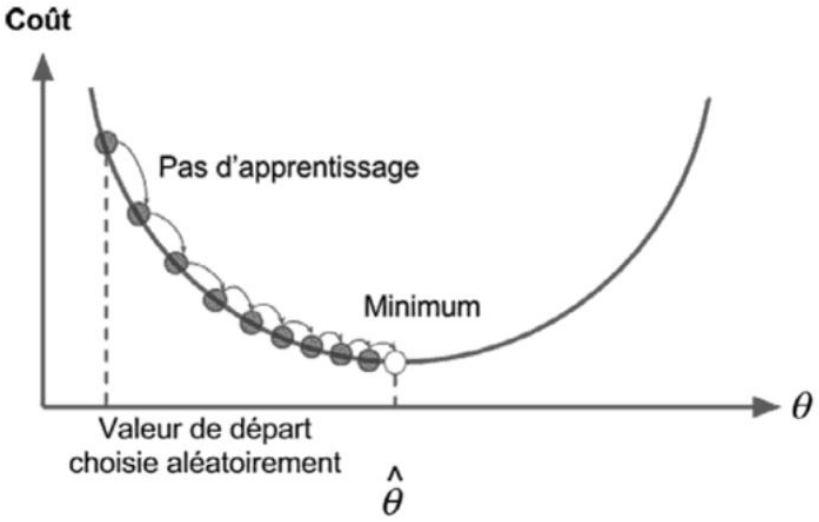



Fig. 3.1.2 - Courbe présentant le principe d'une **descente de gradient** (source : Aurélien géron, Deep Learning avec Tensorflow, 2017).

Il faut également relever le principe de « pas ». L'évaluation du **gradient**, et donc le moment où on se questionne sur le degré de la pente, n'est pas constante. La définition du pas est très importante, et sera d'ailleurs un sujet de réglage durant l'expérimentation.

Un pas trop faible (figure 3.1.3) permet difficilement de trouver le minimum optimal, car trop de pas sont nécessaires pour effectuer le parcours. C'est comme si, dans la montagne, on se demandait quel était le sens de la pente tous les 10 centimètres ; on n'est pas prêt d'arriver !

Un pas trop grand en revanche (figure 3.1.4) ferait louper le minimum recherché. Cette fois c'est comme si on posait la question de la pente tous les 10 kilomètres, sans se rendre compte que l'on vient de passer le fond de la vallée.

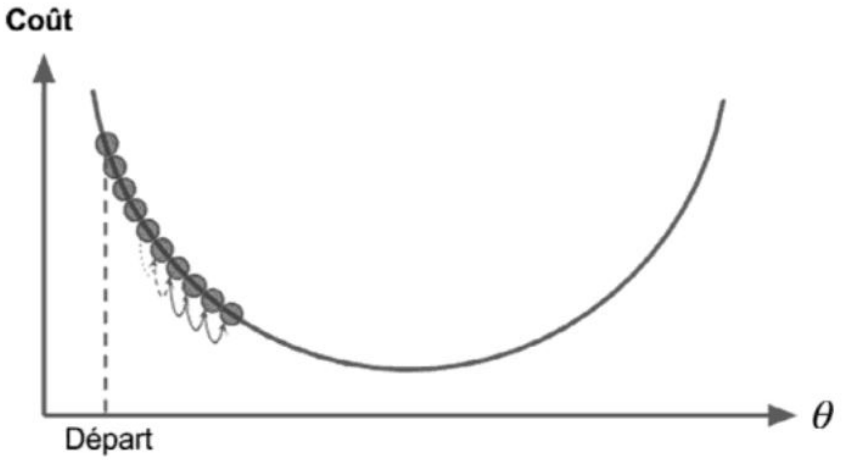



Fig. 3.1.3 - Exemple d'un pas trop faible lors d'une **descente de gradient** (source : Aurélien géron, Deep Learning avec Tensorflow, 2017).

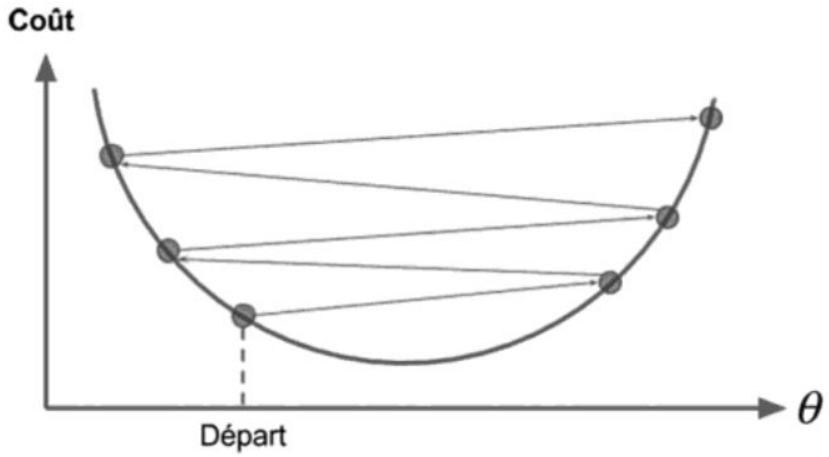



Fig. 3.1.4 - Exemple d'un pas trop élevé lors d'une **descente de gradient** (source : Aurélien géron, Deep Learning avec Tensorflow, 2017).

Dans le cadre d'une vallée, la solution est simple : le fond de la vallée. Les courbes précédentes sont dites convexes, c'est à dire qu'il y a bien un minimum facilement identifiable. Mais, il se peut que le terrain soit plus incertain, et trouver le point le plus bas d'une montagne sans vallée devient plus ardu. La figure 3.1.5 présente une autre courbe, moins régulière. Il est alors possible que le modèle se « coince » dans un minimum local, c'est-à-dire qu'il considère l'emplacement comme étant le plus bas alors qu'il y en a un autre plus bas encore, correspondant au minimum global.

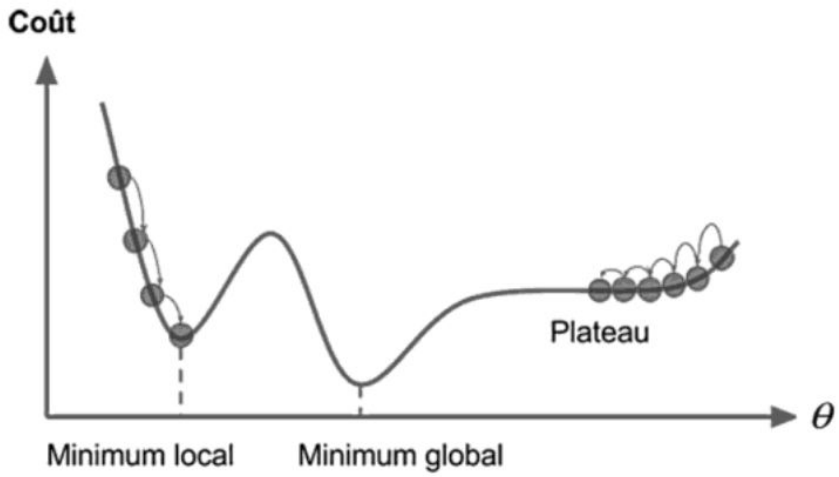



Fig. 3.1.5 - Courbe irrégulière de forme non convexe présentant des minimum locaux et globaux (source : Aurélien géron, Deep Learning avec Tensorflow, 2017).

Ce genre de problème est très courant, et pour s'en sortir il faut soit jouer sur le pas, soit changer totalement de modèle. C'est d'ailleurs un biais qui était arrivé au logiciel de pilotage automatique des voitures Tesla, logiciel qui a dû être retravaillé pour ne pas tomber à nouveau dans un minimum global et continuer à apprendre pour devenir plus performant.

Les voitures autonomes font cependant appel à des **algorithmes** de **Machine Learning** bien plus complexes encore, et qui appartiennent à une catégorie spécifique : le **deep learning**.

### *3.1.c Le deep learning*

Le **deep learning** (ou apprentissage profond en français) est donc un sous-domaine du **Machine Learning** qui va adopter des architectures beaucoup plus complexes et abstraites. Il va lui-même définir ses propres caractéristiques et se révèle donc plus efficace dans le cadre de données complexes (images, son…) et en grand nombre.
C'est bel et bien ce niveau qui est très en vogue aujourd'hui, et qui s'apparente au fonctionnement du cerveau (dans les grandes lignes du moins).

*« Les **algorithmes** du **deep learning** traitent l'information reçue de façon similaire à ce que feraient nos réseaux de neurones en réponse aux signaux nerveux qui leur sont destinés. En fonction du type et de la fréquence des messages reçus, certains réseaux de neurones vont se développer quantitativement et qualitativement alors que d'autres vont régresser.»*[2]

Ces neurones artificiels vont effectuer des calculs, selon un certain poids qui leur est attribué, et communiquer ces résultats entre eux au travers d'une organisation en plusieurs couches (voir figure 3.1.6 page suivante). Au fil de sa progression, le réseau de neurones va voir ses poids évoluer et s'adapter selon le résultat souhaité. La première couche est communément appelée couche d'entrée, et la dernière, couche de sortie. Entre les deux se trouvent plusieurs couches « cachées » dont le nombre n'est pas fixe.

Les **algorithmes** de **deep learning** font donc preuve d'une grande adaptabilité, et nécessitent une intervention moindre de l'homme durant le processus. Ainsi comme le montre le schéma en page suivante (figure 3.1.7), là où l'homme doit expliquer à l'**algorithme** de **Machine Learning** ce qu'il doit chercher et à quoi ça ressemble, l'**algorithme** de **deep learning** lui reconnaîtra seul les caractéristiques importantes et en fera plusieurs couches de neurones de façon plus ou moins abstraite.

---

[2]**Léa S,** « Quelle différence entre Machine Learning et Deep Learning ? », Site Web DataScientest, Section Deep Learning, Juillet 2020, consulté le 28/12/20. Disponible sur : https://datascientest.com/quelle-difference-entre-le-machine-learning-et-deep-learning.

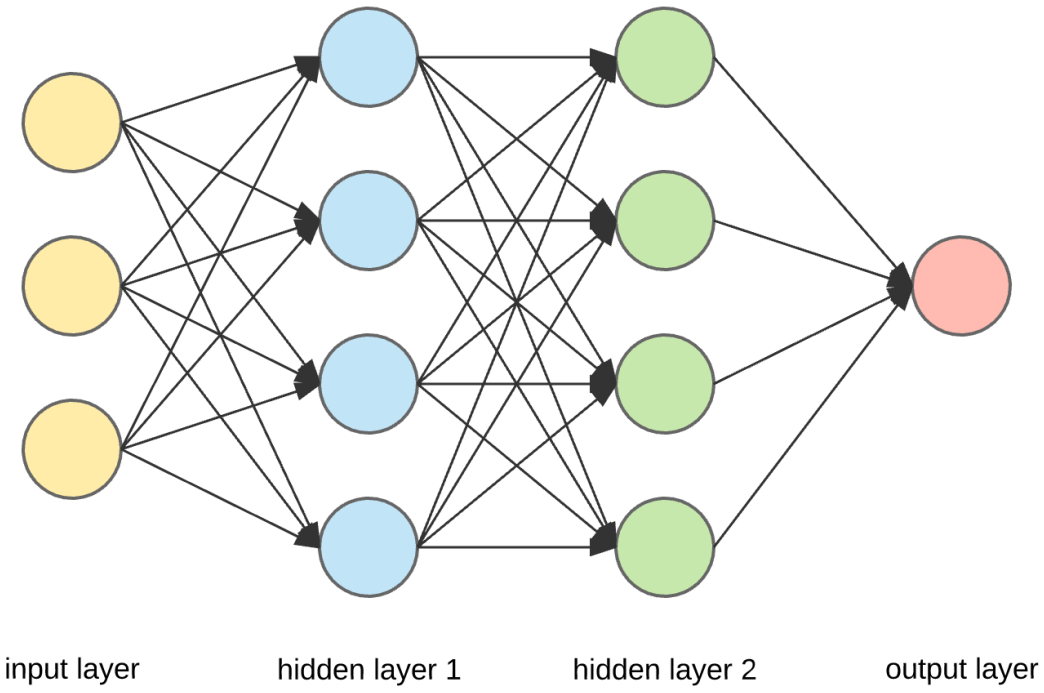



Fig. 3.1.6 - Schéma simplifié de la structure en couche d'un réseau de neurones (source : Gavril Ognjanovski, Towards DataScience).

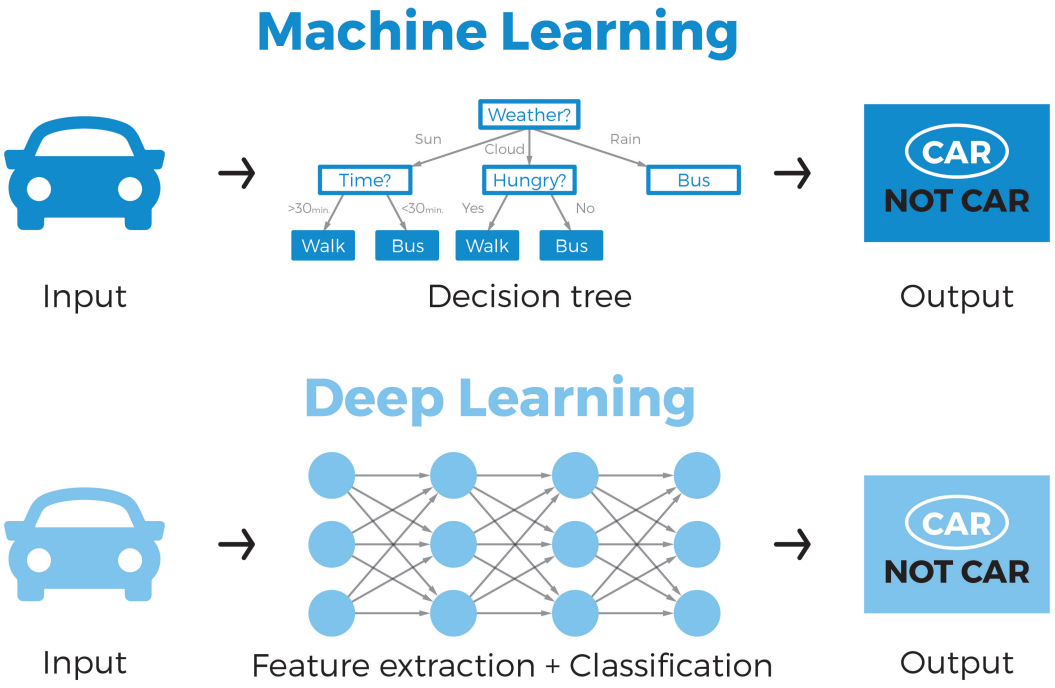



Fig. 3.1.7 - Schéma comparatif entre les grandes étapes du **Machine Learning** et du deep learning (source : stock image).

Maintenant que le sujet a été suffisamment recadré, il est possible de citer des architectures, soit des formes connues que peuvent prendre ces **algorithmes**. C'est également l'occasion de citer quelques exemples d'application pour chacune de ces architectures, pour aboutir enfin à celle qui sera utilisée dans ce mémoire.

## CNN

L'architecture la plus repandue est matérialisée par les **réseaux de neurones convolutifs** (convolutional neural network en anglais, d'où l'abréviation **CNN**). Ils sont principalement utilisés dans la reconnaissance d'images et de vidéos. C'est par exemple sur cette architecture que se basent les filtres vidéo (Snapchat, Messenger, Zoom…) ou encore la reconnaissance faciale de nos smartphones.

Le **CNN** va convertir l'image en chiffres, selon une méthode lui permettant de repérer les changements de couleurs, les contours, les limites des éléments présents sur l'image. Cette transformation fait appel à un procédé mathématique, le produit de convolution, qui a donné son nom à l'architecture.

L'avantage non négligeable d'un **CNN** est qu'il « apprend » pour de bon : il décompose les éléments à reconnaître en caractéristiques, qu'il peut ensuite recouper et remanier afin de s'adapter à une autre image. Bien que deux lapins ne soient jamais parfaitement identiques en termes de pixels, ils auront globalement les mêmes caractéristiques et le réseau de neurones saura indiquer leur ressemblance.

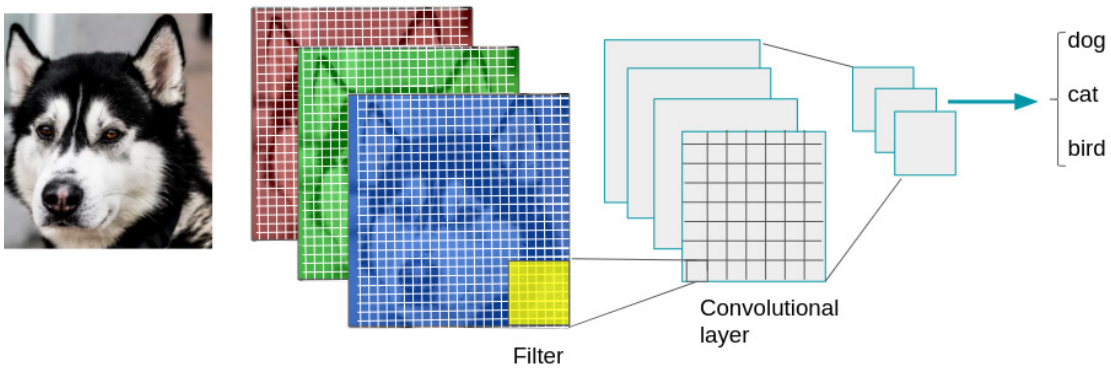



Fig. 3.1.8 - Schéma des grandes étapes d'un **réseau de neurones convolutifs / CNN** (source : Gaël G, DATAKEEN).

RNN

Les **réseaux de neurones récurrents** (**RNN** pour recurrent neural network) sont également une architecture très répandue. Ils ont la particularité de tenir compte de l'enchaînement successif des données. Ils vont ainsi garder en mémoire les différentes données successives, ce qui va les rendre très performants dans le domaine de la traduction automatique ou encore l'analyse de séquences ADN.

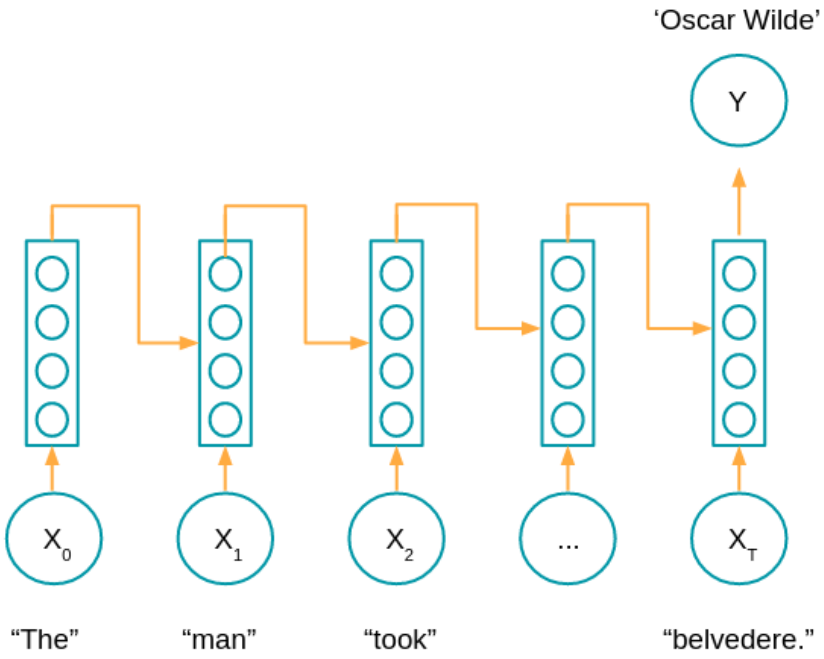



Fig. 3.1.9 - Schéma d'une succession d'étapes d'un **réseau de neurones récurrent / RNN** (source : Gaël G, DATAKEEN).

AE

Les **auto-encodeurs** incarnent également une architecture courante, mais qui encore une fois est très différente des autres architectures présentées. Le principe est en revanche assez simple (voir figure 3.1.10) : l'**algorithme** va dans un premier temps « encoder » les données qui lui sont fournies, soit les traduire de manière mathématique et abstraite en plusieurs couches cachées. Ensuite une seconde partie va venir décoder ces données, afin de leur faire retrouver leur forme d'origine.

Mais pourquoi tout ce travail pour revenir aux résultats de base ? Il s'avère que sous forme encodée, les données sont alors beaucoup moins nombreuses et lourdes, ce qui est plutôt pratique quand les bases de données peuvent inclure des millions, voire milliards d'éléments.

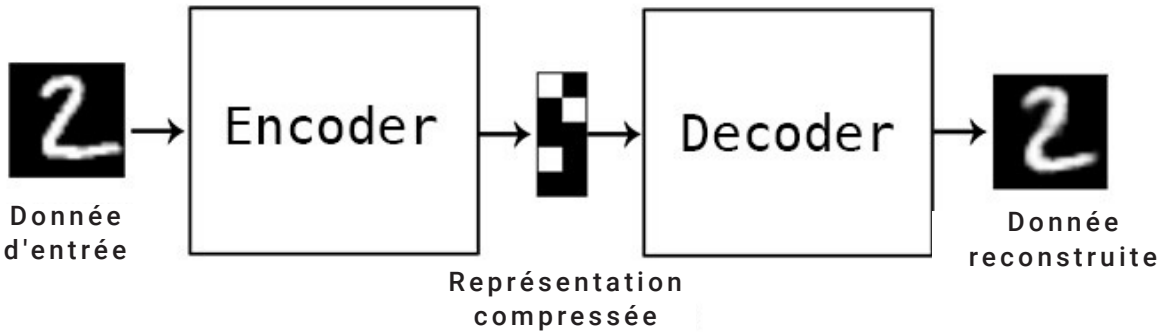



Fig. 3.1.10 - Schéma des grandes étapes d'un réseau de neurones auto-encodeurs (source : Will Badr, Towards Data Science).

L'application principale des **auto-encodeurs** reste cependant la détection d'anomalie. Comme le décodeur a appris à décoder des données très précises, et si curieusement il n'y arrive pas ou moins bien, la donnée en question présente une différence. Ce processus étant entièrement autonome (peu importe le type de données en entrée, il saura travailler correctement), ce qui en fait un outil redoutable et facile à implémenter.

GAN

Enfin, il convient d'aborder un dernière architecture clé : les **réseaux de neurones antagonistes génératifs**, que j'abrège par la dénomination **GAN** en raison de son nom anglais (Generative Adversarial Networks). C'est cette architecture en particulier qui sera choisie pour l'expérimentation de ce mémoire, grâce à son potentiel créatif de génération d'images. Ainsi, elle mérite un certain approfondissement, car une bonne compréhension de son fonctionnement m'a été plus que nécessaire pour faire face aux nombreux problèmes et erreurs.

### *3.1.d Le **GAN***

Le principe d'un **réseau de neurones antagonistes génératifs** ou **GAN** est un peu plus complexe que les architectures précédentes. Dans son nom, on retrouve déjà les deux composantes clés de son fonctionnement : « antagoniste » fait référence à une adversité, une compétition, et « génératif » montre bien la capacité à créer des données.

Son aspect génératif permet d'offrir au **GAN** un large champ d'applications. En effet à partir de n'importe quelles données, il saura à son tour en générer de la même sorte. Ainsi en apprenant sur une base de données de visages, le **GAN** saura à son tour produire des visages humains (figure 3.1.11). Il peut également reconstruire des images dont une partie serait manquante (figure 3.1.12) s'il a appris sur des images similaires.

C'est un procédé que l'on retrouve également dans la création automatique de cartes (figure 3.1.13) : comme le **GAN** sait différencier en vue aérienne les espaces verts des routes et des habitations ainsi que leurs dimensions, il peut à son tour générer des cartes de type « IGN ».

Les déclinaisons sont alors aussi nombreuses qu'il existe de type de données. Voyons maintenant comment cela fonctionne.

Fig. 3.1.11 - Exemple de génération de visages féminins (source : Jason Bronwlee, Machine Learning Mastery).

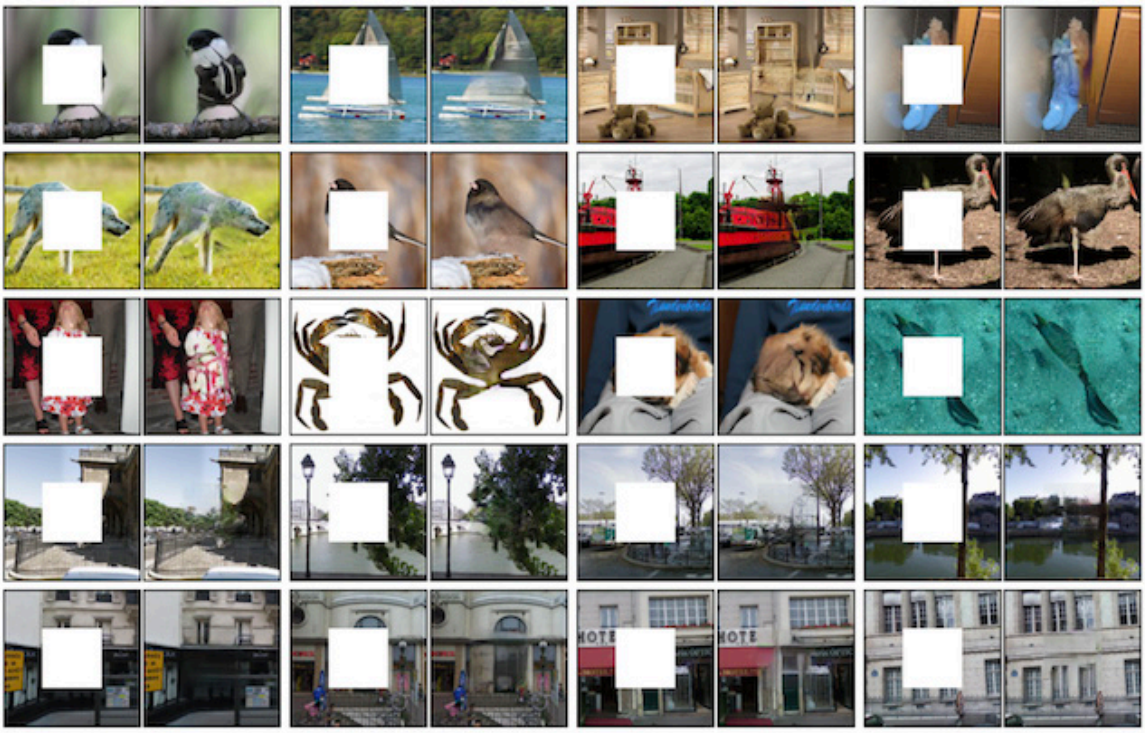


Fig. 3.1.12 - Exemple de reconstruction d'images (source : Jason Bronwlee, Machine Learning Mastery).


Fig. 3.1.13 - Comparaison entre la génération d'une carte IGN et sa version réalisée par un humain (source : Jason Bronwlee, Machine Learning Mastery).

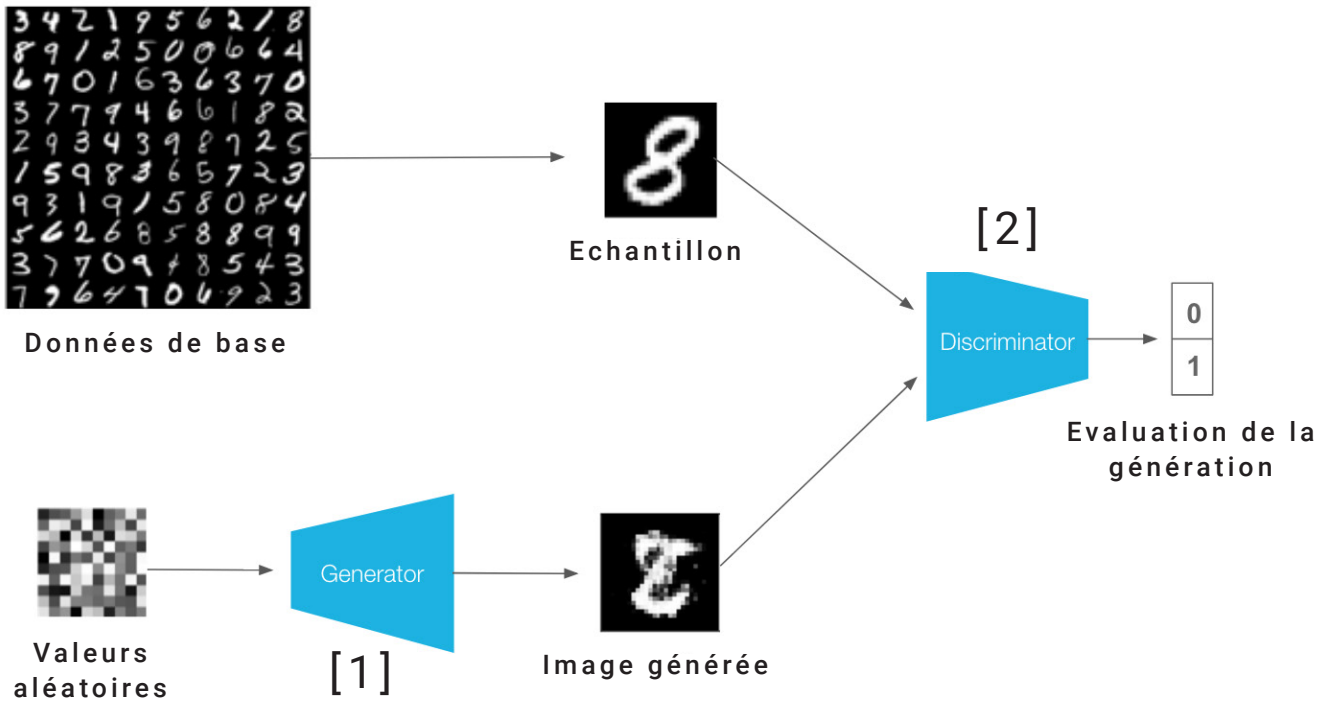



Fig. 3.1.14 - Schéma du fonctionnement d'un **GAN** (source : Afnan Amin Ali, The Startup, Medium).

Le schéma ci-dessus illustre le fonctionnement d'un **GAN** sur des chiffres manuscrits de 0 à 9. L'objectif du **GAN** dans cet exemple est de produire un de ces chiffres de 0 à 9.

Le **générateur** [1] n'a aucune idée de ce qu'il doit faire, et va produire une image pixellisée au hasard. Le **discriminateur** [2] quant à lui va comparer le résultat du **générateur** avec la véritable base de données, et noter le travail du **générateur**. Le processus continue ainsi de suite, le **générateur** essayant de s'approcher au mieux de ce que le **discriminateur** note correctement.

C'est comme si un élève d'architecture (le **générateur**) n'avait jamais fait de projet. Le professeur (le **discriminateur**) va juger le travail de l'élève et le noter assez mal, car le premier rendu est franchement mauvais. De rendu en rendu l'élève va comprendre les attentes du professeur, et va s'améliorer et enfin produire des projets cohérents avec les attentes.

Cependant, il y a une subtilité. Dans le cas d'un **GAN**, on considère que le professeur apprend également, comme si c'était ses premières corrections. Ainsi de rendu en rendu, le professeur devient de plus en plus exigeant car plus expérimenté. Il en va de même pour le **GAN** : le **générateur** comme le **discriminateur** s'améliorent en parallèle au fil des **itérations**. C'est de là que vient le phénomène de compétition, d'adversité : élève et professeur ne cessent de devenir meilleurs. Bien qu'ils ne soient par conséquent jamais satisfaits du travail de l'autre, les données produites en fin de chaîne sont de plus en plus proches du résultat escompté.

L'ensemble des solutions produites vont prendre place dans un espace, bien souvent en plus de 3 dimensions. Cet espace dit « **espace latent** » va permettre de retrouver certaines solutions en fonction de leurs coordonnées dans l'espace, et ainsi découvrir aussi bien des solutions étranges que réalistes, mais toujours intéressantes.

Dans l'expérience de ce mémoire, je vais ainsi partir d'un **GAN** existant, fabriqué pour générer des chiffres de 0 à 9, puis le modifier pour qu'il fonctionne sur des chiffres que j'aurai moi-même produit paramétriquement. Je remplacerai ensuite ces chiffres par d'autres images paramétriques, de plus en plus proches de l'architecture, afin de montrer que la génération du **GAN** est source d'enrichissement créatif.

3.2 Etat de la recherche

L'architecture et le **deep learning** n'ont jusqu'alors été que très peu rapprochés, même si cela suscite de plus en plus de curiosité (ou de débat). Aucun sujet d'ailleurs ne parle d'y incorporer une dimension paramétrique, ce qui m'a permis de poser cette problématique. En revanche, deux papiers de recherches sont très proches de mon travail, et ont grandement contribué à forger mes premières réflexions.

Le premier travail est celui de Sophie Oros[3], dans le cadre d'un mémoire de Master à l'ENSA Paris-la-Villette : *« L'apprentissage machine au service de la conception architecturale ? Une application : extraire des informations à partir de plans. »*. Centré sur la lecture de plans, sa stratégie d'approche est très proche de la mienne, avec un processus de création de données puis de traitement par un réseau de neurones. Son travail m'a permis de développer un processus, le tester puis l'appliquer à l'architecture.

Un autre travail significatif est celui de Stanislas Chaillou[4], *« AI + Architecture »*, une thèse très poussée dont les résultats ont su m'émerveiller et me motiver à choisir ce domaine de recherche. Son travail se concentre sur la création d'empreintes de bâtiments sur une parcelle grâce à un réseau de neurones, entraîné sur une base de données de maisons individuelles à Boston. Ces empreintes sont ensuite divisées en pièces, meublées, et triées selon les orientations solaires et la capacité de circulation à l'intérieur. Ce tri permet enfin de choisir des solutions pertinentes, selon nos préférences.

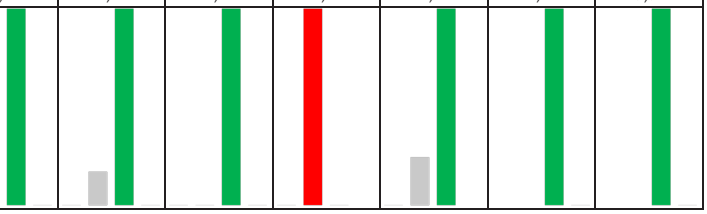
Cette recherche d'une très grande qualité, tant dans le processus que dans les résultats, m'a beaucoup inspiré, mais surtout poussé à choisir un sujet à ma portée. En effet, les résultats très concluants proposés par Stanislas Chaillou sont le fruit d'années de recherche, indispensables pour pouvoir véritablement concevoir un « outil ».

Ces travaux ont ainsi servi de socle à ma réflexion, tant dans les idées générales que les processus adoptés, et m'ont permis de me concentrer sur la notion d'enrichissement d'une base de données paramétrique.

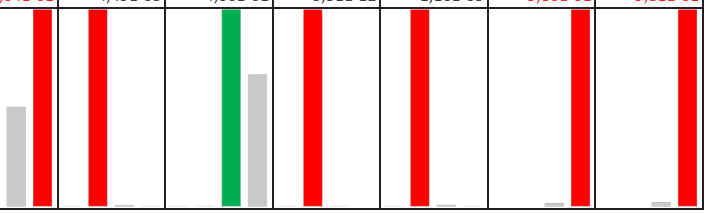
---

[3] **Sophie Oros**, « L'apprentissage machine au service de la conception architecturale ? Une application : extraire des informations à partir de plans », Mémoire de master dans le cadre du séminaire Activités et Instrumentation de la Conception, ENSA de Paris la Villette, 2019.

| | Plans modifiés manuellement | | | | | | |
|---|---|---|---|---|---|---|---|
| | Original | Modifié 1 | Modifié 2 | Modifié 3 | Modifié 4 | Modifié 5 | Modifié 6 |
| Image |  |  |  |  |  |  |  |
| Réponse attendue | 2 | 2 | 2 | 2 | 2 | 2 | 2 |

| Réponses du modèle entraîné avec la base de donnée BASIQUE | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1,38E-05 | 4,92E-04 | 3,76E-05 | 1,55E-07 | 5,40E-05 | 4,54E-10 | 5,22E-10 |
| 1 | 2,44E-05 | 1,45E-01 | 4,96E-04 | 1,00E+00 | 1,96E-01 | 2,48E-09 | 2,90E-09 |
| 2 | 1,00E+00 | 8,54E-01 | 9,99E-01 | 7,70E-08 | 8,04E-01 | 9,99E-01 | 9,99E-01 |
| 3 | 1,64E-04 | 4,00E-08 | 1,09E-04 | 8,36E-16 | 2,98E-09 | 8,81E-04 | 7,74E-04 |
| Histogrammes (échelle relative) |  | | | | | | |

Taux de réussite= 86%

| Réponses du modèle entraîné avec la base de donnée EVOLUEE | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1,46E-04 | 9,59E-05 | 2,09E-05 | 7,70E-07 | 7,39E-06 | 4,97E-15 | 8,89E-15 |
| 1 | 4,93E-04 | 9,94E-01 | 6,08E-04 | 1,00E+00 | 9,94E-01 | 1,19E-10 | 1,18E-10 |
| 2 | 3,35E-01 | 5,72E-03 | 5,99E-01 | 1,36E-07 | 6,00E-03 | 1,41E-02 | 1,92E-02 |
| 3 | 6,64E-01 | 4,49E-05 | 4,00E-01 | 5,51E-12 | 1,16E-05 | 9,86E-01 | 9,81E-01 |
| Histogrammes (échelle relative) |  | | | | | | |

Taux de réussite= 14%

Fig. 3.1.15 - Résultats d'analyses de plans par un réseau de neurones, mémoire de Shopie Oros[3].

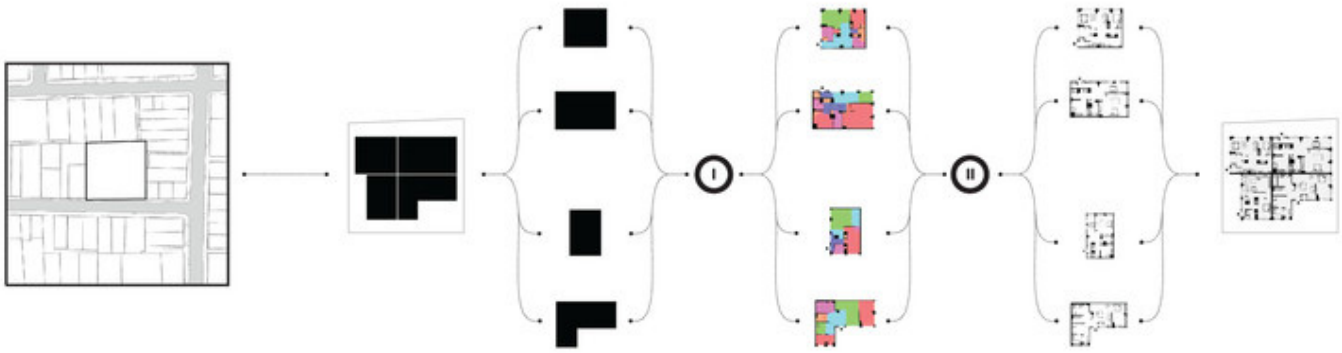


Fig. 3.1.14 - Schéma des étapes du processus élaboré pour la thèse de Stanislas Chaillou[4].

[4] **Stanislas Chaillou,** « AI + Architecture », Thèse de doctorant à l'Université de Harward, Graduate School of Design, Cambridge, 2019.

# 4
# EXPÉRIMENTATION

A la lumière des connaissances accumulées lors du chapitre précédent, il est nécessaire d'envisager l'expérimentation qui saura étayer la problématique du mémoire : *« Comment enrichir un espace de solutions paramétriques grâce aux réseaux antagonistes génératifs ? »*.

Plus qu'une expérience unique, il s'agira avant tout de mettre en place une logique expérimentale, et établir un processus à suivre. Je commencerai donc par mettre en place un schéma paramétrique permettant de réaliser des données imagées en grande quantité, qui serviront ensuite de jeu d'apprentissage et d'entraînement au réseau de neurone antagoniste génératif. L'étude des résultats, en parallèle de la perfection des paramètres aux différents niveaux de travail permettront de valider ou non l'hypothèse.

## 4.1 Les logiciels

Cette expérimentation a nécessité le travail sur de nombreux outils, logiciels et interfaces existantes, chacun avec ses propres codes et objectifs. Au-delà de savoir les utiliser, il est aussi bon de savoir quel est leur intérêt et à quoi vont-ils servir au sein de mes expériences. Particularité supplémentaire, l'utilisation de tous ces outils informatiques s'est faite en grande partie de façon « imbriquée », les résultats d'un logiciel étant traités par le suivant pour être acceptés par un dernier, par exemple.

- **Rhinocéros 3D** :


C'est un logiciel ce conception assistée par ordinateur (CAO). Bien que très pratique pour réaliser des formes complexes et tridimensionnelles, il sera ici employé majoritairement pour ses capacités de dessin vectoriel en deux dimensions, et sa compatibilité avec le **plugin Grasshopper** et ses fonctionnalités paramétriques. Très utilisé dans le design industriel, il est également connu des architectes pour des modélisations plus « libres » que les logiciels métiers prédominants dans la profession tels que ArchiCAD ou Revit.

- **Grasshopper :**

*« **Grasshopper (GH)** est un **plugin** de **Rhinoceros 3D**. Un **plugin** (encore appelé module d'extension ou module externe) est un programme complétant les fonctionnalités d'un logiciel. Le **plugin GH** permet de créer des modèles paramétriques sur **Rhinoceros** grâce à de la programmation visuelle. »*

Il s'agira du principal outil de programmation de l'expérience, son paradigme permettant d'être plus facile à appréhender que la programmation brute. En effet la programmation visuelle permet de manipuler des éléments graphiques plutôt que du texte. Par exemple dans l'exemple ci-dessous, il est possible de voir le procédé en code (**Python**) et sous **Grasshopper** pour arriver au même résultat : une courbe sinusoïdale. Pour les non-initiés, il est plus simple de comprendre et refaire la version visuelle, celle-ci n'ayant pas recours aux codes du langage textuel (respecter chaque fonction, chaque espace, chaque caractère au risque de faire bugger l'ensemble).

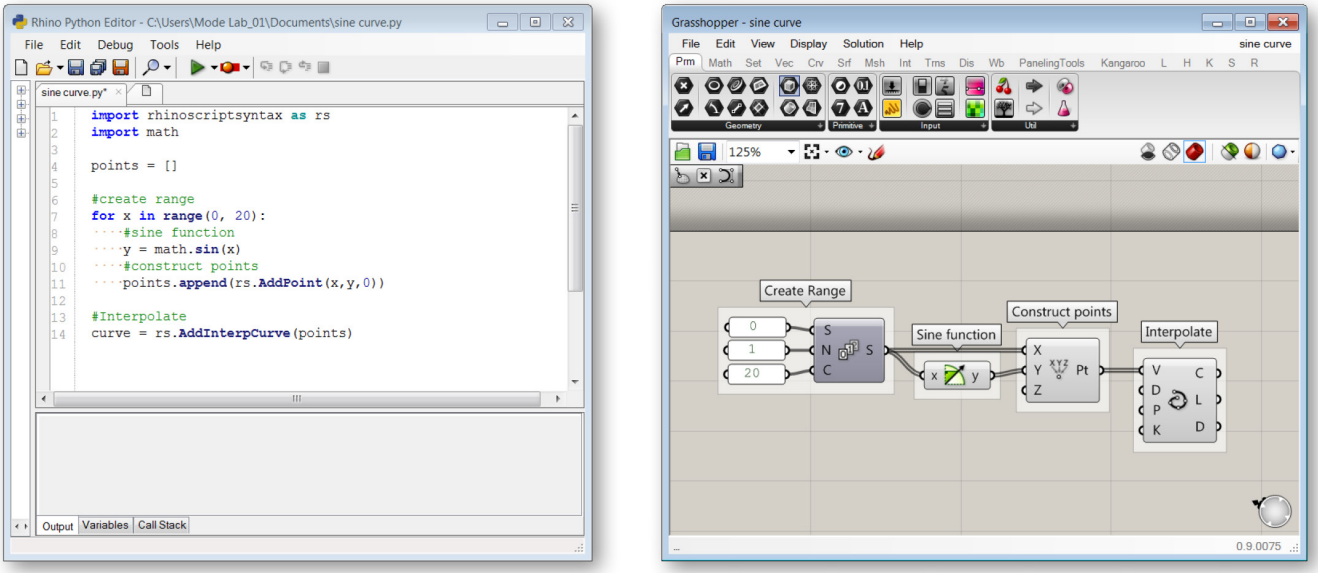



Fig. 4.1.1 - Comparatif entre programmation brute et visuelle (**Python**/**Grasshopper**). Source : Ehsan Bazafkan, *Assessment of Usability and Usefulness of New Building Performance Simulation Tools in the Architectural Design Process.*

Mais quel intérêt de « programmer » en architecture ? Plutôt que de modéliser les éléments un par un, j'indique leurs paramètres et c'est le logiciel qui rendra compte de leur visuel. Je peux alors changer les paramètres entrés et observer dynamiquement le résultat, sans avoir besoin de « remodéliser ». La conception paramétrique amène donc à de nouvelles formes et concepts qui iront jusqu'à devenir la marque de fabrique de certains architectes. **Grasshopper** est donc déjà présent dans la discipline et dans les esprits.

Dans le cadre de l'expérience, c'est cette qualité de paramétrisation qui va prédominer, ainsi que la possibilité de réitérer un même processus à une vitesse machinique.

Tout comme **Grasshopper** est un **plugin** de **Rhinocéros**, il est possible d'ajouter des **plugins** à **Grasshopper** afin d'étendre ses fonctionnalités. J'utiliserai donc le **plugin** TT TOOLBOX par CORE studio afin de capturer et enregistrer en masse les images générées par le fichier **Grasshopper**.

- **Python :**

**Python** n'est pas un logiciel à proprement parler, mais un langage de programmation, soit un regroupement de règles de grammaire et de significations afin de produire des **algorithmes** informatiques. Parmi les nombreux langages existants, j'ai choisi **Python**. En effet, il est déjà grandement répandu, bien documenté, relativement compréhensible et employé par de nombreux logiciels. Au-delà de ces considérations assez personnelles, Gérard Swinnen pointe des éléments plus concrets dans le manuel qui m'a servi à apprendre les bases de **Python** :

*« Pour nos débuts dans l'étude de la programmation, il nous semble préférable d'utiliser un langage de plus haut niveau, moins contraignant, à la syntaxe lisible. Après avoir successivement examiné et expérimenté quelque peu les langages Perl et Tcl/Tk, nous avons finalement décidé d'adopter **Python**, langage très moderne à la popularité grandissante. »*[1]

[1] **Gérard Swinnen,** « Apprendre à programmer avec Python 3 », Eyrolles, 2017, préface.

Ce qu'est **Grasshopper** à **Rhino**, les **libraries** (bibliothèques en français, mais le terme anglais est utilisé quelle que soit la langue) le sont à **Python**. Elles regroupent une multitude de fonctions préprogrammées qui permettront d'étendre encore les possibilités de **Python** et ce de manière rapide et lisible. Parmi les **libraries** les plus importantes, ou du moins celles avec lesquelles je vais travailler le plus on trouve :

- **TensorFlow :** ensemble de **libraries** pour le **Machine Learning** avec **Python**, développé par Google. Outil open-source et flexible, il peut être employé pour toutes sortes de tâches à plusieurs échelles, du téléphone portable aux fermes de serveurs. J'ai pu réaliser beaucoup d'essais avec **TensorFlow**, mais c'est finalement **Keras** qui portera l'expérience retenue pour ce mémoire.

- **Keras :** très similaire à **TensorFlow**, cette **library** est plus ergonomique et plus flexible.


- **NumPy : library** destinée à la création de matrices et tableaux multidimensionnels.


- **Matplotlib :** spécialisée dans la création de graphiques 2D et 3D de données mathématiques, pouvant coopérer avec **NumPy**.


- **MLxtend :** tout comme **TensorFlow** et **Keras**, cette **library** est destinée au **Machine Learning**, mais apportant de nouvelles fonctionnalités auxquelles j'aurai recours.

Toutes les **libraries** n'étant pas toujours compatibles entre elles, il arrive de créer un **environnement virtuel** particulier, afin de séparer une expérience de l'autre, une installation de **Python** avec certaines **libraries** dans certaines versions d'une autre, afin d'éviter tout conflit. Pour mettre en place cet environnement, je peux le créer directement à partir de l'**invite de commande Windows**, ou se simplifier la vie et utiliser une interface :

- **Anaconda Navigator :** cette interface permet de créer et gérer des environnements virtuels en quelques clics sans ligne de programmation, ainsi que d'ajouter, supprimer et mettre à jour les **libraries** de chaque environnement.

- **Jupyter Notebook :** également une interface, ouverte depuis **Anaconda Navigator**. C'est une application web (s'ouvrant via un navigateur) permettant de mieux visualiser le code rédigé sous **Python** : les fonctions se teintent en couleur, certains éléments de syntaxes également, ce qui facilite grandement la lisibilité. Je peux également séparer le code en plusieurs cellules, pouvant être lancées seules ou à plusieurs, étape par étape.

Les présentations étant faites, je peux enfin passer à l'élaboration plus concrète de l'expérimentation de ce mémoire.

4.2 Mise en place du processus

La logique expérimentale ayant été abordée ainsi que les notions de base et la présentation des différents logiciels, je peux maintenant m'intéresser dans le détail au processus expérimental, structure sur laquelle se construira l'expérience.

Commençons par rappeler la problématique du mémoire : *« Comment enrichir un espace de solutions paramétriques grâce aux réseaux antagonistes génératifs ? »*. L'expérience doit donc témoigner de l'enrichissement des données créées paramétriquement par un **réseau de neurones antagonistes génératifs**. Ces données seront nécessairement des images, facilement interprétables et pouvant être jugées et comparées les unes aux autres. Voici donc la structure grossière de cette expérience :

1. Je génère des images de façon paramétrique.
2. Le réseau de neurones apprend de ces images.
3. Il génère à son tour des images « enrichies ».

Cependant comment qualifier l'enrichissement des nouvelles images générées ? Pour ce faire il faut définir plus précisément les critères qui seront testés ici. J'établis donc que pour qu'une expérience valide l'hypothèse, je dois observer parmi les résultats fournis par l'**algorithme** 3 types d'éléments :

- **A. Une image fidèle aux images d'apprentissage : cela montre que l'algorithme a correctement appris, et qu'il sait a minima imiter à l'identique,**
- **B. Une image « chimérique » : un mélange entre plusieurs images, premier degré d'enrichissement recherché,**
- **C. Des étrangetés sans nom qui observent un potentiel créatif recherché également (second degré d'enrichissement).**

*4.2.a Le phasage*

Une seule et même expérience ne serait évidemment pas assez pertinente. En effet comme je l'évoquais dans la partie dédiée à la logique expérimentale, il est préférable d'avoir une expérience témoin, mais également séparer les variables à tester. Ainsi je décide de structurer l'expérimentation en 3 parties distinctes :

- Partie 1, l'expérience témoin

L'objectif n'est pas de valider directement l'hypothèse, mais d'assurer dans un premier temps la validité de l'expérience et de son processus, ainsi que son potentiel de précision d'apprentissage.
Pas besoin donc de faire référence de près ou de loin à l'architecture, qui serait même ici un handicap. En effet, juger de la pertinence d'une image architecturale, suite à son passage dans un **algorithme** reste fortement subjectif. L'idée est donc de choisir des symboles connus de tous, dont l'interprétation est assez évidente : j'ai retenu les chiffres. En effet il est aisé de dire qu'un 9 généré ressemble bien à un 9 et non à un trait aléatoire ou à un vélo.

Cette expérience témoin doit également montrer que le processus observe un degré de précision, de « finesse » d'apprentissage. Aussi, plutôt que de travailler avec des chiffres dactylographiés, je favorise l'écriture manuscrite. C'est un processus exercé depuis des milliers d'années, qui a été naturellement optimisé par son usage par l'Homme et qui est donc un pur produit du cerveau et de l'apprentissage. C'est également un processus renseigné : par exemple les manuels pour enfants d'aujourd'hui proposent les techniques de représentation les plus optimales, et je peux m'en inspirer.

Au terme de cette première partie, si les critères A, B et C décrits en amont sont validés, l'expérience témoin l'est également et je peux procéder à la partie suivante. Je considère alors le processus comme fixe : le réseau de neurones restera strictement inchangé d'une expérience à l'autre et seule une partie du modèle paramétrique évoluera afin de générer des données différentes au fil des parties. La "variable" entre chaque expérience est donc le base de données paramétrique.

- Partie 2, critère d'étude : la forme

Maintenant que je sais (ou suppose) que le processus expérimental valide grâce à l'expérience témoin, je peux tester un premier critère : la FORME. Je peux donc me défaire de la recherche de complexité de la partie 1, et me concentrer sur des symboles plus simples que sont les formes basiques telles que le carré, le rond, le triangle ou encore un polygone irrégulier.

Là où les chiffres étaient complexes graphiquement mais tracés avec la même logique (mêmes dimensions, même nombre de traits) pour évaluer le potentiel de l'**algorithme**, cette seconde partie traite des symboles plus simples dans l'aspect, mais sans logique similaire de construction (un carré avec 4 côtés, un rond qui n'a plus vraiment de principe de côté, un polygone étrange…).

La difficulté d'apprentissage est donc tout autre pour l'**algorithme**. En tant qu'humain j'aurais tendance à dire qu'apprendre à dessiner un rond est plus simple qu'un joli 6, mais l'apprentissage machine ne fonctionne pas vraiment (et aussi bien) qu'un cerveau humain. La complexité n'est plus graphique, mais paramétrique.

J'établis donc une série de formes tout d'abord simples, avec peu de paramètres qui vont varier, puis de plus en plus complexes.
Là encore, je peux juger de la pertinence des résultats par le biais des critères A, B et C.

- Partie 3, critère d'étude : la disposition

Enfin je m'intéresse à la DISPOSITION. Les tests précédents abordaient des éléments situés sur un même centre géométrique, je vais essayer de m'en abstraire avec des formes bien plus simples encore que celles retenues au niveau des parties 1 et 2, mais de façon plus dynamique.

Ainsi on va observer une forme carrée ou rectangulaire variant assez peu en soit, mais dont les positions seront très différentes sur l'image (dans un angle, gauche, droite, centre).

C'est la réflexion que je pourrais qualifier de plus « architecturale », ou du moins, urbanistique dans son état le plus élémentaire. Je peux ainsi imaginer qu'il s'agit d'un alignement nécessaire à satisfaire en bord de parcelle par exemple, ou un retrait minimum par rapport à une limite de parcelle.

Chaque phase a donc ses propres critères d'étude, son propre apport à l'expérimentation et donc à la problématique. Je considère donc que si ce tableau se teinte de validation, la réponse à l'hypothèse sera plutôt positive (voir tableau 4.2.1 ci-dessous). Cette grille d'évaluation sera mise à jour au fur et à mesure des différentes expériences et résultats.

Pourquoi ne pas rassembler toutes les parties et laisser la machine apprendre ? Au-delà de la rigueur scientifique qui tend à séparer les facteurs, on est également limité par la capacité de calcul de l'ordinateur et, par conséquent, la résolution des données de travail. Croiser des milliers d'exemples de formes très complexes et aux dispositions changeantes risquerait d'amener à des temps d'apprentissages babyloniens ainsi qu'à des résultats peu pertinents ou même inexploitables.

| Critère / Expérience | Fidélité | Chimères | Etrangetés |
|---|---|---|---|
| Partie 1 "chiffres" | | | |
| Partie 2 "formes" | | | |
| Partie 3 "dispositions" | | | |

Tab. 4.2.1 - Fond de tableau bilan des résultats obtenus.

*4.2.b Structuration d'une partie*

L'expérimentation se déroule en 3 parties certes, mais il faut également établir une procédure, des étapes à respecter identiques pour chaque partie.

- *Mise en place d'une stratégie*

Dans un premier temps, il faut définir les données à analyser, puis ce que je souhaite obtenir. Par exemple pour la première expérience je me base sur l'écriture manuscrite des chiffres de 0 à 9, et je veux aboutir à une base de données constituée d'images de chaque chiffre, individuelles, en grande quantité et normalisées (mêmes dimensions, résolution, couleurs ou non…).

Il est donc nécessaire de s'interroger sur l'essence même de la donnée que je veux exploiter. Dans le cadre des chiffres, il y a lieu de se demander comment chaque chiffre est tracé par la main, quels sont les paramètres de variation d'un essai à l'autre, et selon quelle envergure.

La définition d'une stratégie précise dès le début permettra ensuite, durant toute l'expérience, de garder à l'esprit les points de départ et d'arrivée, ce qui n'est pas toujours chose aisée quand le sujet d'expérience est abstrait et/ou complexe.

- *Le modèle paramétrique*

En lien étroit avec la stratégie définie précédemment et les conditions qui en découlent, la seconde étape de l'expérience consiste à établir le modèle paramétrique capable de générer des images, via le logiciel **Grasshopper**. Dans le cadre des chiffres, le but sera donc de simuler l'écriture de chacun des chiffres comme s'il était écrit à la main, en faisant varier sa forme. Cela permettra de mettre en place une première base de données constituée d'une multitude de variations de chacun de ces chiffres, d'ordre paramétrique.

- *Le réseau de neurones*

Ensuite, il y a lieu d'écrire le code (ou du moins, s'inspirer, modifier et compiler des codes existants) d'un réseau de neurones afin qu'il puisse accepter en entrée la base de données paramétriques précédente, et pouvoir apprendre correctement depuis celle-ci et générer à son tour un espace de solution, supposément plus riche.

- *Exploitation des résultats*

Enfin, une fois les images générées par l'étapes précédente, il convient de les analyser. Je peux alors établir si les critères A, B et C validant l'hypothèse sont respectés ou non, et dans quelle mesure.

Plusieurs **itérations** de certaines étapes (notamment la génération d'images par le réseau de neurones) permettront également de comparer les résultats entre eux, ainsi que les paramètres employés et les statistiques d'apprentissage qui en résultent.

Je réserve cependant la comparaison plus globale des parties entre elles dans la section suivante dédiée à l'analyse des résultats.

**Récapitulatif du processus de l'expérimentation :**

Partie 1 : les chiffres

a. Stratégie d'approche
b. Modèle paramétrique
c. Réseau de neurones
d. Résultats

Partie 2 : les formes

a. Stratégie d'approche
b. Modèle paramétrique
c. Réseau de neurones
d. Résultats

Partie 3 : la disposition

a. Stratégie d'approche
b. Modèle paramétrique
c. Réseau de neurones
d. Résultats

## 4.3 Partie 1 : les chiffres

Il est temps de s'attaquer à la première partie de l'expérimentation : la partie dite témoin visant à tester le processus mis en place.

### *4.3.a Mise en place de la stratégie*

Le choix de travailler sur les symboles que sont les chiffres permet déjà de restreindre le champ de variables, mais il faut encore apporter des précisions. Tout d'abord je me place évidemment dans le système que nous connaissons et utilisons tous, soit le système décimal, basé sur la combinaison des dix chiffres de base. De nombreuses écritures se partagent le principe du système décimal, mais là encore je me concentrerai sur ce que je connais : l'écriture latine ou arabe occidentale (0, 1, 2, 3, 4, 5, 6, 7, 8, et 9).

Les chiffres sont le langage de base des mathématiques et de nombreuses sciences, et d'actions dans la vie quotidienne, mais ils représentent surtout un exercice, un mécanisme d'apprentissage pour les plus jeunes. L'alphabet, indispensable pour l'instruction à la lecture, l'écriture et donc à l'éducation occidentale reste un ensemble de symboles complexes pour les plus petits. Les dix premiers chiffres sont donc nécessairement moins nombreux, et plus facile à mettre en relation avec des expériences de la vie (voici 2 stylos, 4 billes, 1 voiture…).

C'est là où je commence à entrevoir l'intérêt du choix des chiffres pour cette expérimentation. En effet, je recherche en tant qu'expérience témoin des éléments à la représentation plutôt complexe, tout en ayant une logique de dessin relativement simple. Ce dernier argument semble pertinent pour l'écriture des chiffres : celle-ci (telle que nous la connaissons) existe depuis plus de 1500 ans et résulte d'une optimisation naturelle par l'homme, qui par son utilisation répétée et par des millions d'individus différents permet d'aboutir à des formes et à un tracé « simple ».

Second point important, je me concentrerai ici sur l'écriture manuscrite, et non la version simplifiée de type numérique (voir ci-dessous). En effet avec le développement de l'imprimerie au XVème siècle puis de l'industrie du livre et enfin de l'informatique, l'aspect des chiffres, bien que repris des usages existants, s'est focalisé sur la simplicité et la lisibilité, plutôt que sur la facilité et la rapidité de dessin, l'automatisation mécanique permettant de s'en libérer.

0 1 2 3 4 5 6 7 8 9

L'écriture manuscrite des chiffres est donc très bien documentée quant à la manière de les tracer. On voit sur de nombreux manuels d'apprentissage les consignes de sens de tracé, de début et de fin, par quels points passer, etc. (voir figure 4.3.1).

Cependant, ces manuels poussent l'enfant à avoir comme modèle l'écriture numérique. Dans les faits, l'écriture manuscrite est beaucoup plus organique et moins stéréotypée, elle est marquée de « variations de taille, de structure, d'inclinaison et de trait au sein d'une même classe »[2] (une classe étant ici un même chiffre), (voir figure 4.3.2). Il sera donc question d'approcher cette écriture manuscrite plus riche et donc plus intéressante.

Les données à créer seront par conséquent ces chiffres tels que je les ai définis, avec un seul chiffre par image, et de nombreuses **itérations** de chaque chiffre. Chaque chiffre sera donc différent du précédent, avec des variations plus ou moins importantes et respectant les variations que respecterait une main humaine.

Comme l'objectif est une complexité visuelle et non dans la logique de tracé, il semble pertinent d'essayer d'avoir le même nombre de variables au sein de chaque chiffre, ce qui appartient au domaine de la paramétrisation que je vais aborder immédiatement.

---

[2] **Clément Chatelain**, (2006), « Extraction de séquences numériques dans des documents manuscrits quelconques », Thèse de doctorat à l'Université de Rouen, UFR de sciences et techniques, discipline informatique. Page 20.

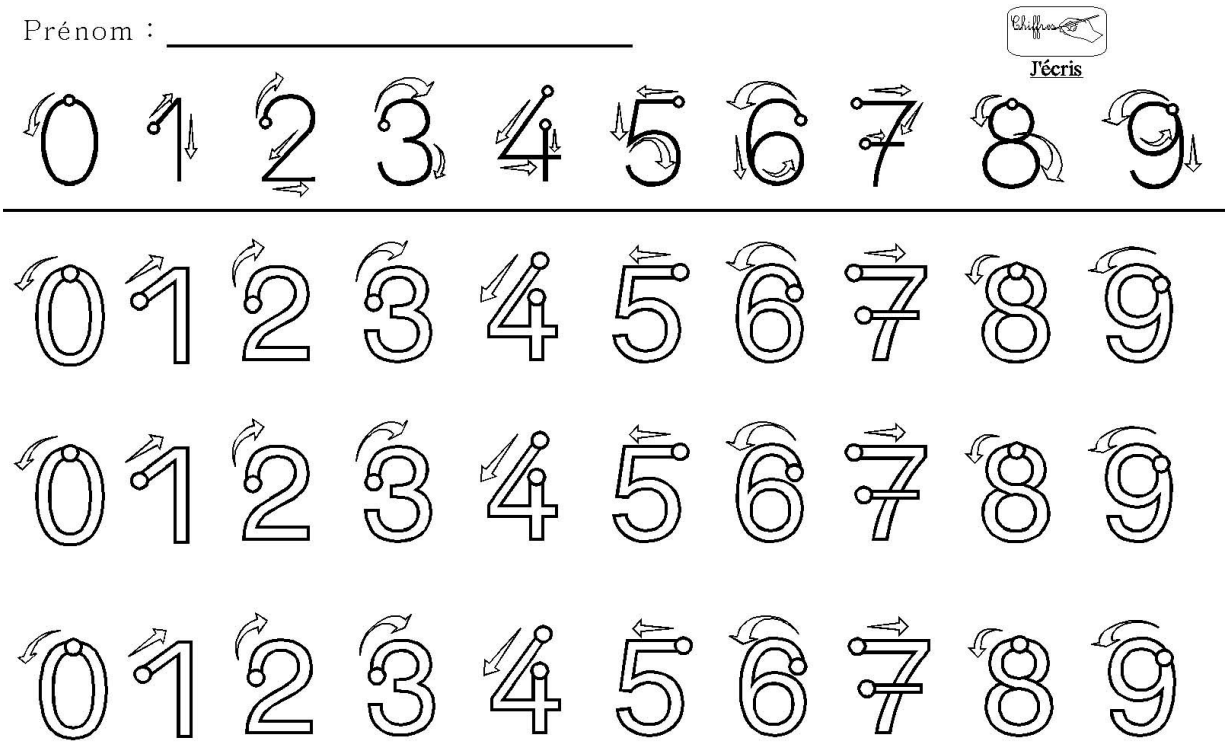



Fig. 4.3.1 - Fiche pour apprendre à travers des chiffres en maternelle et CP (source : site Web fiche-maternelle).

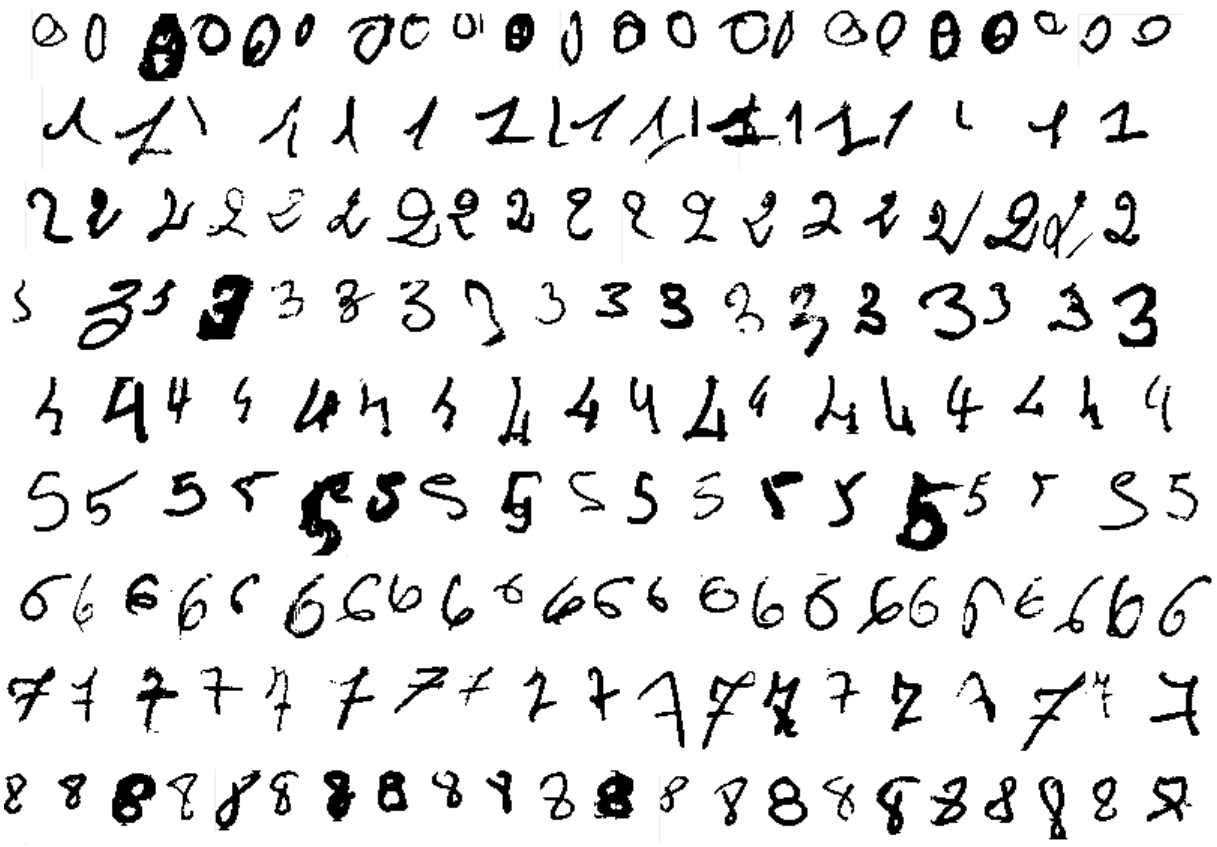


Fig. 4.3.2 - Variabilité des caractères manuscrits selon Clément Chatelain[2].

*4.3.b Le modèle paramétrique*

Il faut maintenant élaborer un modèle **Grasshopper** qui prenne en compte toutes les caractéristiques établies précédemment.

La programmation visuelle de **Grasshopper** étant parfois difficilement compréhensible, je propose dans cette partie de mettre en parallèle de l'aperçu **Grasshopper**, les explications de ce qu'il s'y passe mais aussi un schéma très simple des actions du modèle. Ils sont également tous disponibles en annexes, entiers et lisibles.

Ainsi, je peux observer en figure 4.3.4 le modèle en sa totalité. Bien qu'assez lourd en apparence, sa logique est assez simple (figure 4.3.3) : je commence par générer un grand nombre de variables (en l'occurrence, une grande quantité de chiffres), qui vont servir de base à la section « génération des chiffres », où les calculs de positions vont se faire. Enfin, une dernière section va venir s'occuper de l'aperçu de ses résultats, pour que je puisse les visionner dans la fenêtre de **Rhinocéros 3D**. Pour que ces résultats soient exploitables par la suite, une section va venir capturer la fenêtre **Rhino** et enregistrer chaque **itération** de chaque chiffre, de manière à constituer une première base de données paramétrique.

Le modèle global étant présenté, il convient d'aborder dans le détail les différentes grandes sections de ce modèle, comment elles fonctionnent et pourquoi avoir fait ces choix de paramétrisation.

Fig. 4.3.3 - Schéma global du modèle paramétrique.

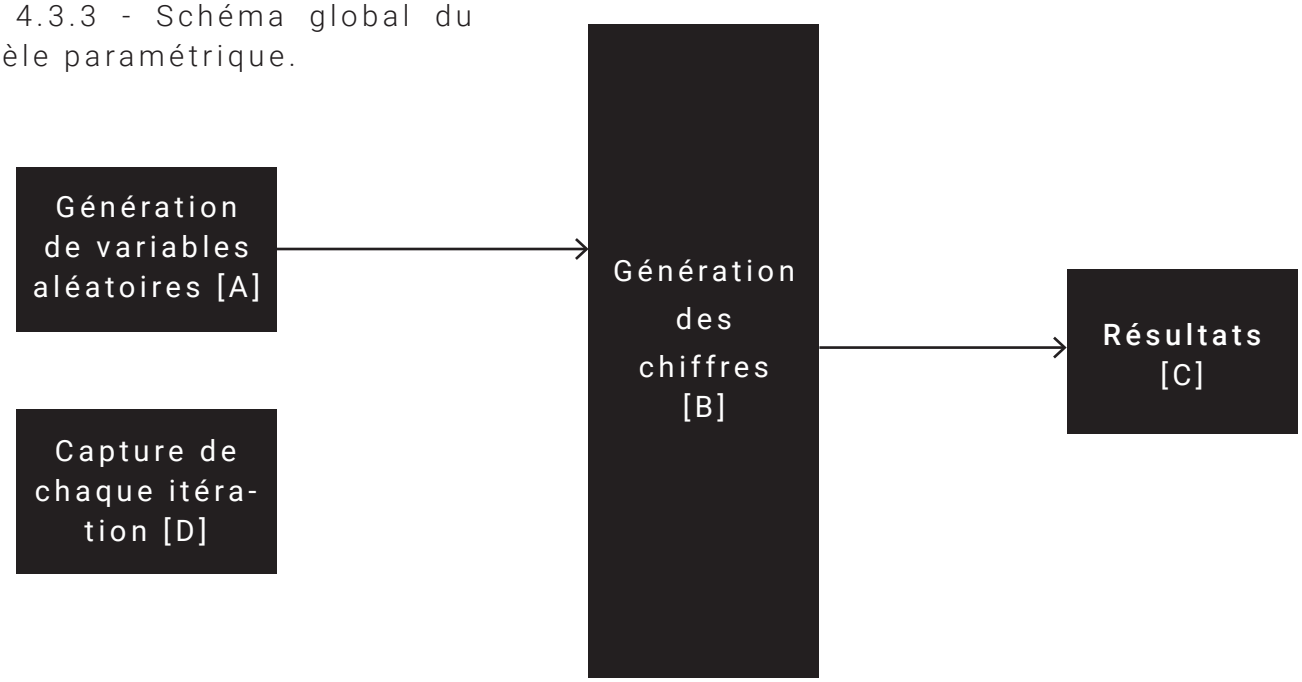
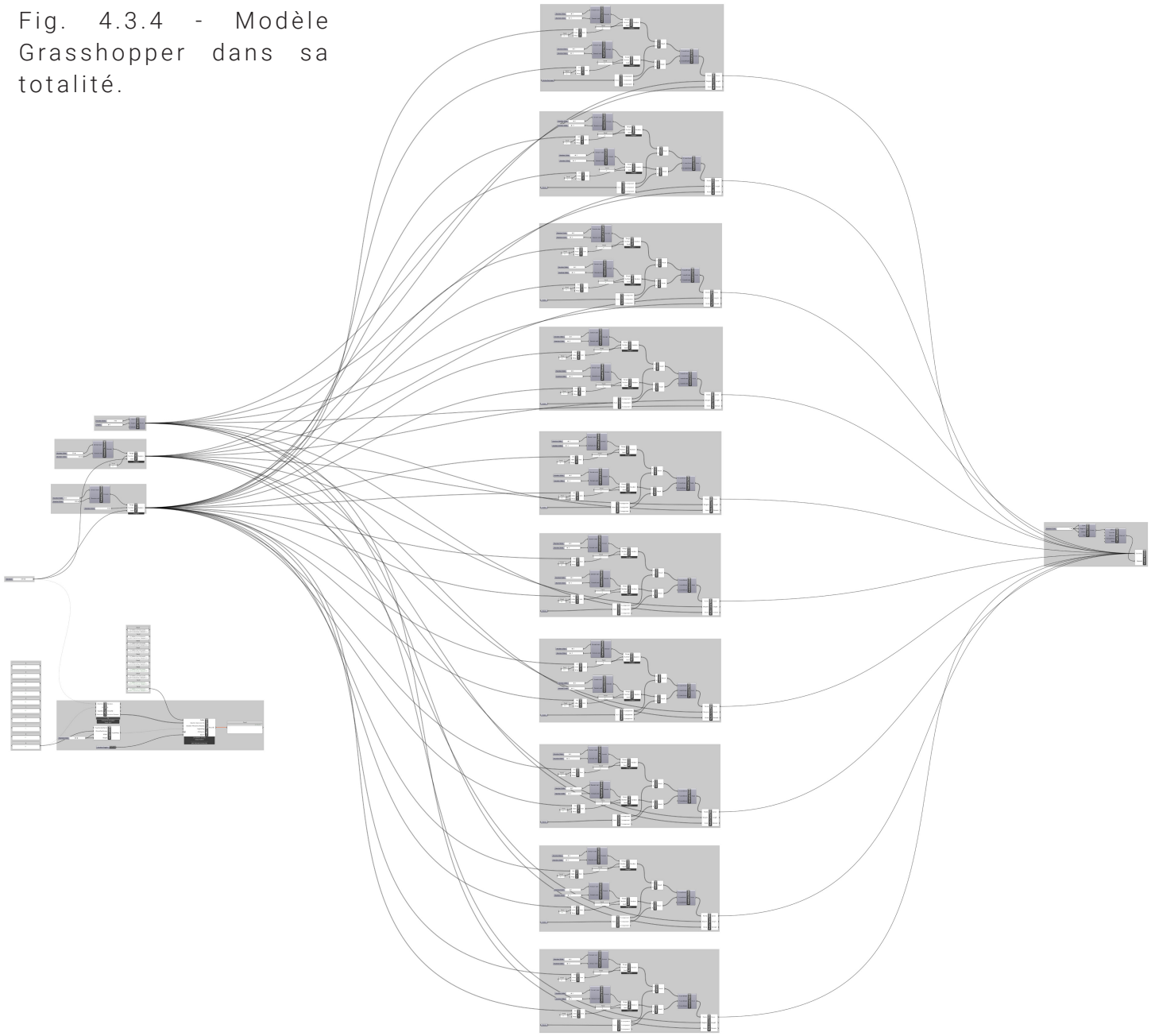



Fig. 4.3.4 - Modèle Grasshopper dans sa totalité.



[B]
Fig 4.3.3

Je peux commencer par m'intéresser au cœur même du modèle : la section qui calcule la forme de chaque chiffre (section [B] du schéma en figure 4.3.3). Ci-contre en figure 4.3.6 se trouve un zoom d'un des 10 blocs centraux, que l'on voyait en figure 4.3.4. Afin de mieux s'y retrouver, j'ai mis en place de nouveaux groupes (encadrés gris), chacun exécutant une tâche précise.

Pour comprendre cette section, il est plus aisé de partir de la fin, des résultats, puis de remonter au fil des différents liens. Cette logique inverse permet de savoir dès le début ce que je cherche et y répondre progressivement, plutôt que d'expliquer de nombreuses étapes et ne comprendre leur utilité qu'à la toute fin.

[1]
Fig 4.3.7

La section [1] a donc pour rôle de tracer le chiffre : il s'agit d'un composant « **NURBS Curve** PWK » (voir figure 4.3.6). Je ne vais pas détailler chaque composant de cette expérimentation, mais celui-ci est sans doute le plus important, et conditionne énormément les résultats de l'expérimentation, toutes parties confondues. Une **NURBS curve** (acronyme anglais pour Non-Uniform Rational Basis Spline) est une courbe dont la forme dépend de points de contrôle auxquels on peut affecter une certaine force d'attraction (le poids). On comprend alors le PWK, P pour Points, W pour Weight (poids) et K pour Knots (nœuds, paramètre qui gère les polynômes qui régissent la courbe mais qui ne seront pas détaillés ici car non déterminants dans le tracé du chiffre). Dans l'exemple proposé ci-dessous, on peut observer les conséquences graphiques du passage de poids uniformes (W=1 pour tous les points) à des poids différents.

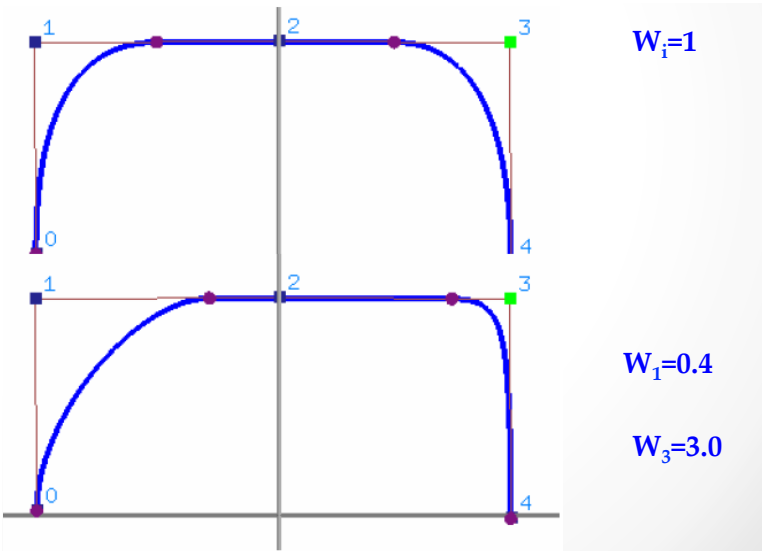



Fig. 4.3.5 - NURBS résultante aux poids variants (source : Christophe Pradal, cours INRIA).

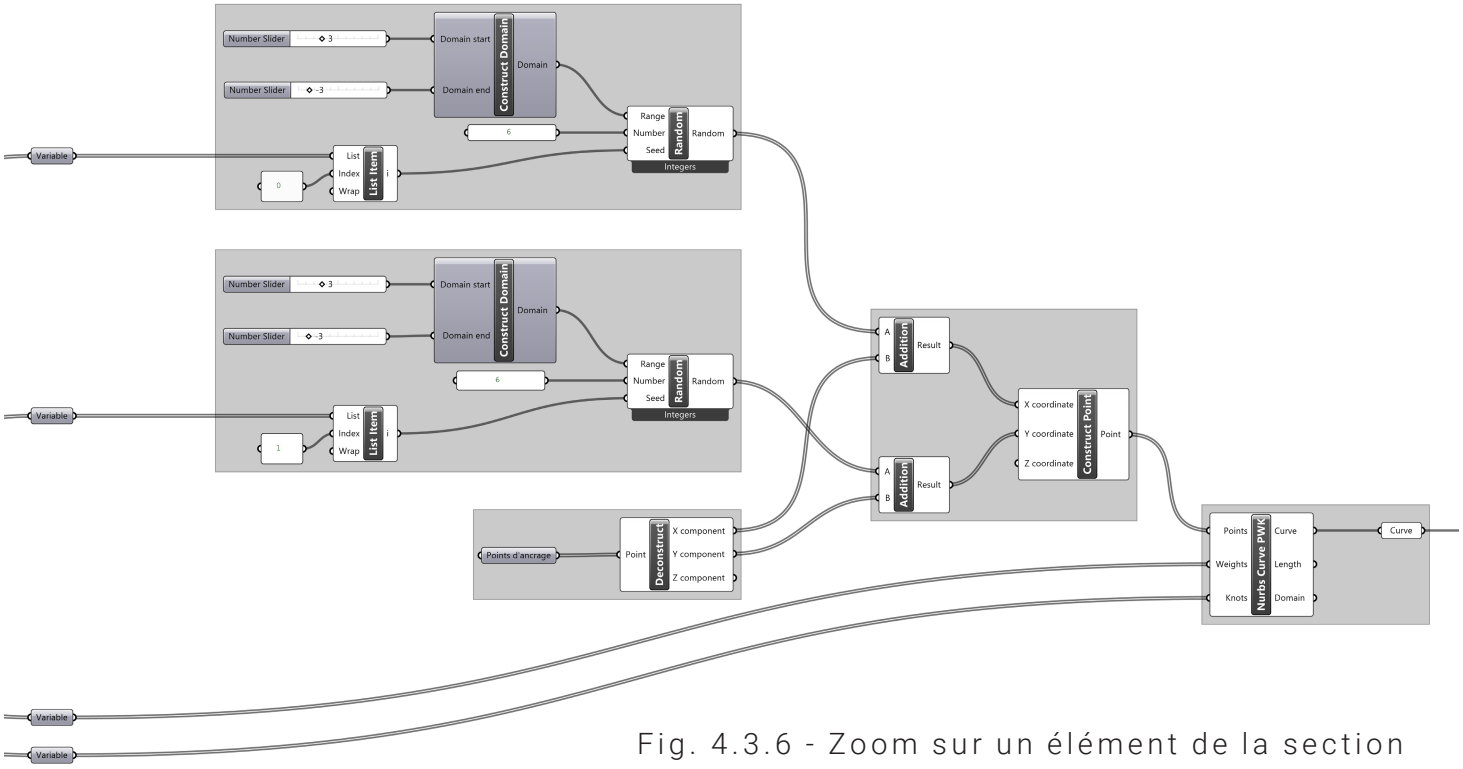



Fig. 4.3.6 - Zoom sur un élément de la section [B], dédié à la génération d'un chiffre.

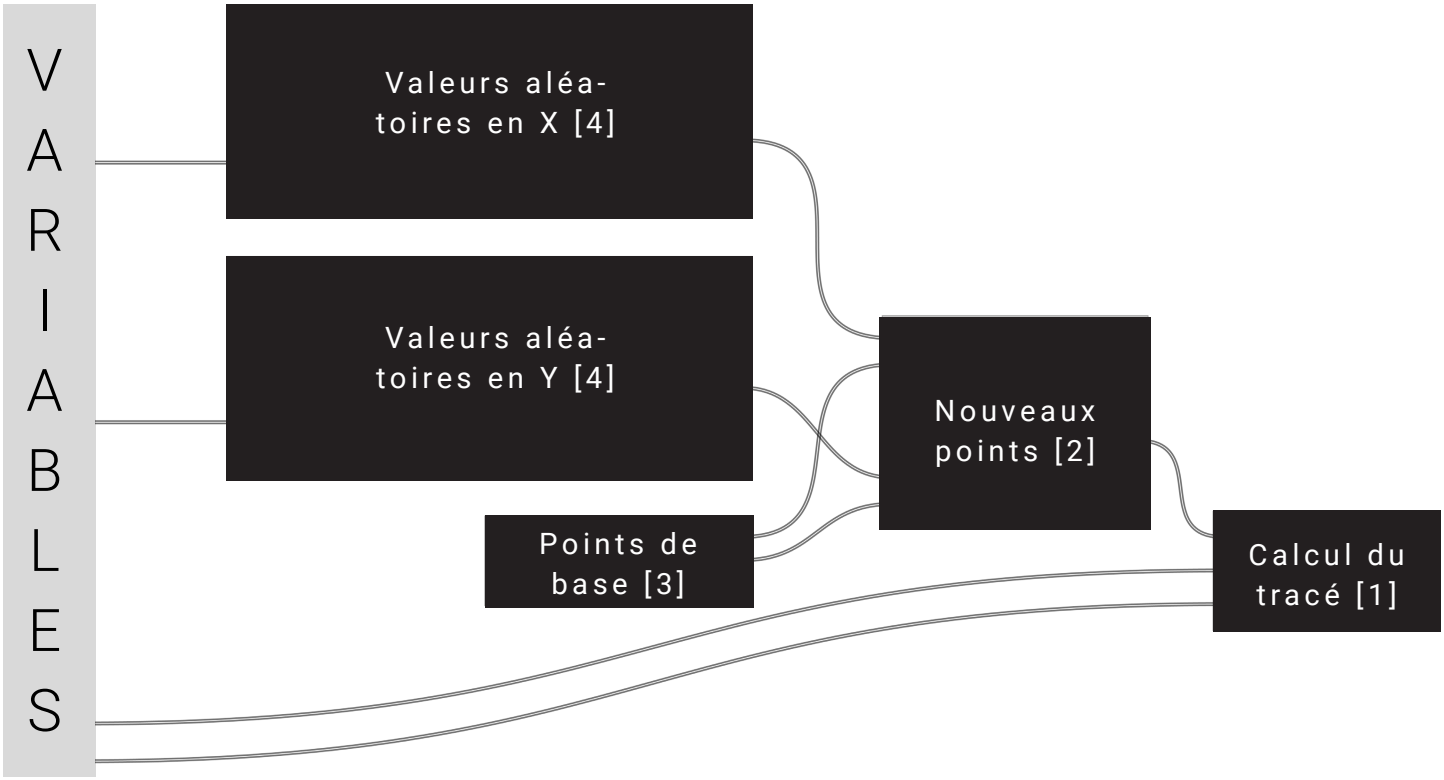



Fig. 4.3.7 - Schématisation de la figure précédente.

*Pourquoi le choix d'une courbe **NURBS** PWK plutôt qu'une autre ?*

La stratégie établie dans le chapitre précèdent évoquait le désir du rapprochement avec une écriture manuscrite et instinctive, en lien avec l'apprentissage que j'ai eu. Tout comme le fait une **NURBS**, la main qui dessine le chiffre tente de relier des points imaginaires qui nous servent de références (l'extrémité supérieure du 1 par exemple, les croisements du 8). Cependant nous ne sommes pas une machine, et rien ne garantit que l'on passera parfaitement par ces points imaginaires : on « tente » d'y passer, du mieux que je peux, avec les aléas de l'encre et du papier. Cette attirance imaginaire vers ces points peut tout à fait être matérialisée par les « poids » composant la **NURBS**.

Ainsi, il faut définir dans cette expérimentation ces fameux points imaginaires « idéaux », qui seront caractérisés par des poids variables. De plus, je vais appliquer des variations de position à ces points de base, car chacun a sa propre représentation mentale de ses points de passages quand il écrit, ce qui définit en grande partie son « style » (chiffres très ronds, ou au contraire raplapla…). Ce sont ces points que je définis au préalable et qui se trouvent dans la section [3] "points de base".

[3]
Fig 4.3.7

La figure ci-dessous met en image cette logique pour le chiffre 2 : à gauche les points de base, reliés symboliquement par des pointillés, et à droite la version "tracée" par le modèle, prenant en compte le déplacement des points d'ancrage et la variation des poids en chaque point. Le "2" obtenu est nécessairement plus petit, les poids n'étant jamais assez importants pour passer pleinement par le point de référence. Les points d'ancrage des autres chiffres sont en annexes. Chaque chiffre est fait avec le même nombre de points d'ancrages et de variables, afin de respecter une même logique de construction. Si certains points semblent manquer en annexe, ils sont en fait superposés pour insister plus fortement sur un même point.

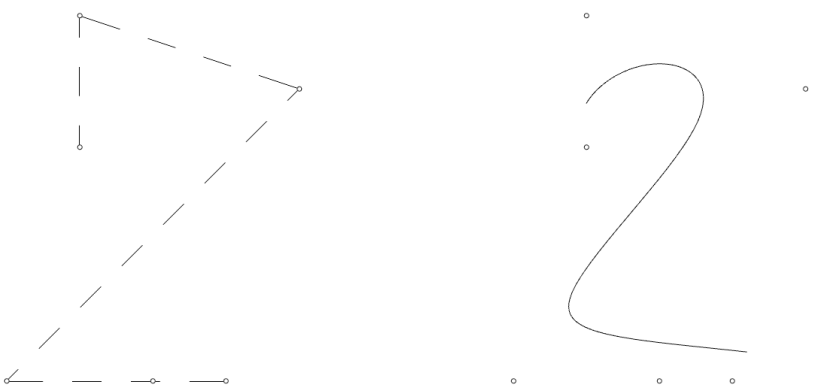


Fig. 4.3.8 - Points de base et points de base avec tracé paramétrique.

[4]
Fig 4.3.7

Pour se « détacher » des points de base, j'utilise un procédé très simple que l'on retrouve dans la section [4], « variation aléatoire ». J'y génère des nombres aléatoires, dans un domaine précis (de -3 à 3 en l'occurrence), qui vont venir s'ajouter (ou se soustraire) aux coordonnées des points de base. Pour rappel, un point est caractérisé dans un plan par 2 valeurs, en X et en Y, qui définissent sa position. Ainsi ajouter -2 ou +3 à ces coordonnées permet de déplacer légèrement ces points.
Je procède de la même façon pour générer des poids aléatoires, en mettant en place un domaine de variation et en allant y chercher des valeurs aléatoires.

[A]
Fig 4.3.3

Pour « chercher » ces valeurs aléatoires, là encore un travail est à faire. En effet, chaque composant « aléatoire » que j'emploie dans **Grasshopper** se base sur une **graine**, qu'il faut faire varier si je veux des valeurs aléatoires différentes à chaque **itération**. C'est ici qu'intervient la section [A] du schéma global (figure 4.3.3), présentée dans le détail ci-dessous. Je génère donc un grand nombre de variables dans des domaines définis, qui iront se brancher à tous les composants ayant besoin d'une **graine** dans le modèle, d'où la multiplication de liens fuyant vers la droite pour se connecter ailleurs. J'ai ici 3 groupes, un générant les nœuds (fixes), les variables des points et les variables des poids. Le bloc vert tout à gauche est le numéro d'**itération** (601 sur la figure), qui sert aussi de **graine** aux composants aléatoires. Multiplier tous ces composants aléatoires permet d'être sûr d'avoir du VRAI aléatoire, sans trouver de récurrence de variables qui pourrait fausser le jeu de donnée paramétrique.

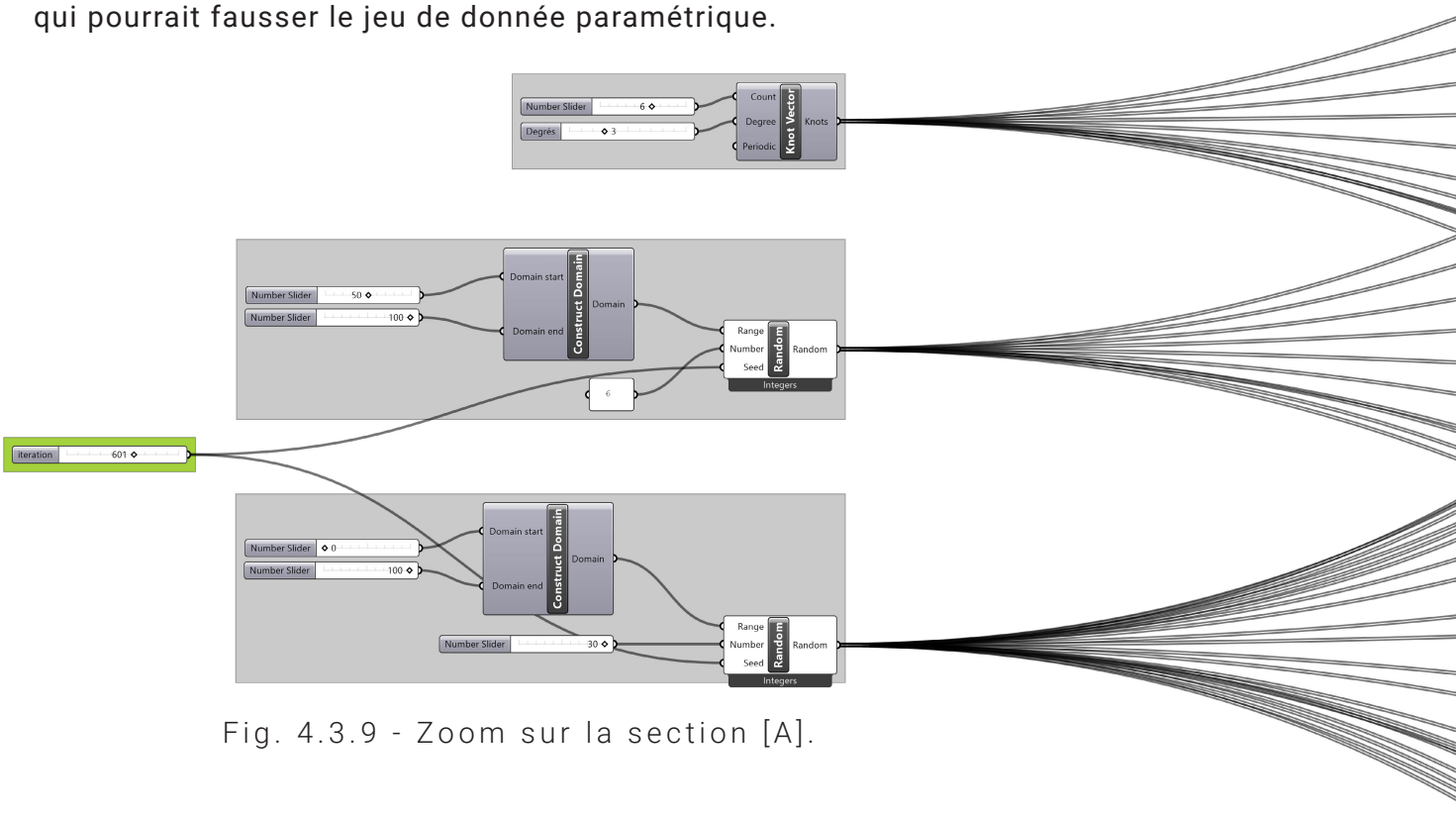



Fig. 4.3.9 - Zoom sur la section [A].

[C]
Fig 4.3.3

Enfin, la section [C], détailée du schéma global va venir générer l'aperçu des résultats. Bien que je puisse tout de même les voir dans la fenêtre **Rhino**, il est nécessaire de préciser leur couleur en vrai noir (valeur 1 pour le Cyan, Magenta et Yellow du **code couleur CMYK**), qui va permettre un export correct et exploitable.

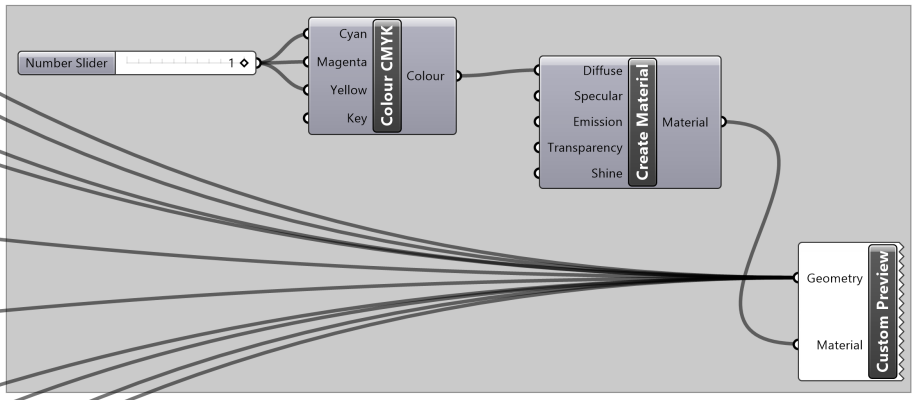



Fig. 4.3.10 - Zoom sur la section [C] du schéma global.

[D]
Fig 4.3.3

Dernier élément du modèle, la capture des éléments générés (section [D]). En effet je cherche à obtenir chaque **itération** de chaque chiffre sur une image indépendante. Deux solutions étaient envisageables : générer toutes les solutions, puis les convertir en image. Ou, produire une image à chaque solution générée. La seconde option est favorisée, permettant d'alléger la **computation**. La section [D] détaillée ci-contre tournera donc en parallèle du modèle, c'est même son moteur : j'utilise ici le **plugin** TT Toolbox, et plus précisément un outil nommé Colibri. Celui-ci va faire défiler les **itérations** qui, étant connectées aux variables, vont à leur tour déclencher les variations de chiffres. Colibri capture chaque chiffre à chaque **itération**, et le place dans un dossier précis que je détermine (tous les 0 ensemble, etc.). Je fixe la résolution de ces images à 50 pixels de côtés, résolution assez faible pour limiter le poids des données mais suffisant pour garantir une correcte lisibilité.

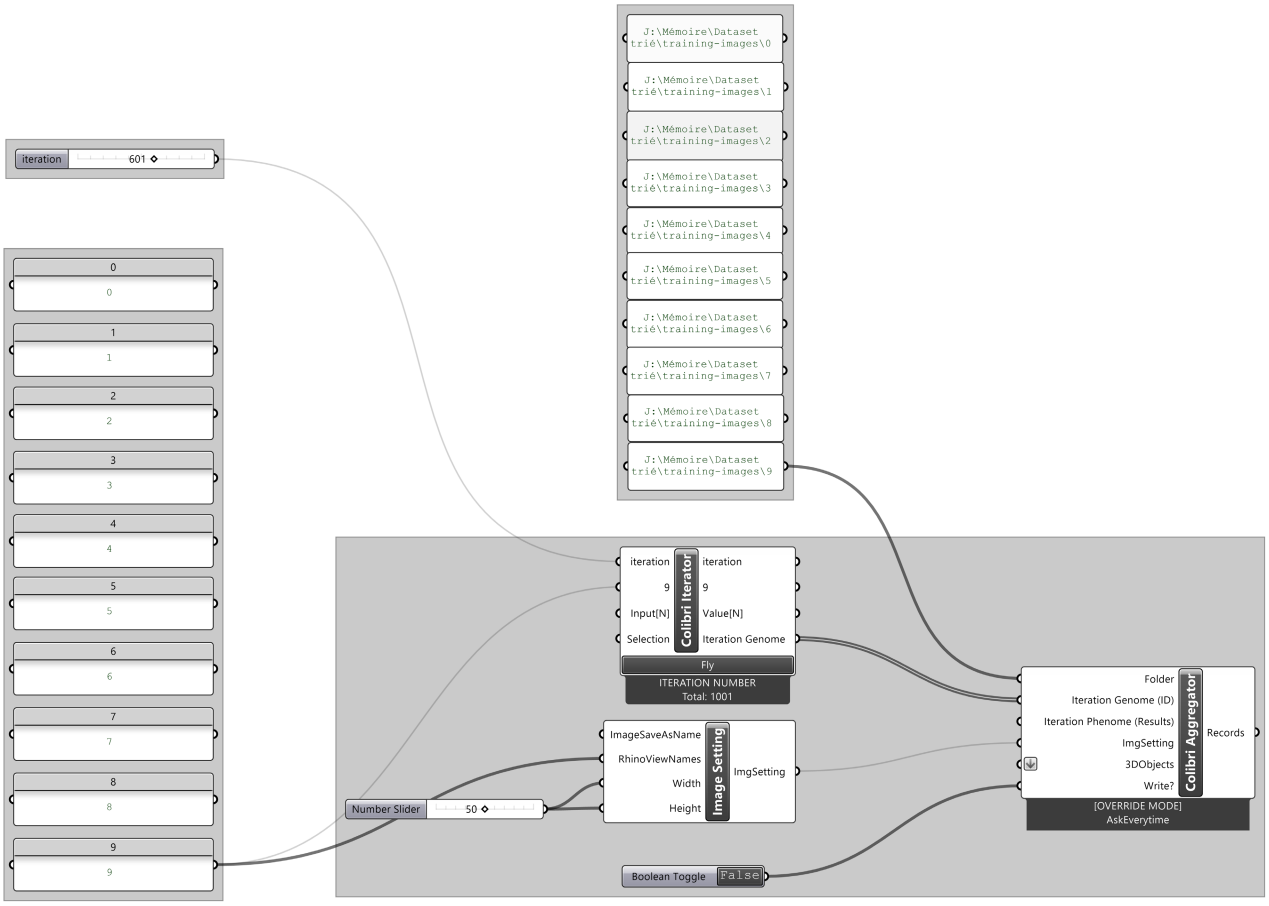



Fig. 4.3.11 - Zoom sur la section [D] du schéma global, la capture des images.

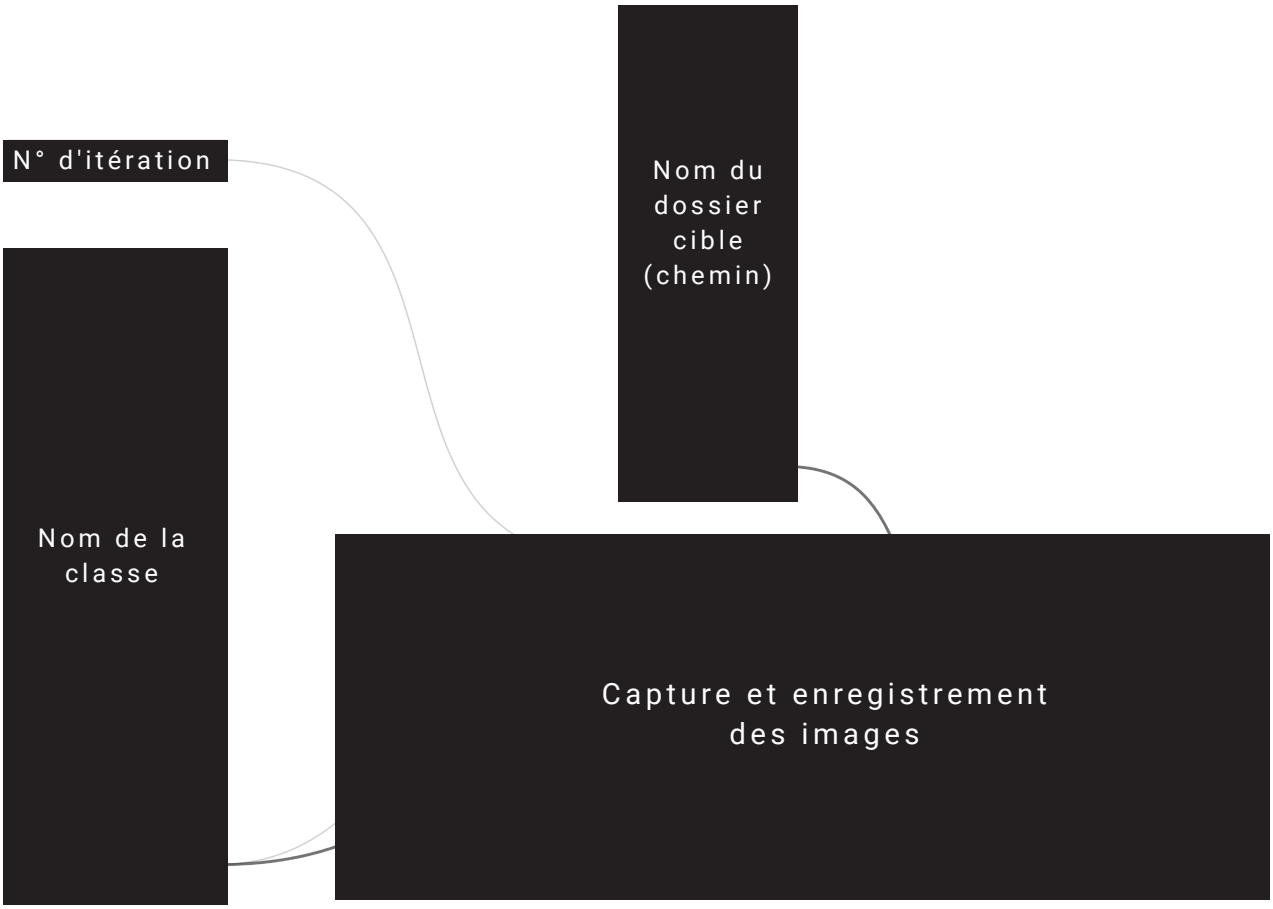



Fig. 4.3.12 - Schématisation de la figure précédente.

Pour conclure sur cette première approche paramétrique, je peux visualiser ce qui vient d'être produit. Voici donc un fragment des images générées, appartenant à un jeu de données paramétriques trié de 7000 images. Chaque colonne représente une classe de chiffre, et par conséquent un dossier regroupant tous ses semblables.

Montrer autant d'images n'est peut-être pas indispensable, mais toute l'énergie investie dans ce modèle paramétrique incite à en montrer autant que possible et à tout le monde.

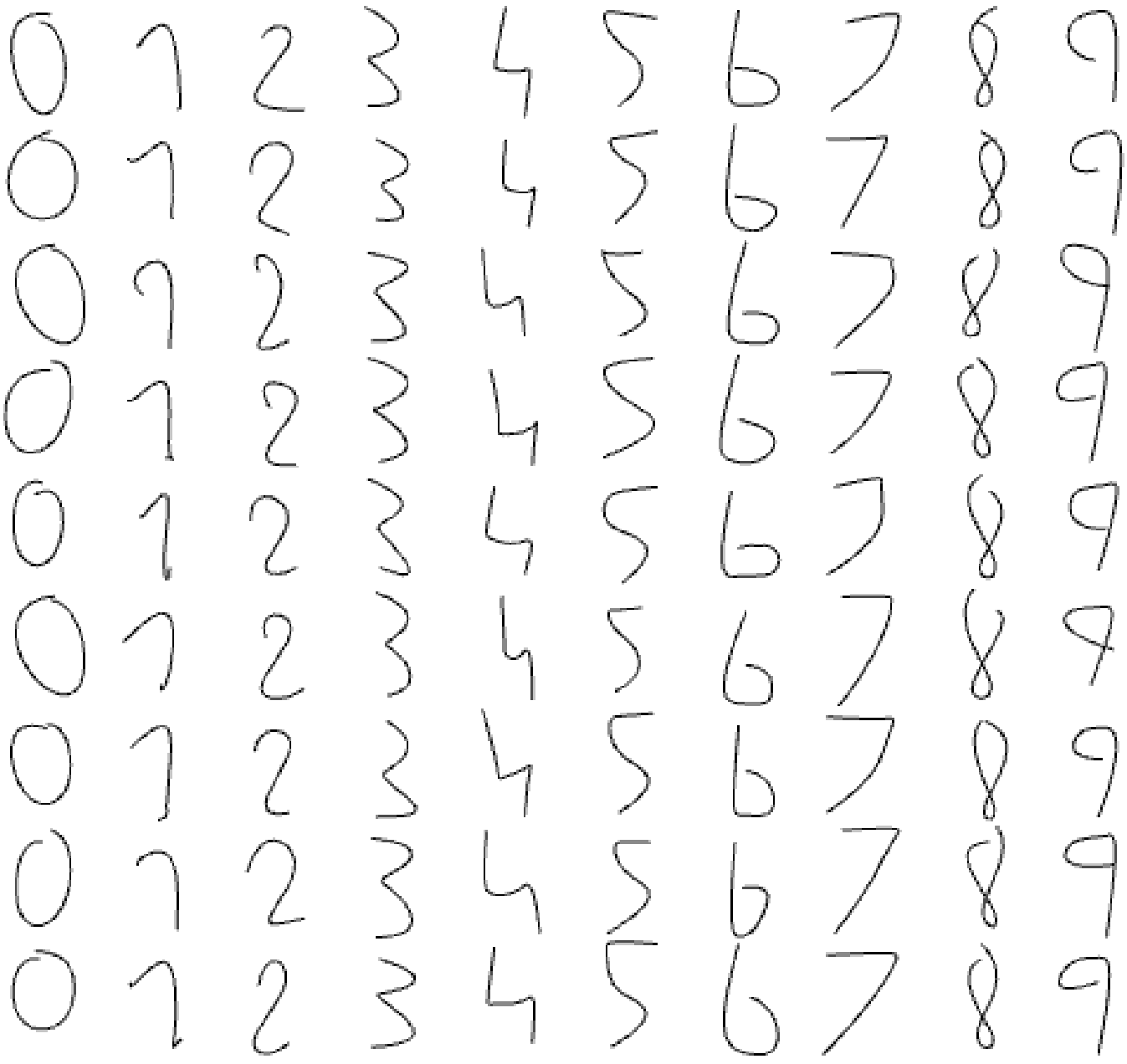
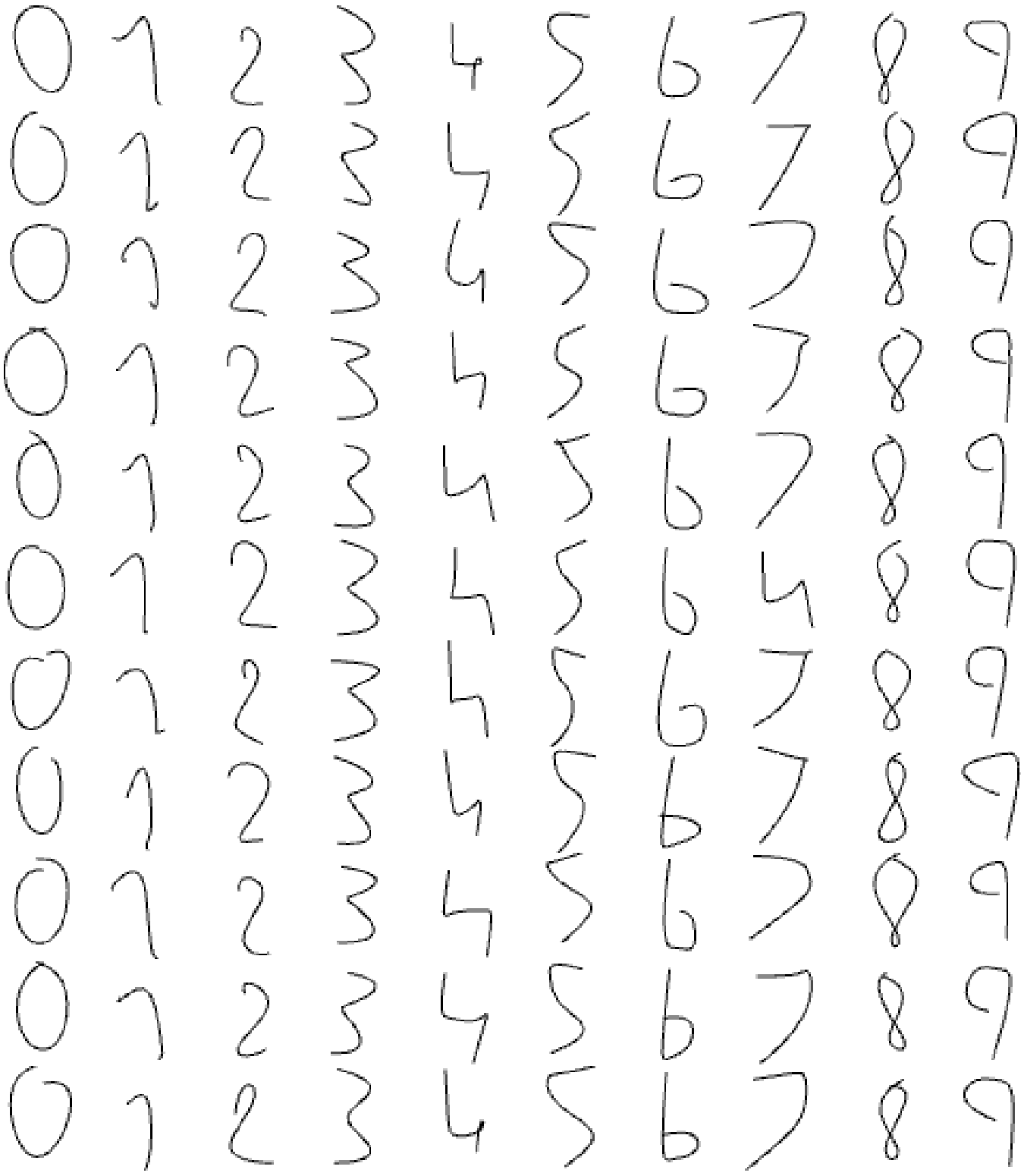


Fig. 4.3.13 - Exemples de chiffres générés par le modèle paramétrique.



A l'instar d'une écriture manuscrite, je trouve dans ces résultats plusieurs éléments caractéristiques. Par exemple pour le 8, la boucle supérieure est parfois correctement jointe, parfois pas assez ou encore trop. Je ressens l'ambiance d'un 8 dessiné à la va-vite. Plus de résultats sont disponibles en annexes.

*4.3.c Le réseau de neurones*

Il s'agit là de la partie la plus complexe de l'expérimentation, du moins de mon point de vue. En effet, bien que la programmation visuelle sur **Grasshopper** me soit familière, la programmation sous **Python** était quelque chose de tout à fait nouveau (mais passionnant) qu'il m'a fallu apprendre. De nombreuses tentatives ont été nécessaires, avec plusieurs approches, plusieurs réseaux de neurones, plusieurs supports. Je n'aborderai ainsi que la dernière série de tentatives, qui a eu le mérite de fonctionner.

Dans le cadre de cette expérimentation, je me base donc sur le travail de Jason Bronwlee, doctorant en Intelligence Artificielle, et dont les publications sur le site *Machine Learning Mastery* m'ont été d'une très grande aide. Le code qui sera abordé par la suite résulte donc en partie d'un travail de compréhension et d'adaptation de deux réseaux de neurones de Jason Brownlee (tous deux des **Generative Adversarial Networks ou GAN**), complété de modifications personnelles afin qu'il corresponde à mon application.

Avant d'analyser et d'expliquer en détail le réseau de neurones résultant de ces opérations, il est nécessaire d'aborder une étape intermédiaire, entre la base de données paramétrique précédemment générée et le **GAN**. En effet, il faut pouvoir « donner » les images de chiffres à ce réseau, et cette étape d'adaptation des données a également été complexe et a nécessité de nouveaux savoirs.

Tout d'abord, pour comprendre l'enjeu de cette étape intermédiaire, parlons du premier **GAN** sur lequel je me base[3] , donc le titre de l'article peut se traduire par « *Comment développer un **GAN** pour générer des chiffres manuscrits **MNIST*** ».

---

[3] **Jason Bronwlee**, (2019-2020), « How to Develop a GAN for Generating MNIST Handwritten Digits", site Web Machine Learning Mastery, section Generative Adversarial Networks, consulté le 19/06/20. Disponible sur : https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-an-mnist-handwritten-digits-from-scratch-in-keras/.

Ce titre semble très prometteur pour l'avancée du mémoire, mais un terme qui pourrait passer inaperçu va constituer un frein monumental : **MNIST**. En effet, cet article ne propose pas de créer un **GAN** apprenant d'images personnelles, mais d'une base de données libre de droits : la **base de données MNIST** (voir figure 4.3.14). Il s'agit d'une base de données regroupant 60 000 chiffres de 0 à 9 écrits à la main, optimisée et très répandue dans le domaine de l'apprentissage artificiel. Bien que pratique, la problématique de ce mémoire s'oriente néanmoins sur la création d'une base de données : une adaptation s'impose.

Fig. 4.3.14 - Fragment de la base de données **MNIST** : les chiffres ressemblent fortement aux miens ce qui est TRES bon signe (source : Madhu Ramiah, Medium).

Le **GAN** proposé par Bronwlee va « chargé » la base de données **MNIST**, avec une fonction prévue à cet effet qui va tout télécharger proprement, dans un format particulier : .idx3-ubyte. Pas besoin de s'y intéresser de près, c'est un format développé pour l'occasion par le créateur de **MNIST** et qui permet de compiler toutes les images en un seul fichier.

*Il faut donc convertir la base de données paramétrique au format attendu par le **GAN**.*

Pour y parvenir j'utilise un petit programme "*JPG-PNG-to-MNIST-NN-Format*"[4], dont l'entièreté du code est disponible sur le site de partage **GitHub**. Là encore, quelques adaptations ont été nécessaires afin de le rendre pleinement compatible avec mon **système d'exploitation** Windows (code prévu initialement pour Linux et Mac).

Je place donc mes images générées dans une série de dossiers sur mon disque dur, dont la hiérarchie est à respecter strictement :

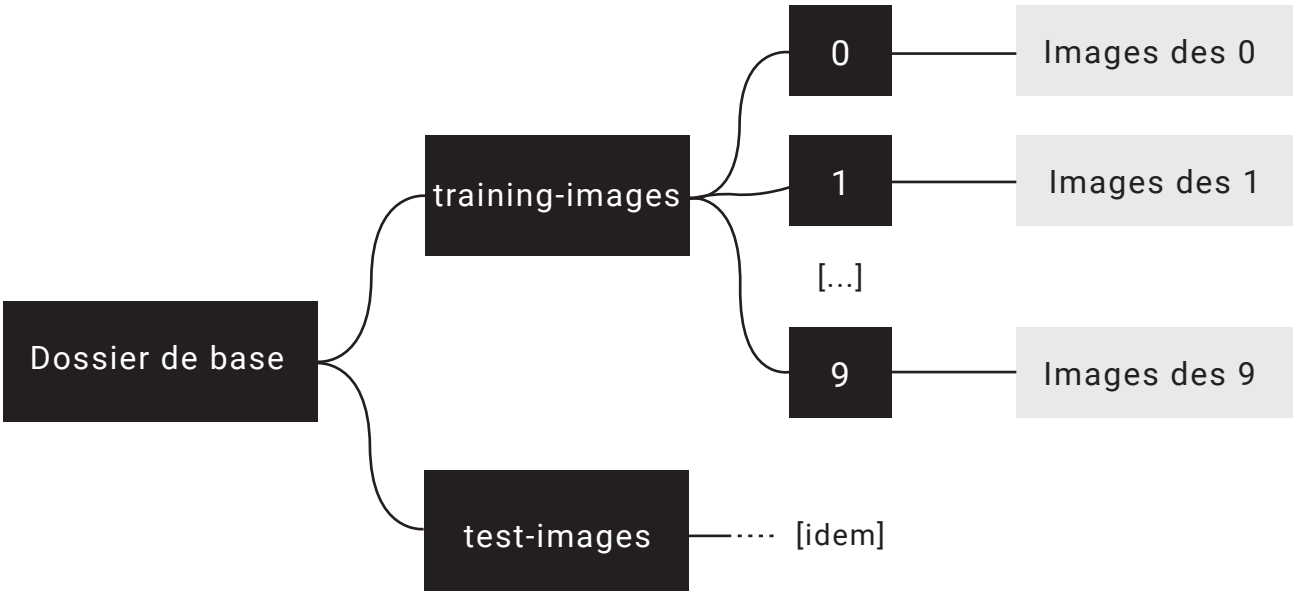
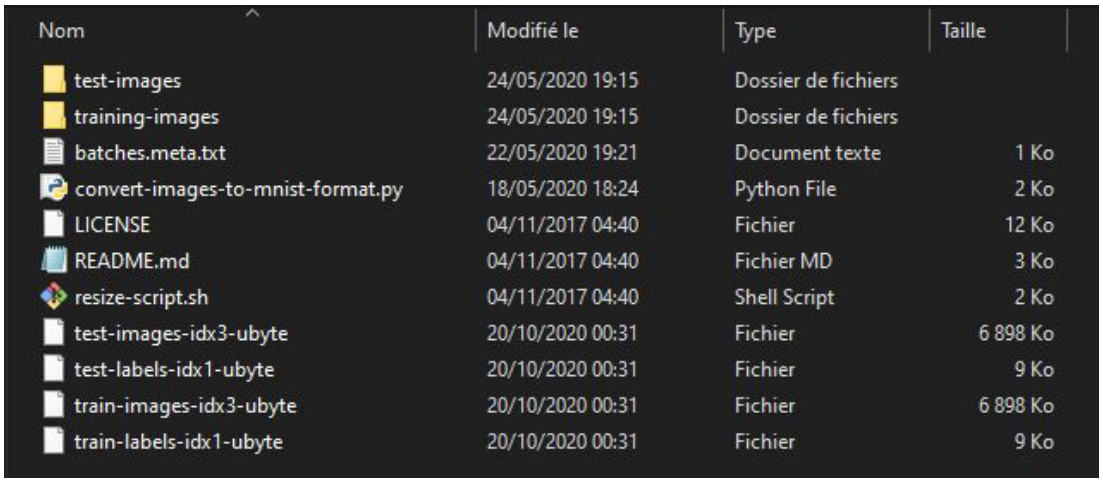



Fig. 4.3.15 - Schéma de la hiérarchie de dossiers nécessaire.

On remarque la séparation entre un jeu de données dit d'entraînement (train) et de test, éléments indispensables au **GAN** comme cela a été abordé dans l'état de connaissances.

Une fois le programme lancé à l'aide de l'**invite de commandes**, plusieurs fichiers au format .idx3-ubyte sont créés. En figure 4.3.16 on voit bien ces 4 nouveaux fichiers, ainsi que les dossiers test et training-images et le fichier à lancer avec l'**invite de commande** (avec l'extension .py car il est rédigé en langage **Python**).

[4] **Gregory S. Kielan**, "JPG-PNG-to-MNIST-NN-Format", Site Web GitHub, consulté le 15/08/20. Disponible sur : https://github.com/gskielian/JPG-PNG-to-MNIST-NN-Format.



| Nom | Modifié le | Type | Taille |
|---|---|---|---|
| test-images | 24/05/2020 19:15 | Dossier de fichiers | |
| training-images | 24/05/2020 19:15 | Dossier de fichiers | |
| batches.meta.txt | 22/05/2020 19:21 | Document texte | 1 Ko |
| convert-images-to-mnist-format.py | 18/05/2020 18:24 | Python File | 2 Ko |
| LICENSE | 04/11/2017 04:40 | Fichier | 12 Ko |
| README.md | 04/11/2017 04:40 | Fichier MD | 3 Ko |
| resize-script.sh | 04/11/2017 04:40 | Shell Script | 2 Ko |
| test-images-idx3-ubyte | 20/10/2020 00:31 | Fichier | 6 898 Ko |
| test-labels-idx1-ubyte | 20/10/2020 00:31 | Fichier | 9 Ko |
| train-images-idx3-ubyte | 20/10/2020 00:31 | Fichier | 6 898 Ko |
| train-labels-idx1-ubyte | 20/10/2020 00:31 | Fichier | 9 Ko |

Fig. 4.3.16 - Capture d'écran du "dossier de base".

Il ne reste alors plus qu'à charger des **ubytes** dans le **GAN**, ce qui là encore requiert une adaptation. Cette étape faisant vraiment appel au code dans sa rédaction, il est plus simple d'avoir la programmation directement sous les yeux : j'y reviendrai sous peu.

Ainsi, il est temps de plonger au cœur de la programmation, ou du moins d'en survoler les éléments principaux et qui sont des plus intéressants dans le cadre de cette expérience.

Tout comme pour le modèle **Grasshopper**, le code dans son entièreté (figure 4.3.17) rentre difficilement dans une page, et son explication va nécessiter plusieurs étapes. Il est disponible en annexe 9.4, car illisible lorsqu'il est présenté en une seule page. Bien sûr il n'est pas question d'aborder chaque ligne de code, mais plutôt les sections structurantes du code et certains détails de son fonctionnement. Malgré tout, l'intégralité du code est disponible de façon lisible en annexe. 4 grandes parties se distinguent donc dans le schéma global de cet **algorithme**, schéma assez courant dans la programmation sous **Python** et **Keras** : je commence par l'import des éléments nécessaires à la programmation, je définis les fonctions et variables, je les lance et enfin je visualise les résultats.

[A]
Fig 4.3.18

```
# example of training a stable gan for generating a handwritten digit
from os import makedirs
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import BatchNormalization
from keras.initializers import RandomNormal
from matplotlib import pyplot
```

Fig. 4.3.16 - Zoom sur la section [A] : import des modules.

L'interface **Jupyter Notebook** a l'amabilité de colorer certains éléments pour faciliter la lecture du code. Ainsi dans la section dédiée à l'import des modules, ces derniers sont mis en avant.

- En vert, je reconnais l'usage d'une fonction. C'est un peu les "verbes" du langage de programmation, sa façon d'exercer des actions. Ainsi "import" est une fonction qui vise à importer du contenu dans l'**algorithme**.
- En bleu sont indiqués les modules importés, c'est à dire des ensembles de fonctions qui ne sont pas présents dans **Python** de base.

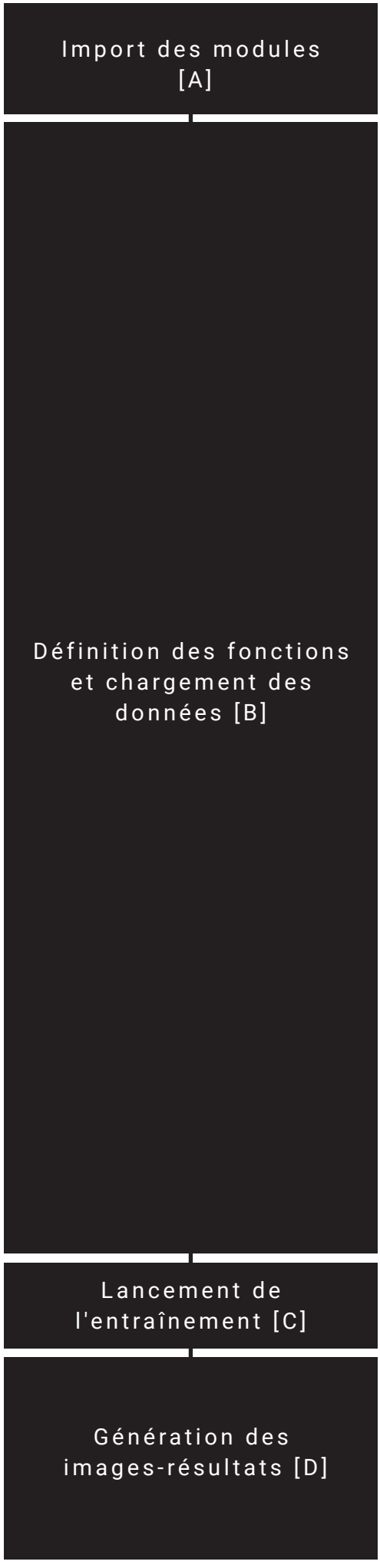
Ainsi la ligne "*from numpy import randn*" veut dire : j'importe la fonction randn (génération de nombres aléatoires) depuis le module **Numpy** dans le code.

```
In [ ]: # example of training a stable gan for generating a handwritten digit
        from os import makedirs
        from numpy import expand_dims
        from numpy import zeros
        from numpy import ones
        from numpy.random import randn
        from numpy.random import randint
        from keras.datasets.mnist import load_data
        from keras.optimizers import Adam
        from keras.models import Sequential
        from keras.layers import Dense
        from keras.layers import Reshape
        from keras.layers import Flatten
        from keras.layers import Conv2D
        from keras.layers import Conv2DTranspose
        from keras.layers import LeakyReLU
        from keras.layers import BatchNormalization
        from keras.initializers import RandomNormal
        from matplotlib import pyplot

        # define the standalone discriminator model
        def define_discriminator(in_shape=(28,28,1)):
                # weight initialization
                init = RandomNormal(stddev=0.02)
                # define model
                model = Sequential()
                # downsample to 14x14
                model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init, input_shape=in_shape))
                model.add(BatchNormalization())
                model.add(LeakyReLU(alpha=0.2))
                # downsample to 7x7
                model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
                model.add(BatchNormalization())
                model.add(LeakyReLU(alpha=0.2))
                # classifier
                model.add(Flatten())
                model.add(Dense(1, activation='sigmoid'))
                # compile model
                opt = Adam(lr=0.0002, beta_1=0.5)
                model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
                return model

        # define the standalone generator model
        def define_generator(latent_dim):
                # weight initialization
                init = RandomNormal(stddev=0.02)
                # define model
                model = Sequential()
                # foundation for 7x7 image
                n_nodes = 128 * 7 * 7
                model.add(Dense(n_nodes, kernel_initializer=init, input_dim=latent_dim))
                model.add(LeakyReLU(alpha=0.2))
                model.add(Reshape((7, 7, 128)))
                # upsample to 14x14
                model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
                model.add(BatchNormalization())
                model.add(LeakyReLU(alpha=0.2))
                # upsample to 28x28
                model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
                model.add(BatchNormalization())
                model.add(LeakyReLU(alpha=0.2))
                # output 28x28x1
                model.add(Conv2D(1, (7,7), activation='tanh', padding='same', kernel_initializer=init))
                return model

        # define the combined generator and discriminator model, for updating the generator
        def define_gan(generator, discriminator):
                # make weights in the discriminator not trainable
                discriminator.trainable = False
                # connect them
                model = Sequential()
                # add generator
                model.add(generator)
                # add the discriminator
                model.add(discriminator)
                # compile model
                opt = Adam(lr=0.0002, beta_1=0.5)
                model.compile(loss='binary_crossentropy', optimizer=opt)
                return model


        import mlxtend
        import numpy as np

        from mlxtend.data import loadlocal_mnist

        trainX, trainy = loadlocal_mnist(
                images_path='/Users/IR/Desktop/Ubytes copie/train-images-idx1-ubyte',
                labels_path='/Users/IR/Desktop/Ubytes copie/train-labels-idx1-ubyte')

        print(trainX.shape[0], trainX.shape[1])
        trainX=np.reshape(trainX,(trainX.shape[0],28, -1))
        print(trainX.shape[0], trainX.shape[1], trainX.shape[2])


        # load mnist images
        def load_real_samples():
                # load dataset
                # expand to 3d, e.g. add channels


                X = expand_dims(trainX, axis=-1)
                # select all of the examples for a given class
                selected_ix = trainy
                X = X[selected_ix]
                # convert from ints to floats
                X = X.astype('float32')
                # scale from [0,255] to [-1,1]
                X = (X - 127.5) / 127.5
                return X

        # select real samples
        def generate_real_samples(dataset, n_samples):
                # choose random instances
                ix = randint(0, dataset.shape[0], n_samples)
                # select images
                X = dataset[ix]
                # generate class labels
                y = ones((n_samples, 1))
                return X, y

        # generate points in latent space as input for the generator
        def generate_latent_points(latent_dim, n_samples):
                # generate points in the latent space
                x_input = randn(latent_dim * n_samples)
                # reshape into a batch of inputs for the network
                x_input = x_input.reshape(n_samples, latent_dim)
                return x_input

        # use the generator to generate n fake examples, with class labels
        def generate_fake_samples(generator, latent_dim, n_samples):
                # generate points in latent space
                x_input = generate_latent_points(latent_dim, n_samples)
                # predict outputs
                X = generator.predict(x_input)
                # create class labels
                y = zeros((n_samples, 1))
                return X, y

        # generate samples and save as a plot and save the model
        def summarize_performance(step, g_model, latent_dim, n_samples=100):
                # prepare fake examples
                X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
                # scale from [-1,1] to [0,1]
                X = (X + 1) / 2.0
                # plot images
                for i in range(10 * 10):
                        # define subplot
                        pyplot.subplot(10, 10, 1 + i)
                        # turn off axis
                        pyplot.axis('off')
                        # plot raw pixel data
                        pyplot.imshow(X[i, :, :, 0], cmap='gray')
                # save plot to file
                pyplot.savefig('results_baseline/generated_plot_%04d.png' % (step+1))
                pyplot.close()
                # save the generator model
                g_model.save('results_baseline/model_%04d.h5' % (step+1))

        # create a line plot of loss for the gan and save to file
        def plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist):
                # plot loss
                pyplot.subplot(2, 1, 1)
                pyplot.plot(d1_hist, label='d-real')
                pyplot.plot(d2_hist, label='d-fake')
                pyplot.plot(g_hist, label='gen')
                pyplot.legend()
                # plot discriminator accuracy
                pyplot.subplot(2, 1, 2)
                pyplot.plot(a1_hist, label='acc-real')
                pyplot.plot(a2_hist, label='acc-fake')
                pyplot.legend()
                # save plot to file
                pyplot.savefig('results_baseline/plot_line_plot_loss.png')
                pyplot.close()

        # train the generator and discriminator
        def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=10, n_batch=128):
                # calculate the number of batches per epoch
                bat_per_epo = int(dataset.shape[0] / n_batch)
                print(bat_per_epo)
                # calculate the total iterations based on batch and epoch
                n_steps = bat_per_epo * n_epochs
                # calculate the number of samples in half a batch
                half_batch = int(n_batch / 2)
                # prepare lists for storing stats each iteration
                d1_hist, d2_hist, g_hist, a1_hist, a2_hist = list(), list(), list(), list(), list()
                # manually enumerate epochs
                for i in range(n_steps):
                        # get randomly selected 'real' samples
                        X_real, y_real = generate_real_samples(dataset, half_batch)
                        # update discriminator model weights
                        d_loss1, d_acc1 = d_model.train_on_batch(X_real, y_real)
                        # generate 'fake' examples
                        X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
                        # update discriminator model weights
                        d_loss2, d_acc2 = d_model.train_on_batch(X_fake, y_fake)
                        # prepare points in latent space as input for the generator
                        X_gan = generate_latent_points(latent_dim, n_batch)
                        # create inverted labels for the fake samples
                        y_gan = ones((n_batch, 1))
                        # update the generator via the discriminator's error
                        g_loss = gan_model.train_on_batch(X_gan, y_gan)
                        # summarize loss on this batch
                        print('>%d, d1=%.3f, d2=%.3f g=%.3f, a1=%d, a2=%d' %
                                (i+1, d_loss1, d_loss2, g_loss, int(100*d_acc1), int(100*d_acc2)))
                        # record history
                        d1_hist.append(d_loss1)
                        d2_hist.append(d_loss2)
                        g_hist.append(g_loss)
                        a1_hist.append(d_acc1)
                        a2_hist.append(d_acc2)
                        # evaluate the model performance every 'epoch'
                        if (i+1) % bat_per_epo == 0:
                                summarize_performance(i, g_model, latent_dim)
                plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist)

        # make folder for results
        makedirs('results_baseline', exist_ok=True)
        # size of the latent space
        latent_dim = 50
        # create the discriminator
        discriminator = define_discriminator()
        # create the generator
        generator = define_generator(latent_dim)
        # create the gan
        gan_model = define_gan(generator, discriminator)
        # load image data
        dataset = load_real_samples()
        print(dataset.shape)
        # train model
        train(generator, discriminator, gan_model, dataset, latent_dim)
```

Fig. 4.3.17 - Code complet.

```
In [ ]: # example of loading the generator model and generating images
        from keras.models import load_model
        from numpy.random import randn
        from matplotlib import pyplot
        import matplotlib.pyplot as plt

        # generate points in latent space as input for the generator
        def generate_latent_points(latent_dim, n_samples):
                # generate points in the latent space
                x_input = randn(latent_dim * n_samples)
                # reshape into a batch of inputs for the network
                x_input = x_input.reshape(n_samples, latent_dim)
                return x_input

        # create and save a plot of generated images (reversed grayscale)
        def save_plot(examples, n):
                # plot images
                for i in range(n * n):
                        # define subplot
                        pyplot.subplot(n, n, 1 + i)
                        # turn off axis
                        pyplot.axis('off')
                        # plot raw pixel data
                        pyplot.imshow(examples[i, :, :, 0], cmap='gray')
                        pyplot.savefig('plot_300dpi.png', dpi=300)
                pyplot.show()

        # load model
        model = load_model('results_baseline/model_700.h5')
        # generate images
        latent_points = generate_latent_points(50, 100)
        # generate images
        X = model.predict(latent_points)
        # plot the result
        save_plot(X, 10)
```

Fig. 4.3.18 - Schéma global.



[B]
Fig 4.3.18

Même en zoomant sur la section [B] du schéma global du code (figure 4.3.18), celle-ci reste immense et quasiment illisible. En effet, le moment de définition des fonctions est la partie la plus laborieuse : c'est là réellement où se situe tout le travail de constitution de l'**algorithme** et de ses paramètres. Aussi je ne vais m'intéresser ici qu'à son schéma avec quelques explications, et le détail de certains paramètres clés.

Cette section [B] est ainsi consacrée à la définition des fonctions. En effet dans **Python** je peux avoir recours aux fonctions courantes (les fameuses if, while, prompt…), importer des fonctions externes comme il a été fait dans la section [A] ou enfin d'élaborer des fonctions soit même. Cela permet de créer précisément ce que je souhaite dans un premier temps, puis de pouvoir y faire référence à n'importe quel moment en citant le nom de la fonction. Ainsi dans la sous-section [1] du schéma (figure 4.3.21) détaillée ci-dessous, je peux voir la création du **discriminateur**, autrement dit la définition de la fonction **discriminateur** (pour rappel un **GAN** est constitué d'un **discriminateur** et d'un **générateur** qui vont être mis en concurrence, voir chapitre dédié à l'état des connaissances). Chaque ligne de code effectue une action spécifique (exceptées les lignes commençant par un symbole # signifiant une indication textuelle), mais inutile de s'y attarder, je ne les ai que très peu ou pas modifiées dans cette section ci.

```
def define_discriminator(in_shape=(28,28,1)):
	# weight initialization
	init = RandomNormal(stddev=0.02)
	# define model
	model = Sequential()
	# downsample to 14x14
	model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init, input_shape=in_shape))
	model.add(BatchNormalization())
	model.add(LeakyReLU(alpha=0.2))
	# downsample to 7x7
	model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
	model.add(BatchNormalization())
	model.add(LeakyReLU(alpha=0.2))
	# classifier
	model.add(Flatten())
	model.add(Dense(1, activation='sigmoid'))
	# compile model
	opt = Adam(lr=0.0002, beta_1=0.5)
	model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
	return model
```

Fig. 4.3.19 - Zoom sur la section [1] : définition du **discriminateur**.

Après la définition du **discriminateur** je définis logiquement le **générateur** en section [2], puis en section [3] la manière dont ils vont fonctionner en concurrence, en simultanéité.

```python
# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
	# weight initialization
	init = RandomNormal(stddev=0.02)
	# define model
	model = Sequential()
	# downsample to 14x14
	model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init, input_shape=in_shape))
	model.add(BatchNormalization())
	model.add(LeakyReLU(alpha=0.2))
	# downsample to 7x7
	model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
	model.add(BatchNormalization())
	model.add(LeakyReLU(alpha=0.2))
	# classifier
	model.add(Flatten())
	model.add(Dense(1, activation='sigmoid'))
	# compile model
	opt = Adam(lr=0.0002, beta_1=0.5)
	model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
	return model
```

Définition du discriminateur [1]

```python
# define the standalone generator model
def define_generator(latent_dim):
	# weight initialization
	init = RandomNormal(stddev=0.02)
	# define model
	model = Sequential()
	# foundation for 7x7 image
	n_nodes = 128 * 7 * 7
	model.add(Dense(n_nodes, kernel_initializer=init, input_dim=latent_dim))
	model.add(LeakyReLU(alpha=0.2))
	model.add(Reshape((7, 7, 128)))
	# upsample to 14x14
	model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
	model.add(BatchNormalization())
	model.add(LeakyReLU(alpha=0.2))
	# upsample to 28x28
	model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
	model.add(BatchNormalization())
	model.add(LeakyReLU(alpha=0.2))
	# output 28x28x1
	model.add(Conv2D(1, (7,7), activation='tanh', padding='same', kernel_initializer=init))
	return model
```

Définition du générateur [2]

```python
# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
	# make weights in the discriminator not trainable
	discriminator.trainable = False
	# connect them
	model = Sequential()
	# add generator
	model.add(generator)
	# add the discriminator
	model.add(discriminator)
	# compile model
	opt = Adam(lr=0.0002, beta_1=0.5)
	model.compile(loss='binary_crossentropy', optimizer=opt)
	return model
```

Combinaison des 2 [3]

```python
import mlxtend
import numpy as np

from mlxtend.data import loadlocal_mnist

trainX, trainy = loadlocal_mnist(
	images_path='/Users/JR/Desktop/Ubytes copie/train-images-idx3-ubyte',
	labels_path='/Users/JR/Desktop/Ubytes copie/train-labels-idx1-ubyte')

print(trainX.shape[0], trainX.shape[1])
trainX=np.reshape(trainX,(trainX.shape[0],28,-1))
print(trainX.shape[0], trainX.shape[1], trainX.shape[2])


# load mnist images
def load_real_samples():
	# load dataset
	# expand to 3d, e.g. add channels


	X = expand_dims(trainX, axis=-1)
	# select all of the examples for a given class
	selected_ix = trainy
	X = X[selected_ix]
	# convert from ints to floats
	X = X.astype('float32')
	# scale from [0,255] to [-1,1]
	X = (X - 127.5) / 127.5
	return X
```

Import de la base de données [4]

```python
# select real samples
def generate_real_samples(dataset, n_samples):
	# choose random instances
	ix = randint(0, dataset.shape[0], n_samples)
	# select images
	X = dataset[ix]
	# generate class labels
	y = ones((n_samples, 1))
	return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
	# generate points in the latent space
	x_input = randn(latent_dim * n_samples)
	# reshape into a batch of inputs for the network
	x_input = x_input.reshape(n_samples, latent_dim)
	return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
	# generate points in latent space
	x_input = generate_latent_points(latent_dim, n_samples)
	# predict outputs
	X = generator.predict(x_input)
	# create class labels
	y = zeros((n_samples, 1))
	return X, y
```

Génération de variables aléatoires [5]

```python
# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, latent_dim, n_samples=100):
	# prepare fake examples
	X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
	# scale from [-1,1] to [0,1]
	X = (X + 1) / 2.0
	# plot images
	for i in range(10 * 10):
		# define subplot
		pyplot.subplot(10, 10, 1 + i)
		# turn off axis
		pyplot.axis('off')
		# plot raw pixel data
		pyplot.imshow(X[i, :, :, 0], cmap='gray')
	# save plot to file
	pyplot.savefig('results_baseline/generated_plot_%03d.png' % (step+1))
	pyplot.close()
	# save the generator model
	g_model.save('results_baseline/model_%03d.h5' % (step+1))
```

Génération de résultats et sauvegardes [6]

```python
# create a line plot of loss for the gan and save to file
def plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist):
	# plot loss
	pyplot.subplot(2, 1, 1)
	pyplot.plot(d1_hist, label='d-real')
	pyplot.plot(d2_hist, label='d-fake')
	pyplot.plot(g_hist, label='gen')
	pyplot.legend()
	# plot discriminator accuracy
	pyplot.subplot(2, 1, 2)
	pyplot.plot(a1_hist, label='acc-real')
	pyplot.plot(a2_hist, label='acc-fake')
	pyplot.legend()
	# save plot to file
	pyplot.savefig('results_baseline/plot_line_plot_loss.png')
	pyplot.close()
```

Génération du graphique [7]

```python
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=10, n_batch=128):
	# calculate the number of batches per epoch
	bat_per_epo = int(dataset.shape[0] / n_batch)
	print(bat_per_epo)
	# calculate the total iterations based on batch and epoch
	n_steps = bat_per_epo * n_epochs
	# calculate the number of samples in half a batch
	half_batch = int(n_batch / 2)
	# prepare lists for storing stats each iteration
	d1_hist, d2_hist, g_hist, a1_hist, a2_hist = list(), list(), list(), list(), list()
	# manually enumerate epochs
	for i in range(n_steps):
		# get randomly selected 'real' samples
		X_real, y_real = generate_real_samples(dataset, half_batch)
		# update discriminator model weights
		d_loss1, d_acc1 = d_model.train_on_batch(X_real, y_real)
		# generate 'fake' examples
		X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
		# update discriminator model weights
		d_loss2, d_acc2 = d_model.train_on_batch(X_fake, y_fake)
		# prepare points in latent space as input for the generator
		X_gan = generate_latent_points(latent_dim, n_batch)
		# create inverted labels for the fake samples
		y_gan = ones((n_batch, 1))
		# update the generator via the discriminator's error
		g_loss = gan_model.train_on_batch(X_gan, y_gan)
		# summarize loss on this batch
		print('>%d, d1=%.3f, d2=%.3f g=%.3f, a1=%d, a2=%d' %
			(i+1, d_loss1, d_loss2, g_loss, int(100*d_acc1), int(100*d_acc2)))
		# record history
		d1_hist.append(d_loss1)
		d2_hist.append(d_loss2)
		g_hist.append(g_loss)
		a1_hist.append(d_acc1)
		a2_hist.append(d_acc2)
		# evaluate the model performance every 'epoch'
		if (i+1) % bat_per_epo == 0:
			summarize_performance(i, g_model, latent_dim)
	plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist)
```

Entraînement du générateur et du discriminateur [8]

Fig. 4.3.20 - Zoom sur la section [B] du schéma global.

Fig. 4.3.21 - Schéma de la section [B] du schéma global.

La section [4] quant à elle mérite un détour. Comme je l'évoquais plus tôt, les **ubytes** contenant les images de chiffres paramétriques doivent être chargées dans le modèle. Pour cela comme on peut le voir dans le détail ci-dessous, j'importe la fonction « loadlocal_mnist » du module **MLxtend** qui va, comme son nom le laisse à penser, permettre d'importer mes **ubytes** depuis mon disque dur. Elles sont ensuite converties au bon format attendu par la fonction suivante, qui va convertir les images en matrices de nombres, format compréhensible pour l'ordinateur.

```
import mlxtend
import numpy as np

from mlxtend.data import loadlocal_mnist

trainX, trainy = loadlocal_mnist(
        images_path='/Users/JR/Desktop/Ubytes copie/train-images-idx3-ubyte',
        labels_path='/Users/JR/Desktop/Ubytes copie/train-labels-idx1-ubyte')

print(trainX.shape[0], trainX.shape[1])
trainX=np.reshape(trainX,(trainX.shape[0],28,-1))
print(trainX.shape[0], trainX.shape[1], trainX.shape[2])


# load mnist images
def load_real_samples():
        # load dataset
        # expand to 3d, e.g. add channels


        X = expand_dims(trainX, axis=-1)
        # select all of the examples for a given class
        selected_ix = trainy
        X = X[selected_ix]
        # convert from ints to floats
        X = X.astype('float32')
        # scale from [0,255] to [-1,1]
        X = (X - 127.5) / 127.5
        return X
```

Fig. 4.3.22 - Zoom sur la section [4] : import de la base de données.

La section [5] va générer des valeurs aléatoires pour choisir des images sur lesquelles apprendre, qui vont être piochées dans l'**espace latent**, un espace multidimensionnel comprenant toutes les solutions hypothétiques (j'y reviendrai dans le détail dans l'analyse des résultats).

La section [6] va générer, ou du moins définir la génération de résultats périodiques ainsi qu'une sauvegarde régulière du modèle. Cela permet d'avoir un œil sur l'avancé de l'apprentissage tout en ayant des sauvegardes de « secours ». Ici, les aperçus de résultats ainsi que les sauvegardes se font toutes les 100 **itérations** (car « n_samples=100 »). La section suivante va mettre en place un graphique comprenant l'évolution de plusieurs valeurs, informant sur la qualité de l'apprentissage. Là encore cette partie sera vue en détail lors de la présentation des résultats.

Enfin, la section [8] va définir l'entraînement du **GAN** en lui-même, en reprenant la plupart des fonctions définies plus tôt. Celle-ci est d'une grande importance, non pas pour sa structure mais plutôt pour certaines valeurs que je vais choisir. En effet il existe plusieurs « leviers », qui vont influer sur la qualité des résultats mais surtout sur le temps de **computation**. Comme la base de données est personnelle, il faut adapter certaines valeurs pour s'assurer des résultats pertinents. Encore une fois de nombreux essais ont été nécessaires pour trouver un équilibre entre qualité, quantité et temps.

```
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=10, n_batch=128):
        # calculate the number of batches per epoch
        bat_per_epo = int(dataset.shape[0] / n_batch)
        print(bat_per_epo)
        # calculate the total iterations based on batch and epoch
        n_steps = bat_per_epo * n_epochs
        # calculate the number of samples in half a batch
        half_batch = int(n_batch / 2)
        # prepare lists for storing stats each iteration
        d1_hist, d2_hist, g_hist, a1_hist, a2_hist = list(), list(), list(), list(), list()
```

Fig. 4.3.23 - Fragment de la section [8], définition de l'entraînement.

Ci-dessus se trouve un extrait de cette section [8], le début de la définition de la fonction « train ». 3 variables constituent les leviers que je vais devoir contrôler :

- n_epoch=10
- n_batch=128
- bat_per_epo = int(dataset.shape[0] / n_batch)

Mais que signifient ces étrangetés ? Pour faire simple, le **GAN** va réaliser son apprentissage sur un certain nombre d'images de chiffres, plusieurs fois, et re-répéter toute l'opération. Pour mieux comprendre, rien de mieux qu'une comparaison.

Imaginons que les images de chiffres sont des mots : le **GAN** va apprendre sur tous les mots d'une page, puis recommencer sur chaque page du livre, et répéter l'opération sur autant de livres que je lui donne.
Ainsi en reprenant les valeurs de l'encadré précédent, il y a 128 mots par page (n_batch=128, **batch** signifiant lot en français), et 10 livres au total (n_epoch=10). Mais combien de pages par livre ? Cela est définit par la variable « *bat_per_epo*", dont la valeur équivaut au nombre total d'images de la base de données divisé par le nombre d'images par **batch**. Ramené à la comparaison, c'est comme si je prenais tous les mots de la langue française et que je faisais autant de pages que nécessaire pour qu'ils entrent tous dans un livre.

Pour résumer avec les données de l'expérience : la base de données contient 9000 images de chiffres manuscrits. Le **GAN** apprend sur 128 images par **itération** et ce 70 fois (car 9000 / 128 = 70). Il répètera cette opération 10 fois, pour un total de 700 **itérations** (70 * 10 = 700). J'ai donc 10 livres de 70 pages avec 128 mots par page.

Ce choix de variables m'a permis d'obtenir des résultats convenables pour une durée de calcul par l'ordinateur d'environ 5 heures. Cela peut sembler acceptable, mais lorsqu'il faut recommencer des dizaines de fois l'opération pour avoir les résultats souhaités, je me dis qu'un mémoire sur la migration des batraciens aurait été tout aussi bien.

[C]
Fig 4.3.18

Pour rappel, aucun « calcul » n'a encore été réalisé. Tout ce qui vient d'être abordé ne relève que de la préparation et la définition de fonctions, qui n'attendent qu'à être lancées. C'est le rôle de la section [C] du schéma global, présentée dans son entièreté en figure 4.3.24. Ces quelques lignes insignifiantes sont pourtant essentielles. On remarque d'ailleurs qu'il n'y a plus de définition de fonctions indiquées en couleur, cette étape étant terminée. La ligne capitale est ainsi la dernière, qui va lancer toute la phase de calculs avec la fonction « *train* » (définie dans la section [8] précédente).

```
# make folder for results
makedirs('results_baseline', exist_ok=True)
# size of the latent space
latent_dim = 50
# create the discriminator
discriminator = define_discriminator()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, discriminator)
# load image data
dataset = load_real_samples()
print(dataset.shape)
# train model
train(generator, discriminator, gan_model, dataset, latent_dim)
```

Fig. 4.3.24 - Zoom sur la section [C] du schéma global, lancement de l'entraînement.

Tout se lance donc, et il est possible d'observer en temps réel (voir ci-dessous), à chaque **itération** donc environ toutes les 2 minutes, l'évolution de plusieurs valeurs. Celles-ci prendront tout leur intérêt durant l'étude des résultats, au sein de graphiques. Pour le moment, inutile de s'attarder sur ces "*d1*", "*d2*" ou "*g*". Tant que toutes les 2 minutes une nouvelle ligne s'ajoute et que les valeurs semblent différentes de la précédente, tout va bien, le calcul se déroule probablement correctement.

Lancement de l'entraînement

```
>1, d1=0.828, d2=0.705 g=1.036, a1=46, a2=51
>2, d1=0.178, d2=1.514 g=0.641, a1=100, a2=10
>3, d1=0.107, d2=1.393 g=0.773, a1=100, a2=7
>4, d1=0.077, d2=1.159 g=0.896, a1=100, a2=23
>5, d1=0.069, d2=1.023 g=1.191, a1=100, a2=28
>6, d1=0.068, d2=0.864 g=1.198, a1=100, a2=42
>7, d1=0.072, d2=0.861 g=1.260, a1=100, a2=42
>8, d1=0.083, d2=0.885 g=1.304, a1=100, a2=35
>9, d1=0.105, d2=0.837 g=1.463, a1=100, a2=32
>10, d1=0.127, d2=0.698 g=1.442, a1=100, a2=51
>11, d1=0.133, d2=0.798 g=1.284, a1=100, a2=40
>12, d1=0.138, d2=0.630 g=1.510, a1=100, a2=60
>13, d1=0.144, d2=0.516 g=1.707, a1=100, a2=78
>14, d1=0.125, d2=0.430 g=1.752, a1=100, a2=84
>15, d1=0.101, d2=0.383 g=1.712, a1=100, a2=90
>16, d1=0.086, d2=0.529 g=1.396, a1=100, a2=73
>17, d1=0.093, d2=0.540 g=1.522, a1=100, a2=76
>18, d1=0.103, d2=0.305 g=1.574, a1=100, a2=98
>19, d1=0.098, d2=0.316 g=1.747, a1=100, a2=95
>20, d1=0.088, d2=0.267 g=1.683, a1=100, a2=98
>21, d1=0.095, d2=0.271 g=1.394, a1=100, a2=96
>22, d1=0.097, d2=0.355 g=0.963, a1=100, a2=92
>23, d1=0.114, d2=0.366 g=0.951, a1=100, a2=96
>24, d1=0.184, d2=0.377 g=1.643, a1=100, a2=92
>25, d1=0.265, d2=0.182 g=1.940, a1=100, a2=100
>26, d1=0.216, d2=0.124 g=1.532, a1=100, a2=100
>27, d1=0.128, d2=0.081 g=1.151, a1=100, a2=100
```

[...] Ellipse

```
>696, d1=0.028, d2=0.096 g=5.213, a1=100, a2=95
>697, d1=0.038, d2=0.012 g=4.909, a1=100, a2=100
>698, d1=0.084, d2=0.069 g=4.123, a1=100, a2=98
>699, d1=0.026, d2=0.055 g=4.885, a1=100, a2=100
>700, d1=0.079, d2=0.061 g=4.251, a1=100, a2=98
```

Fin de l'entraînement

Fig. 4.3.25 - Aperçu des statistiques de chaque **itération** en direct.

*4.3.d Exploitation des résultats*

Séparer la génération des résultats du réseau de neurones n'est pas qu'une nécessité de discrimination en chapitres, mais principalement car un **GAN** fonctionne en 2 temps. D'abord l'entraînement, qui est plutôt long et qui vient de se terminer à la page précédente, puis la génération de données. L'entraînement une fois achevé a permis de créer un fichier au format .h5, regroupant tout le réseau de neurones et les poids résultants de l'apprentissage. Ce fichier est relativement léger, car il ne contient au final aucune image : il s'agit de la « recette » pour créer des chiffres. C'est l'heure de fabriquer le "gâteau".

[D]
Fig 4.13

Ci-contre se trouve donc le détail de la dernière partie de l'**algorithme**, la section [D] du schéma global, où je vais m'occuper de ce fameux fichier .h5. Comme c'est une partie pouvant s'exécuter sans avoir nécessairement lancé les sections précédentes du code, il est préférable d'importer de nouveau certains modules ([1]).

Ensuite en section [2] je génère des points aléatoires dans l'**espace latent**. Pour rappel, l'**espace latent** est un espace multidimensionnel comportant toutes les solutions existantes. La section [2] va ainsi générer aléatoirement les coordonnées, pour ensuite en déduire des points de l'**espace latent** et ainsi des images de chiffres.
La section [3] s'occupe d'afficher des images de chiffres fraîchement créées et les sauvegarder sur l'ordinateur.

Encore une fois les actions que je viens de citer ne sont pas réalisées, mais simplement définies. C'est donc le rôle de la section [4] de lancer les différentes opérations les unes après les autres : je charge le fichier .h5 comprenant le **générateur**, je génère les points dans l'**espace latent**, je génère les images et j'affiche tout cela.
A titre d'exemple dans le code ci-contre l'**espace latent** est en 50 dimensions (je sais que 2 dimensions c'est le domaine du plat, 3 dimensions la perspective, mais 50 dimensions ne sont pas concevables pour l'esprit), et je génère 100 images de chiffres.

```python
from keras.models import load_model
from numpy.random import randn
from matplotlib import pyplot
import matplotlib.pyplot as plt

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
        # generate points in the latent space
        x_input = randn(latent_dim * n_samples)
        # reshape into a batch of inputs for the network
        x_input = x_input.reshape(n_samples, latent_dim)
        return x_input

# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, n):
        # plot images
        for i in range(n * n):
                # define subplot
                pyplot.subplot(n, n, 1 + i)
                # turn off axis
                pyplot.axis('off')
                # plot raw pixel data
                pyplot.imshow(examples[i, :, :, 0], cmap='gray')
                pyplot.savefig('plot_300dpi.png', dpi=300)
        pyplot.show()

# load model
model = load_model('results_baseline/model_700.h5')
# generate images
latent_points = generate_latent_points(50, 100)
# generate images
X = model.predict(latent_points)
# plot the result
save_plot(X, 10)
```

Fig. 4.3.26 - Zoom sur la section [C] du schéma global, génération des images.

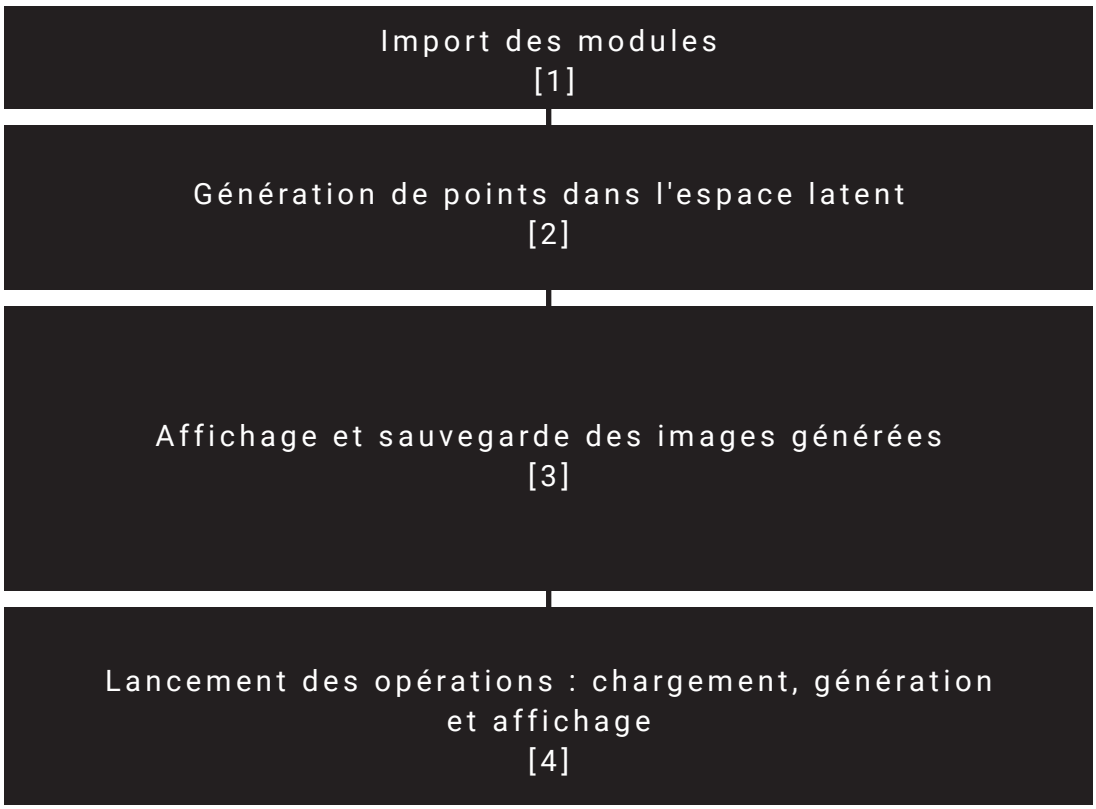



Fig. 4.3.27 - Schématisation de la figure précédente.

Mais que serait cette partie « résultats » sans une visualisation d'images ? Voici donc ci-contre les 100 images générées dont je parlais précédemment. Il s'agit ici de la 70ème **itération**, les résultats sont donc assez vagues et pleins de « grain ». Sur la page de droite se trouvent ainsi les images de deux essais différents, l'un en blanc sur noir et l'autre en noir sur blanc. En effet il s'agit d'un réseau de neurones, j'obtiens des résultats différents à chaque essai, parfois meilleurs, parfois moins bons. Cela humanise un peu ce **GAN**, et je me réjouis de ses réussites. Je commence ainsi à voir quelques arrondis apparaître, parfois l'ombre d'un 8.

Ci-dessous se trouve l'évolution de plusieurs statistiques au fil de l'apprentissage. La courbe verte « *gen* » est la qualité du **générateur**. Plus elle a une valeur élevée, plus les résultats sont réalistes. Les courbes bleues et oranges « *d-real* » et « *d-fake* » sont quant à elle l'erreur du **discriminateur**, qui doit être minimisée (proche de 0). Ce sont les mêmes informations que l'on trouve sur le graphique du dessous, mais cette fois on inverse la donnée : « *acc-real* » et « *acc-fake* » doivent être proche de 1. La lecture est cependant meilleure car l'échelle du graphique est plus petite.

Je constate donc qu'à 70 **itérations** (sur l'axe horizontal dit des abscisses), le **générateur** n'est pas encore arrivé à son maximum, les résultats sont donc visuellement mitigés.

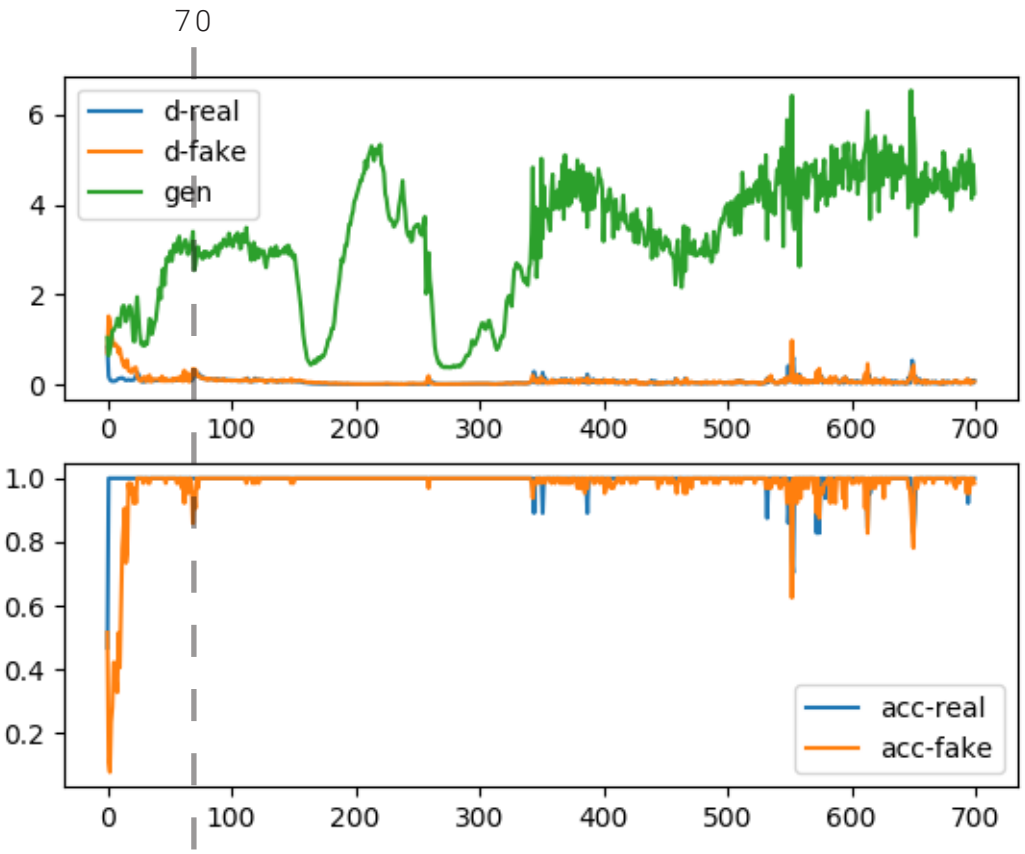



Fig. 4.3.28 - Statistiques de l'essai A.

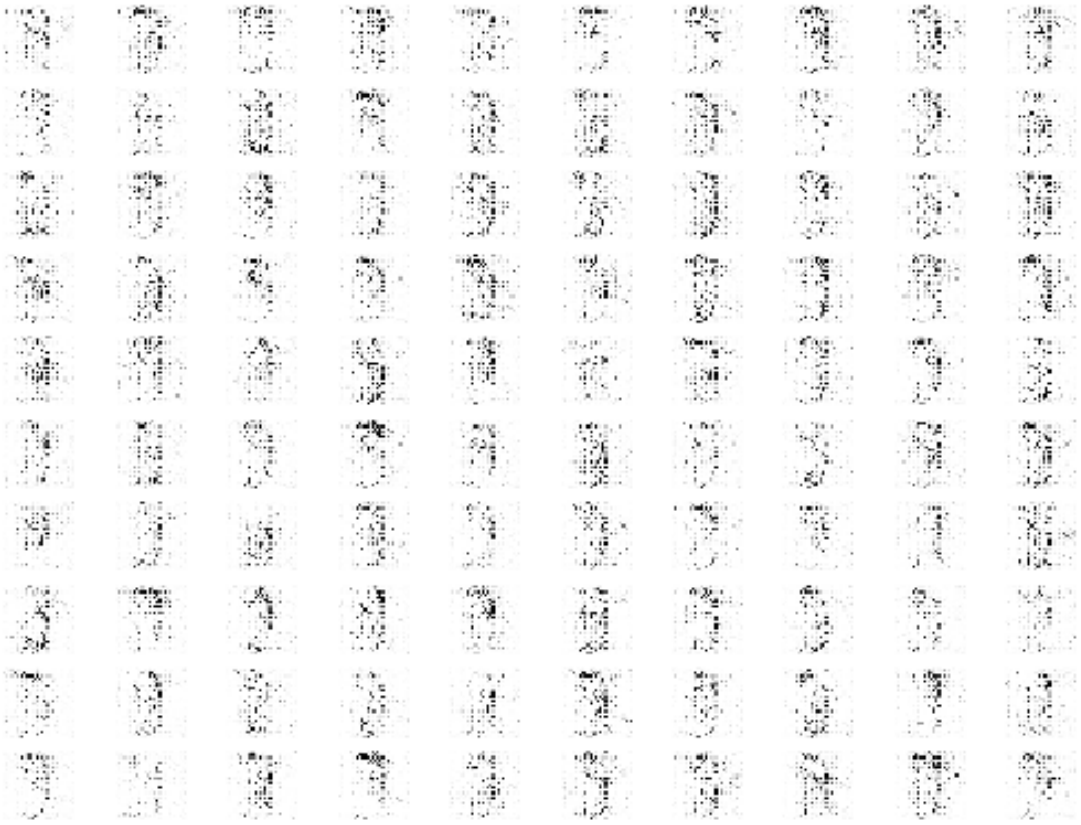


Fig. 4.3.29 - Essai A, noir sur blanc, 70ème **itération**.

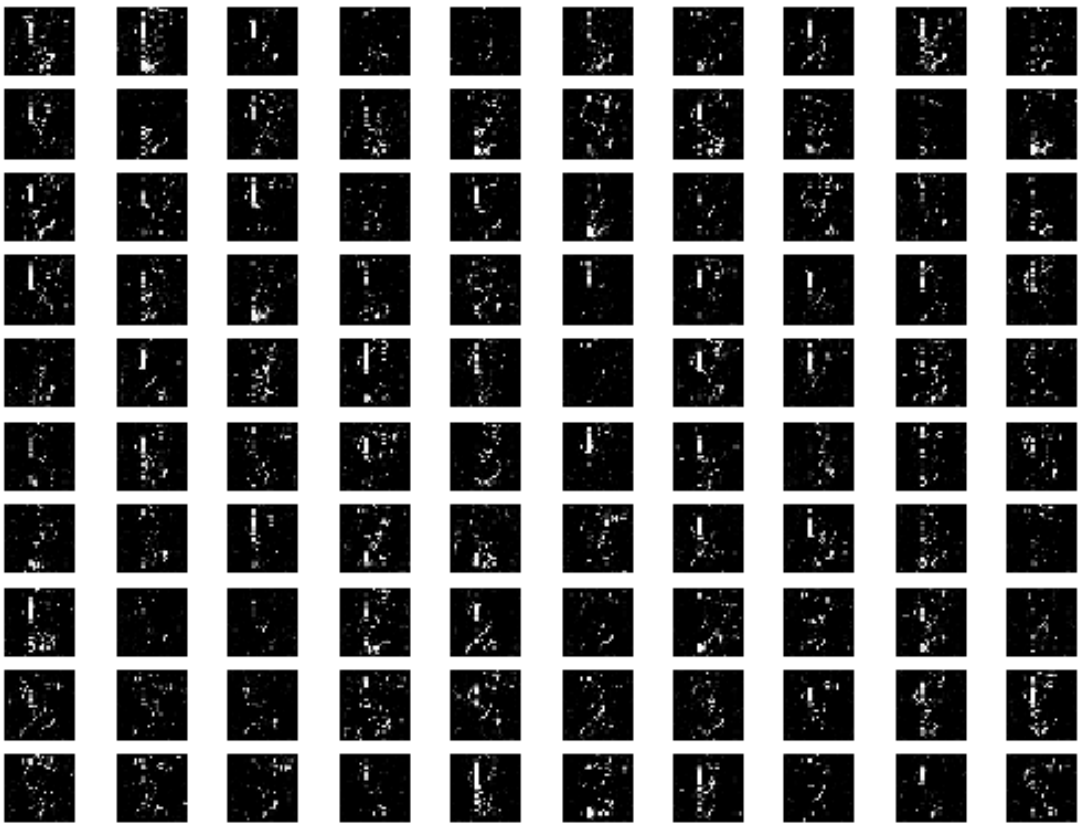


Fig. 4.3.30 - Essai B, blanc sur noir, 70ème **itération**.

210 **itérations** plus tard, misère ! Les résultats de l'essai A en noir sur blanc sont très mauvais, pire que ceux obtenus précédemment. Cela se confirme sur la courbe verte du **générateur** : il s'agit d'un creux. Ce sont les aléas de l'apprentissage, on a tous nos mauvais moments.
L'essai B en revanche semble se porter mieux, et je commence clairement à distinguer des formes de chiffres. Cela se confirme sur la courbe du **générateur** en bas de la page, ou la 280ème **itération** se trouve en sortie de creux. Bien que les deux essais soient différents, les courbes sont tout de même assez similaires dans leur aspect, ce qui témoigne d'une régularité du processus d'apprentissage d'un essai à l'autre.

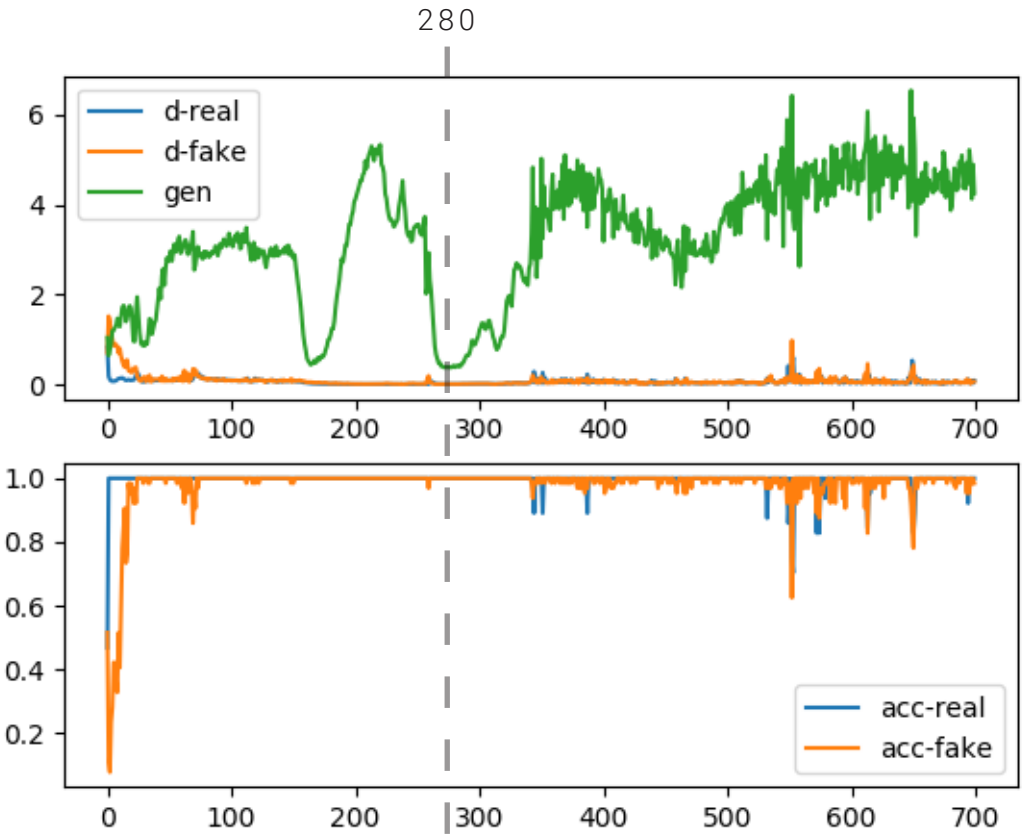



Fig. 4.3.31 - Statistiques de l'essai A.

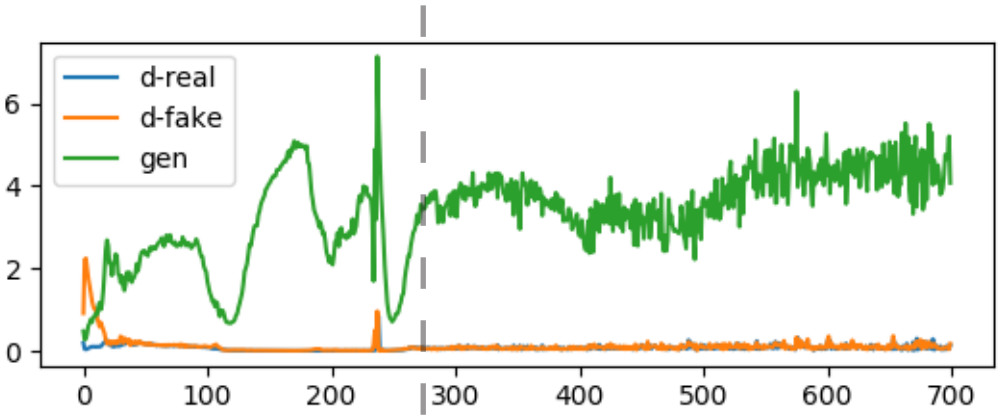



Fig. 4.3.32 - Statistiques de l'essai B (graphique 1/2).

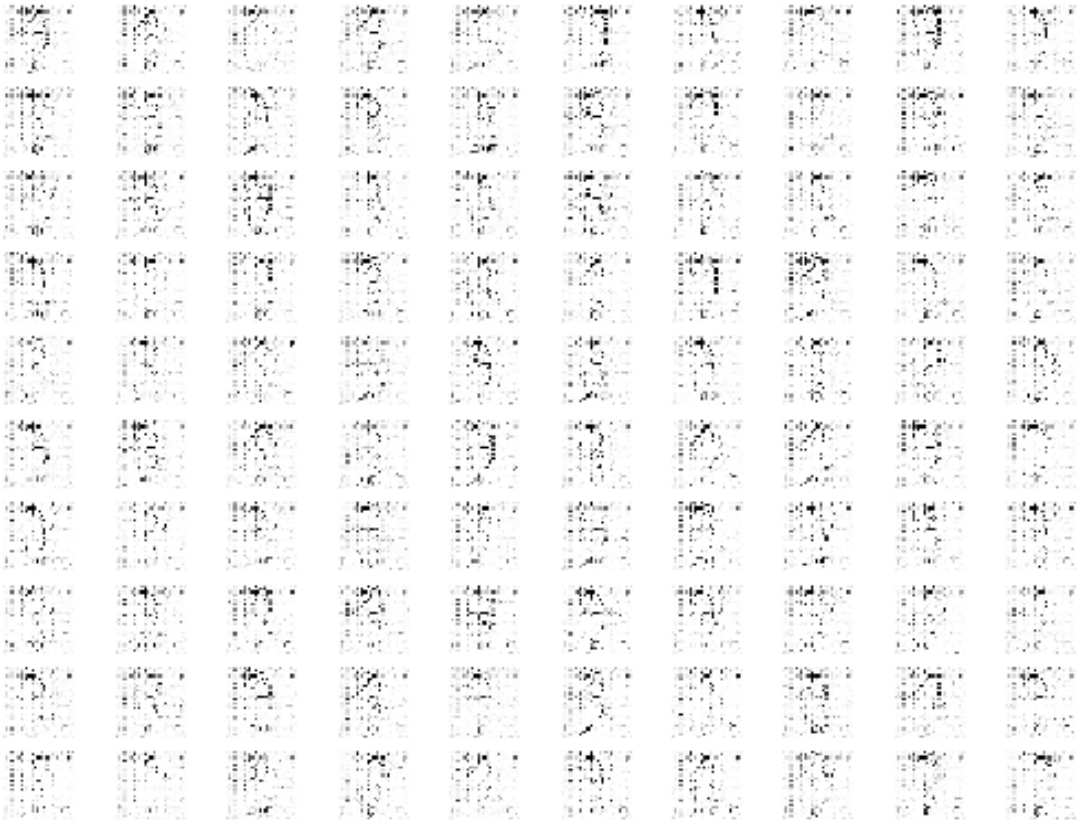


Fig. 4.3.33 - Essai A, noir sur blanc, 280ème **itération**.

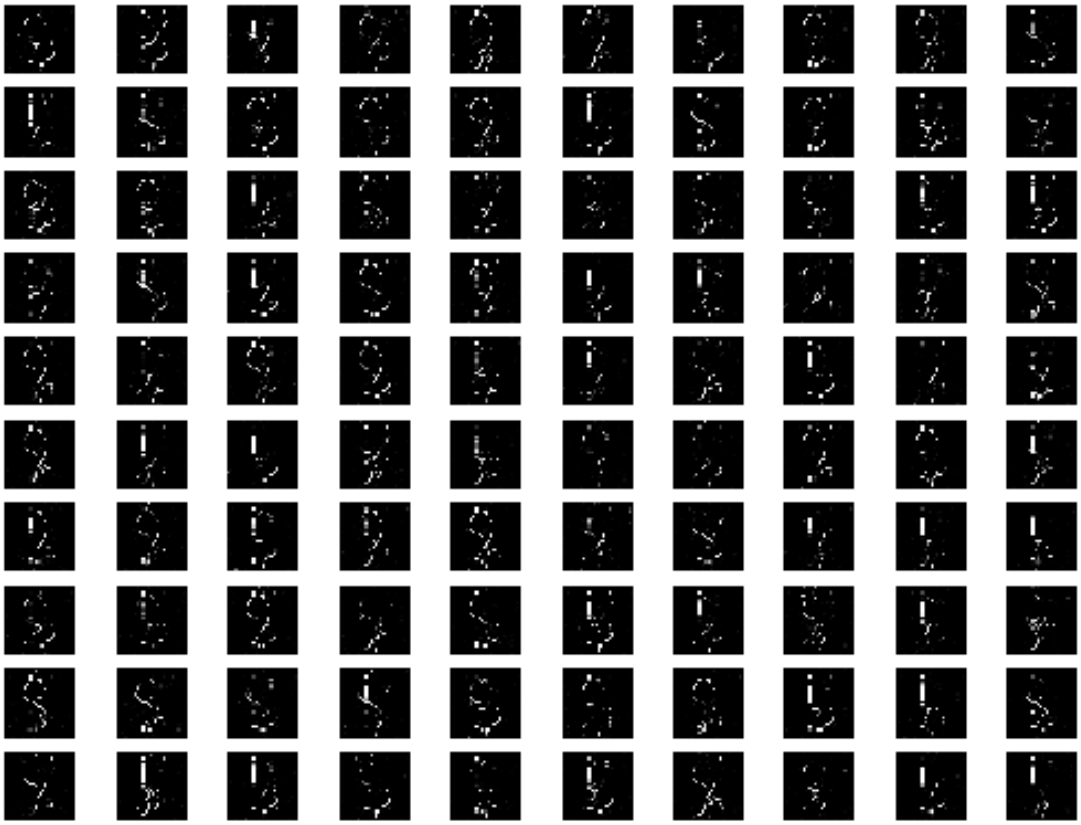


Fig. 4.3.34 - Essai B, blanc sur noir, 280ème **itération**.

Il est possible maintenant me focaliser un peu plus sur l'essai A (plus de résultats disponibles en annexes). A la 490ème **itération**, les résultats sont visuellement bien meilleurs que lors du creux (je reconnais des 8 et des 9), mais également bien meilleurs que lors de la 70ème **itération**, alors que la courbe verte était au même niveau (environ 2,5). En effet le **générateur** est en combat perpétuel avec le **discriminateur** : même s'il travail le de mieux en mieux, le **discriminateur** lui aussi est de plus en plus fort et va donc juger plus sévèrement. Ainsi les résultats sont meilleurs malgré une courbe identique, 420 **itérations** plus tard.

Enfin à la 630ème **itération**, j'obtiens les résultats les plus réalistes, avec l'apparition de 3 de 6 et de 7 assez nets. La courbe du **générateur** va se stabiliser, et les résultats resteront de cet ordre de réalisme, il semble que les limites du réseau de neurones sont atteintes.

Jusqu'ici je parle de "meilleurs" résultats dans le sens de leur réalisme, soit leur proximité avec la base de données paramétrique. En effet je veux commencer par prouver que le processus expérimental fonctionne, qu'il est capable de générer des données proches des données initiales. Or la problématique de ce mémoire est l'enrichissement, la création de nouvelles solutions riches et créatives. Inspectons donc les résultats de cette 630ème **itération** selon cette optique.

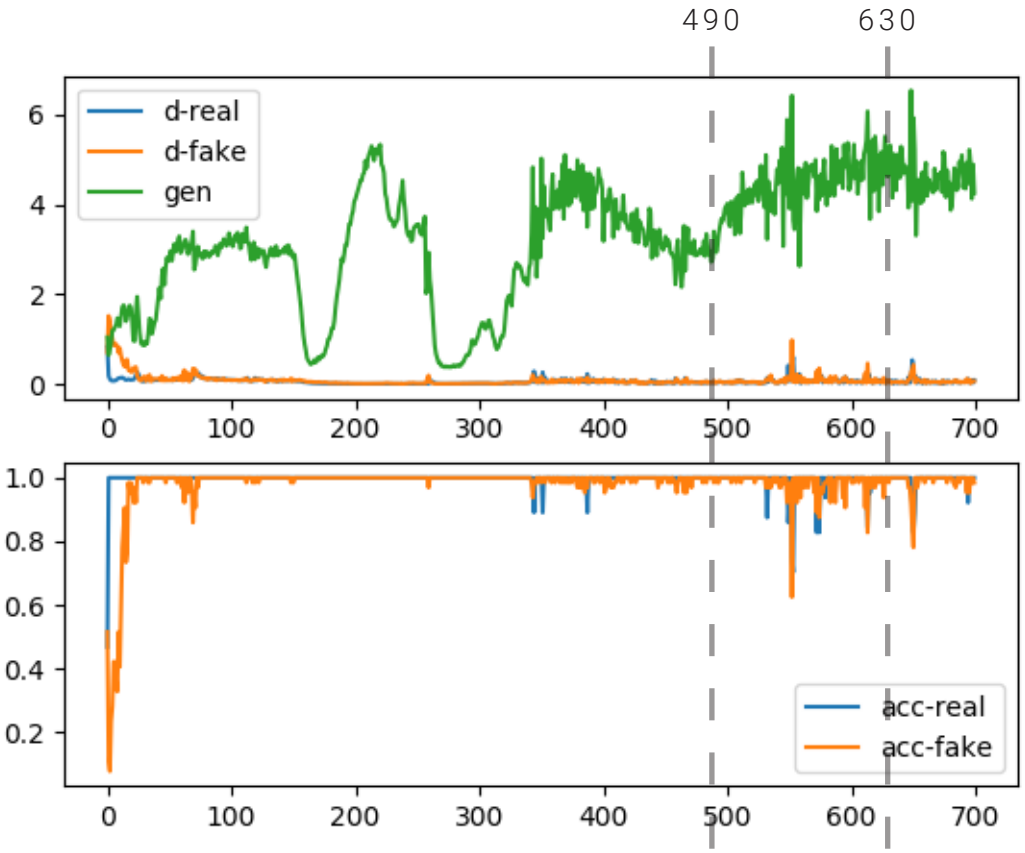



Fig. 4.3.35 - Statistiques de l'essai A.

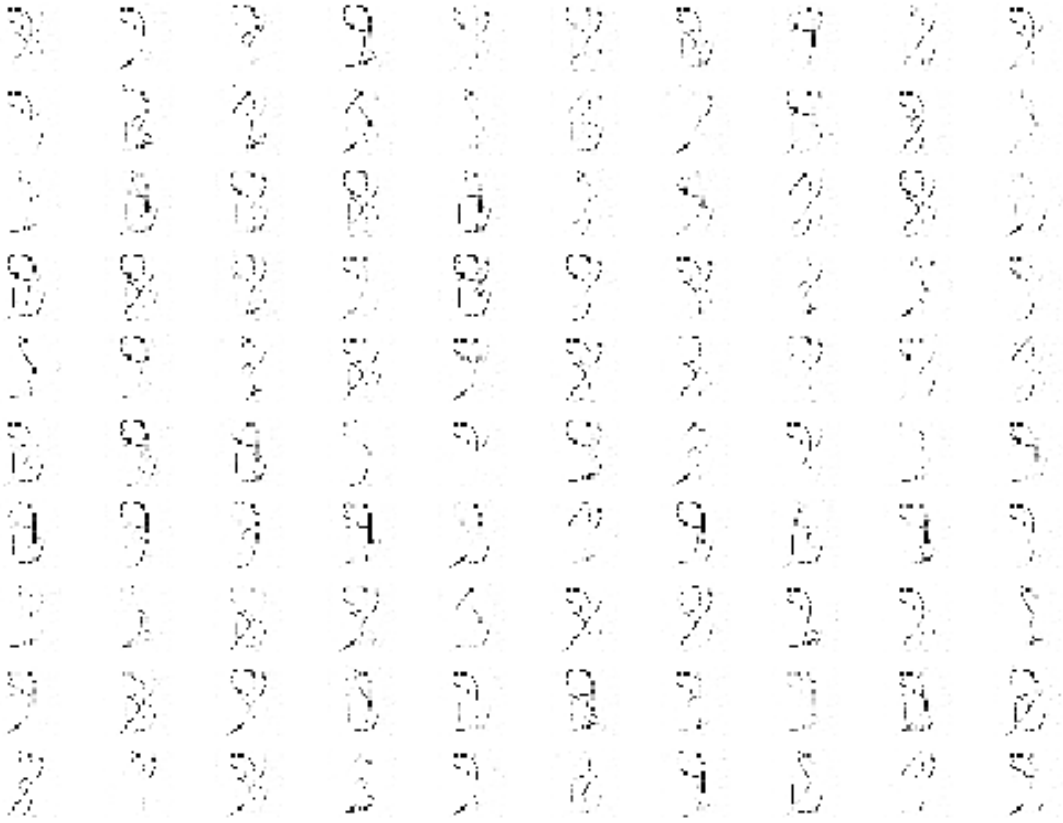


Fig. 4.3.36 - Essai A, noir sur blanc, 490ème **itération**.

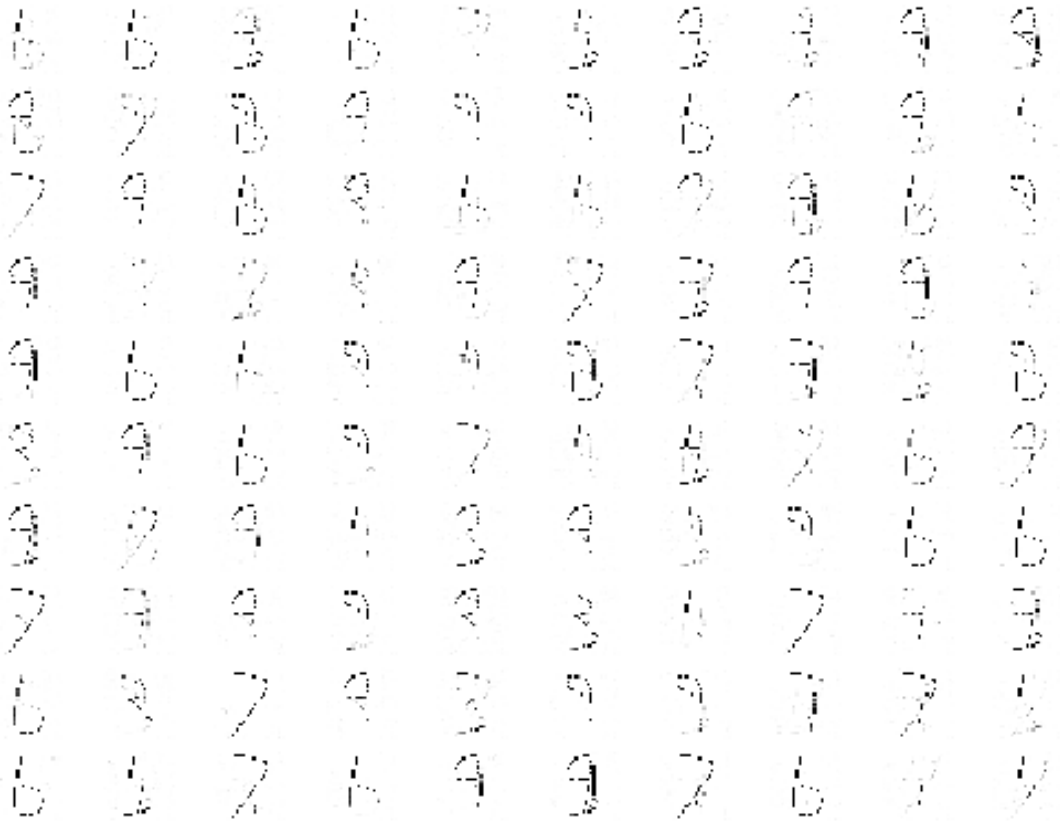


Fig. 4.3.37 - Essai A, noir sur noir, 630ème **itération**.

Rappelons les critères à retrouver dans les images générées, caractérisant un enrichissement notable :

- **1.** Une image fidèle aux images d'apprentissage : cela montre que le l'**algorithme** a correctement appris, et qu'il sait a minima imiter à l'identique,

- **2.** Une image « chimérique » : un mélange entre plusieurs images, premier degré d'enrichissement recherché,

- **3.** Des étrangetés sans nom qui, second degré d'enrichissement, observe un potentiel créatif recherché également.

Il faut également préciser que les sélections d'images qui vont suivre sont très subjectives. Elles sont le fruit de ma propre analyse des formes et de ce que j'ai pu y reconnaître. Ainsi il se peut que là où je reconnais une forme il n'en soit pas de même pour quelqu'un d'autre, et inversement.

Afin de rester rigoureux, je base ma stratégie de sélection sur une série de réflexions simples, fortement basée sur mes réactions à chaud redécouvrant les résultats. Si l'image m'évoque immédiatement un chiffre que je connais, c'est une image fidèle. Par contre, si j'hésite entre un chiffre ou un autre, voir entre plus de deux chiffres, je considère le résultat comme étant une chimère. Enfin, si les pixels me permettent de distinguer une forme, mais qu'elle ne me fait penser à aucun chiffre au premier regard, je peux la classer comment étant une étrangeté.

Ce système me permet donc d'identifier et classer les résultats obtenus, et valider ou non les 3 critères de la grille. Je pourrais envisager une 4ème catégorie, incluant les « chiffres ratés », soit les images intermédiaires entre chiffres fidèles et étrangetés, des chiffres que l'**algorithme** n'aurait pas réussi à aboutir suffisamment. Je décide cependant de me concentrer sur les extrêmes (fidèle, chimère, étrangeté) et ne pas considérer les intermédiaires pour ne pas s'y perdre.

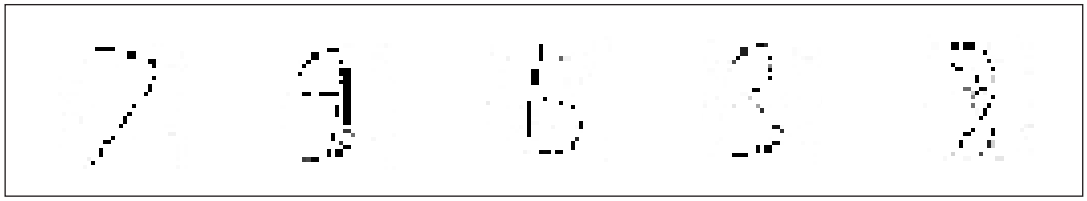


Fig. 4.3.38 - Selection de chiffres fidèles.

Le premier critère est assez simple à confirmer, je retrouve bien en figure 4.3.38 des chiffres tels que je pouvais les trouver dans la base de données paramétrique, avec cette esthétique manuscrite particulière, la simplicité du 7, le 8 pas toujours droit ou le 3 tortueux.

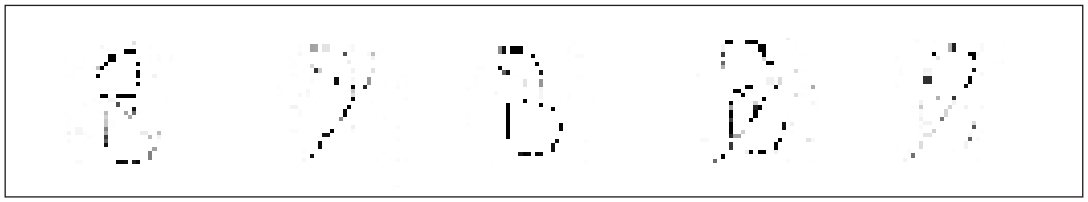


Fig. 4.3.39 - Selection de chimères.

Les chimères sont plus difficiles à identifier, mais j'en trouve quelques-unes. Dans l'ordre : un mélange 9 et 8, un 7 avec la boucle du 9, un 8/6, un trio étrange entre 6,7 et 9 et enfin un 8/7.

Fig. 4.3.40 - Selection d'étrangetés.

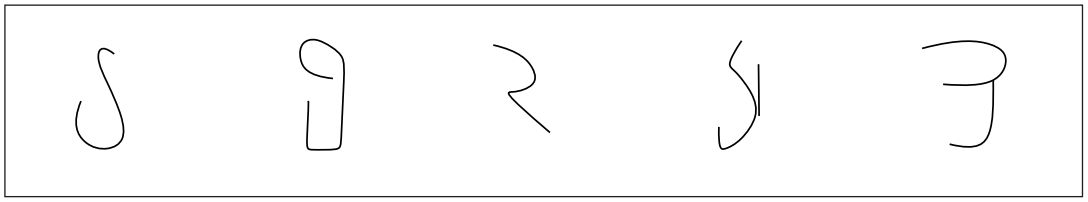


Fig. 4.3.40 - Retraçage des "étrangetés".

Enfin, certaines solutions ne relèvent ni de la chimère ni du chiffre, et valident le critère « bizarreries ». Mais que signifient-elles vraiment ? Elles ne sont pas le fruit du hasard, ni de tracés aléatoires créant des forment. Ces solutions ont été créées dans la même logique que les chiffres de 0 à 9, sauf qu'elles ne ressemblent pas à ces 10 possibilités fixées par l'Ecriture et son évolution. Ces images auraient pu être des symboles utilisés par tous, elles en ont le potentiel car elles suivent le même principe de tracé que les 10 élus que l'on connait.

En figure 4.3.41, je me suis amusé à retracer ces prototypes de chiffres pour voir ce que cela donne. Le résultat est tout aussi intéressant que le procédé d'interprétation. J'imagine facilement ces symboles apparaître dans une langue étrangère ou encore fantaisiste.

Créer de nouveaux chiffres n'est malheureusement pas vraiment une des préoccupations de la société, ni une des miennes. Néanmoins, ce potentiel créatif peut présenter un réel apport en architecture. C'est comme si ce processus expérimental avait proposé de nouvelles typologies d'habitations jamais vues jusqu'alors, en s'inspirant des typologies existantes. Cette hypothèse est bien ambitieuse, surtout que je ne sais toujours pas si le processus en a les capacités. Les prochaines parties de l'expérience qui vont suivre participeront ou non (suspens), à renforcer cette hypothèse.

Tout ce travail de création et de tâtonnement pour trouver les paramètres optimum, tant dans le réseau de neurones que dans le modèle paramétrique est heureusement terminé. En effet, la réussite de la première partie de l'expérience sur les chiffres permet de valider le processus tel qu'il a été exécuté. Je peux maintenant tester d'autres données en entrée (les formes, les dispositions…), qui pourront être mises en comparaison les unes aux autres car le processus reste fixe.

Pour en terminer avec cette première première expérience, je peux compléter le tableau récapitulatif, qui me servira au terme des 3 expériences à déterminer si oui ou non, l'expérimentation valide l'hypothèse.

| Critère / Expérience | Fidélité | Chimères | Etrangetés |
|---|---|---|---|
| Partie 1 "chiffres" | ✓ | ✓ | ✓ |
| Partie 2 "formes" | | | |
| Partie 3 "dispositions" | | | |

Tab. 4.3.1 - Tableau bilan mis à jour 1.

## 4.4 Partie 2 : les formes

Le processus étant validé par l'expérience témoin précédente, je peux maintenant réutiliser en grande partie les éléments mis en place et faire varier un critère : le type de donnée et sa stratégie de conception. Cette seconde partie de l'expérimentation sera donc beaucoup plus rapide, tant dans sa réalisation que dans son explication.

### *4.4.a Mise en place de la stratégie*

L'objectif de cette seconde expérience est de tester un nouveau critère, la forme, afin de valider un peu plus l'hypothèse selon laquelle ce processus expérimental fournit bien un enrichissement. Mais premièrement pourquoi la « forme », et qu'est-ce qui se cache derrière ce terme assez vague ?

Là où les chiffres étaient des symboles assez stéréotypés, et laissant peu de marge aux innovations, les formes, et plus particulièrement les formes géométriques en 2 dimensions (2D) (le carré, le rond, le triangle, etc.) observent moins de frontières. Il en existe une infinité, même si j'essaye de les ranger dans des cases pour savoir en parler. De plus, ces formes en 2D vont permettre de se rapprocher un peu plus encore des problématiques architecturales. En effet, même si cela reste simpliste pour le moment, ces formes peuvent s'apparenter à l'empreinte d'un bâtiment.

Certes il aurait été plus alléchant pour un architecte d'étudier non pas des formes géométriques, mais directement des empreintes de différentes typologies de bâtiment, mais force est de constater que la plupart de ces typologies ont des empreintes similaires proches du rectangle. Tout comme il peut être difficile de comprendre le fonctionnement d'une typologie rien que par son empreinte, le réseau de neurones aurait tout autant de mal à l'apprendre. Beaucoup de données seraient alors nécessaires (principes de surfaces, d'éclairement, nombre de pièces…) et en plus de s'empêtrer dans la complexité, je sortirais de l'objectif de ce mémoire. L'idée est ici de savoir si oui ou non, le processus mis en place sait apprendre des formes différentes.

Plus précisément, là où les chiffres étaient visuellement complexes mais dessinés avec la même logique (même nombre de points de passage), les formes seront ici plus simples graphiquement, mais toutes avec une logique différente (4 côtés pour le carré, 3 pour le triangle, le rond un seul mais qui se joint à lui même).

Comme pour la première expérience, pour savoir comment aborder l'apprentissage d'un réseau de neurones je peux m'inspirer de l'apprentissage de nos chers petits humains. Sur la figure ci-contre, (que je ne peux pas réellement référencer car tous les sites de ventes d'images de stock se revendiquent propriétaires), je peux observer les premières formes qui sont enseignées aux enfants. Je remarque qu'à l'inverse de l'apprentissage des chiffres (voir figure 4.3.3), il n'y a pas d'ordre spécifique de tracé, il faut juste reproduire l'image. Aussi dans cette expérience je n'essayerai pas de reproduire l'aspect manuscrit des formes, car contrairement aux chiffres les formes doivent être le plus fidèles possibles à leurs modèles.

Je peux ainsi, en plus des formes géométriques ci-contre, ajouter quelques formes à générer paramétriquement, un peu moins communes. Par exemple un polygone irrégulier, donc les côtés ne sont pas tous égaux (figure 4.4.1). Pour tester les limites du système je peux même concervoir des "**blobs**" (figure 4.4.2), plus difficiles à définir et relevant plus de la tache que de la forme géométrique.

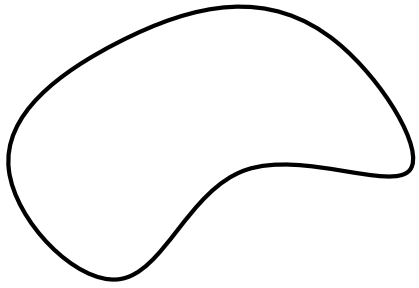


Fig. 4.4.1 - "**Blob**".

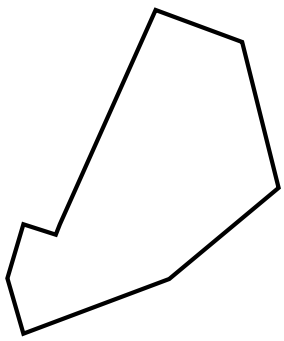


Fig. 4.4.2 - Polygone irrégulier.

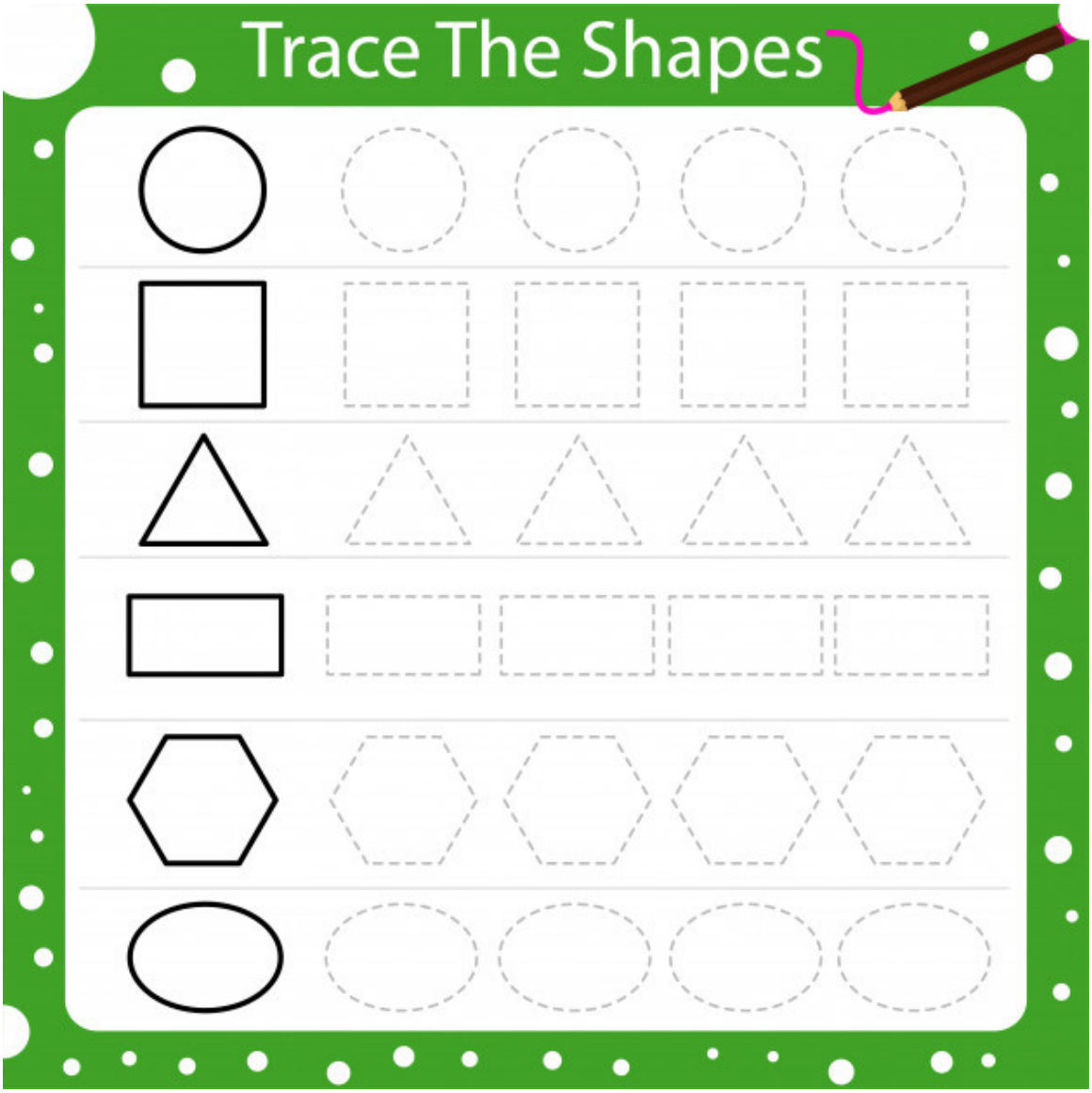



Fig. 4.4.3 - Guide pour enfant pour apprendre à tracer des formes (source : image de stock).

*4.4.b Le modèle paramétrique*

Etant donné que la stratégie est différente, le modèle Grasshopper est à modifier en conséquence. Heureusement, il est possible d'en conserver la logique, ainsi que certaines parties. La section [A] générant les variables est légèrement simplifiée, car il n'y a pas besoin d'autant de variables que lors de l'expérience précédente. La section [D] qui capture chaque image reste intacte, tout comme la section [C] qui affiche les résultats.

Par conséquent, c'est la section [B] générant les formes qui est principalement modifiée. Là où il fallait générer 10 chiffres, il faut maintenant générer 10 formes différentes, d'où les 10 blocs gris centraux, chacun générant une forme. On remarque d'ailleurs que beaucoup de ces blocs gris sont de dimensions différentes, alors que les blocs générant les chiffres étaient tous identiques. C'est une des premières conséquences de la stratégie mis en place ci-avant. En effet chaque forme se dessine avec une logique est une approche différente, ce qui résulte de programmations visuelles différentes également.

Je vais donc m'attarder un peu sur cette section [B], afin de comprendre comment se dessinent quelques formes. La première expérience était lourde en explications, mais je peux enfin profiter d'un processus qui fonctionne pour aller beaucoup plus vite.

Fig. 4.4.4 - Schéma global du modèle paramétrique.

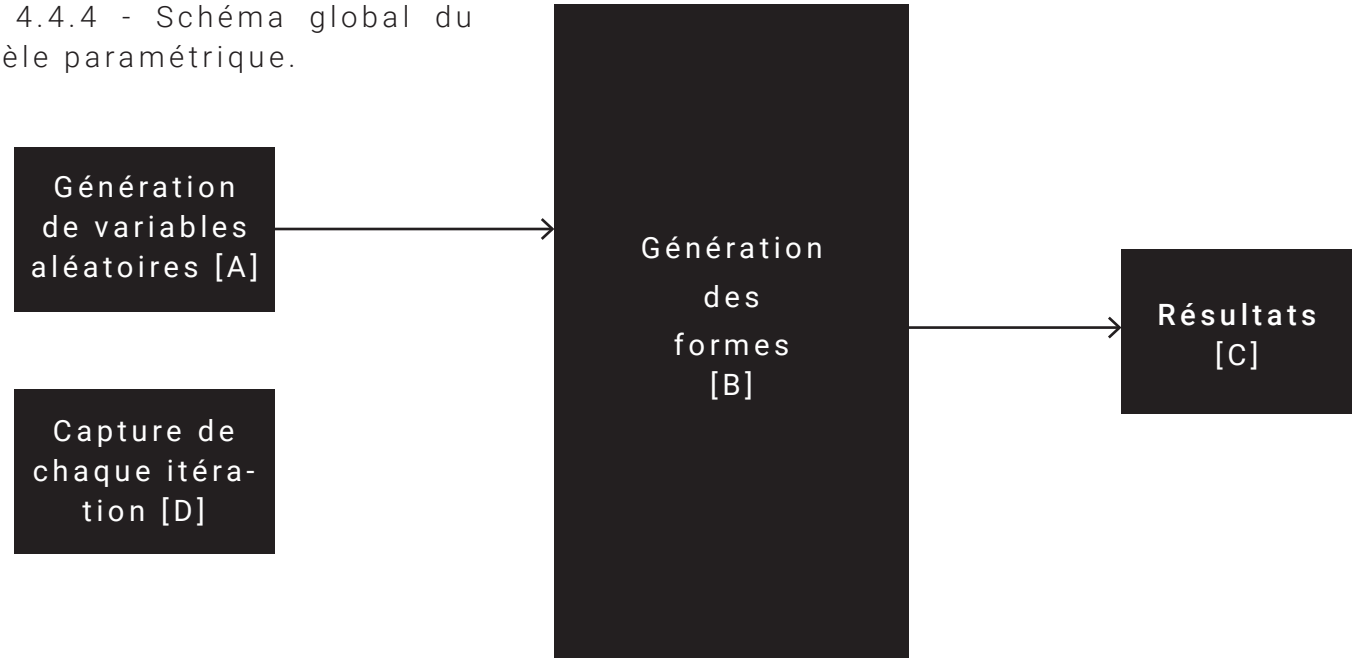
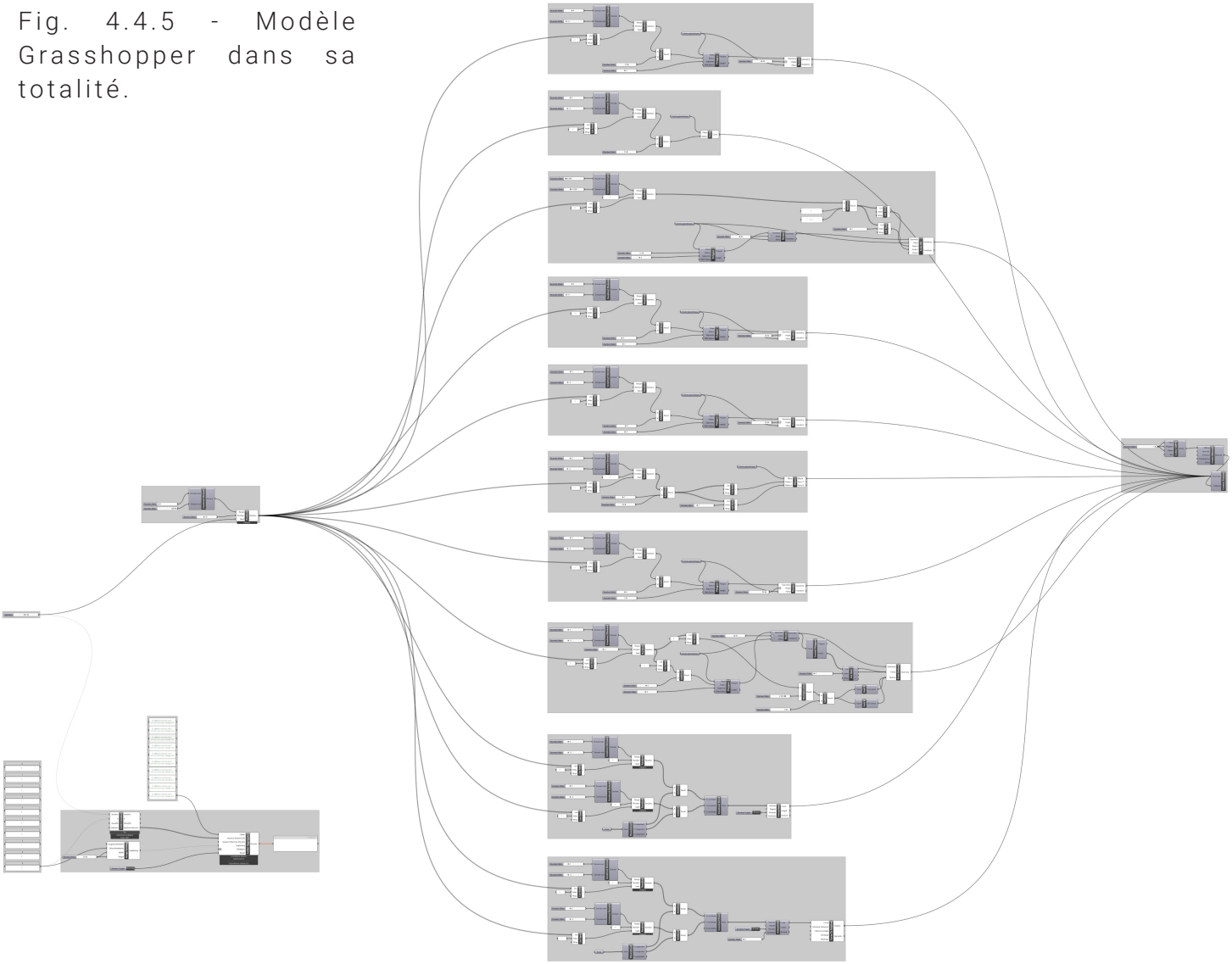



Fig. 4.4.5 - Modèle Grasshopper dans sa totalité.



La section [B] du schéma global est donc composée de 10 blocs, et je vais rapidement en aborder 3 principaux. Le premier est consacré à une des formes les plus simples : le carré. Pour cela rien de plus simple, j'utilise un composant « *polygon* » dans la section [1] du schéma en figure 4.4.7. J'indique 4 côtés à ce composant, qui va ainsi produire un carré. Un second composant est nécessaire pour pivoter la géométrie et obtenir un beau carré comme on le connait et non un losange. Cependant, je souhaite faire varier les dimensions de ce polygone afin de provoquer l'aléatoire dû aux aléas du tracé à la main ou à l'ordinateur. Pour cela, grâce à la section [3] je génère une valeur aléatoire (entre 4 et -4 ici) qui va venir s'additionner (ou se soustraire) aux dimensions existantes du carré (grâce à la section [2] du même schéma). Cette section « *génération des valeurs aléatoires* » est identique à celle qui générait l'aléatoire pour les chiffres, avec le même principe de nouvelle **graine** à chaque **itération**.

A partir de ce bout de modèle paramétrique, il est possible de l'adapter très facilement pour obtenir des variations du carré. Par exemple, j'en parlait au paragraphe précédent, en supprimant le composant de rotation j'arrive à obtenir un losange. En jouant sur le nombre de côtés j'aboutis à un triangle ou même un pentagone, et en variant les dimensions des côtés opposés se forme un rectangle. Enfin, en poussant certains angles j'arrive à créer un parallélogramme.

Toutes ces modifications sont mineures, et ne nécessitent pas de les détailler plus que cela. En revanche, la figure 4.4.8 ci-contre génère la création d'un cercle, relativement simple également mais j'ai cette fois recours au composant « *circle* » sans grande surprise. Là encore je fais varier son rayon aléatoirement à chaque **itération** grâce à la même section [3]. Je peux aisément adapter la création du cercle à une ellipse, en jouant cette fois sur l'échelle horizontale.

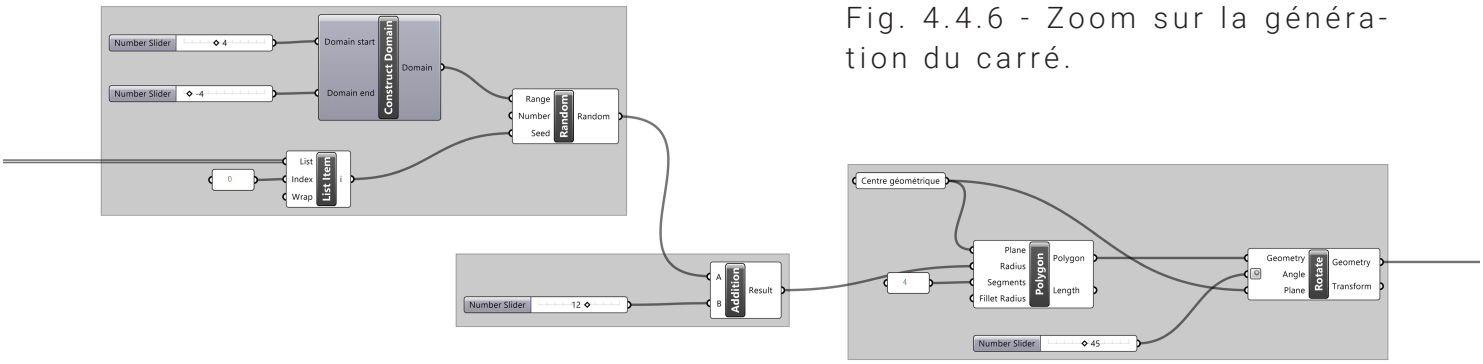



Fig. 4.4.6 - Zoom sur la génération du carré.

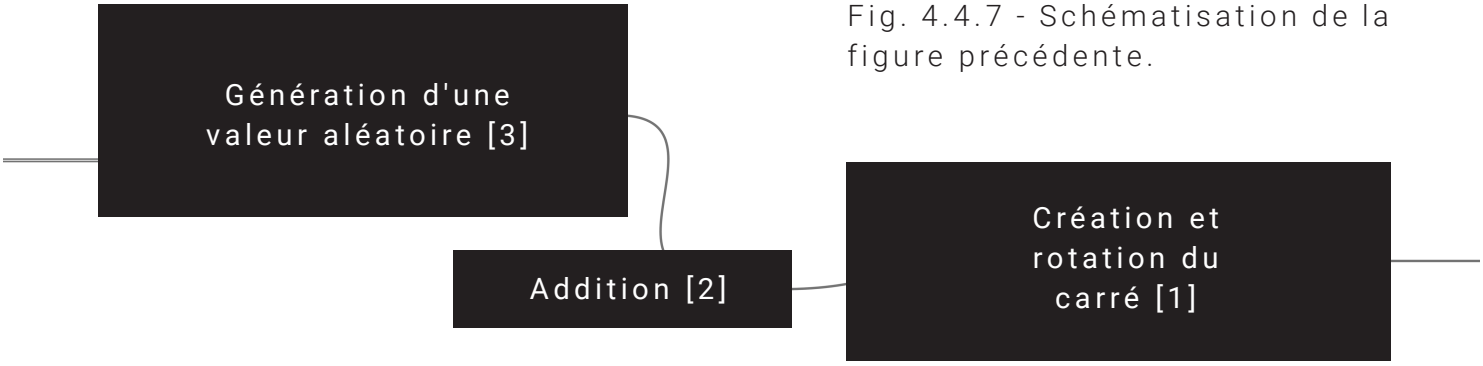



Fig. 4.4.7 - Schématisation de la figure précédente.

CARRÉ

ROND

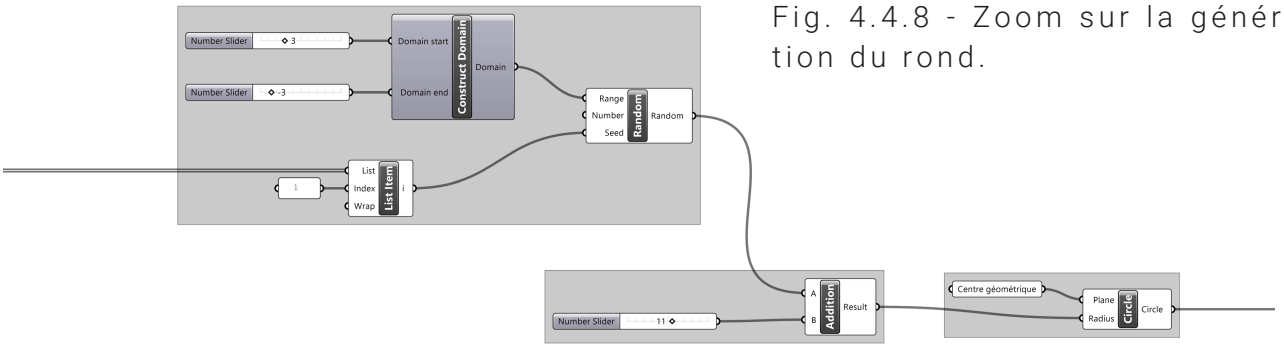



Fig. 4.4.8 - Zoom sur la génération du rond.

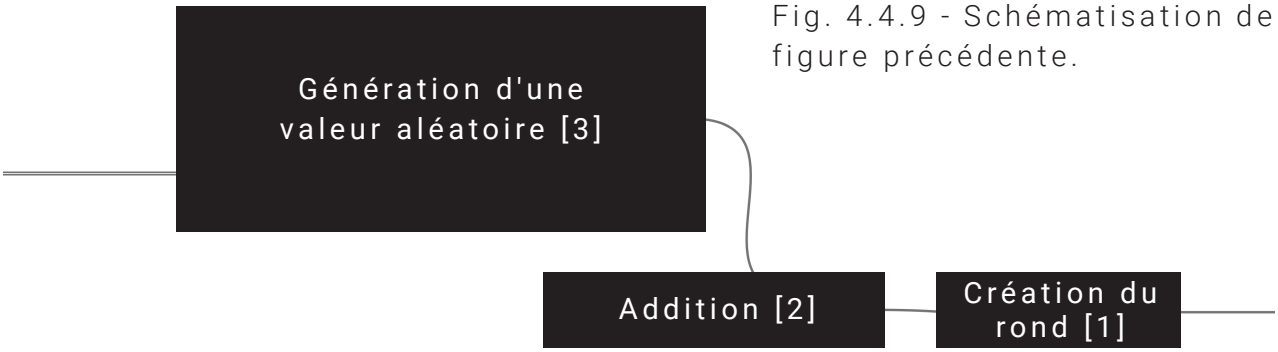



Fig. 4.4.9 - Schématisation de la figure précédente.

Afin d'explorer un peu les limites du réseau de neurones, je vais ajouter 2 formes plus complexes, que sont le **blob** et le polygone irrégulier.

Je réutilise la méthode employée pour les chiffres : je place manuellement des points dans **Rhinocéros**, qui vont ensuite être déplacés légèrement et aléatoirement dans les axes X et Y. La section [1] va convertir ces nouveaux points en courbes, formant des **blobs**. Ces **blobs** peuvent ensuite être aisément convertis en polygone irréguliers.

Le nouveau modèle paramétrique est en place, et paré à générer des formes diverses qui vont varier aléatoirement. Voyons ce que cela donne ! (plus de résultats en annexes).

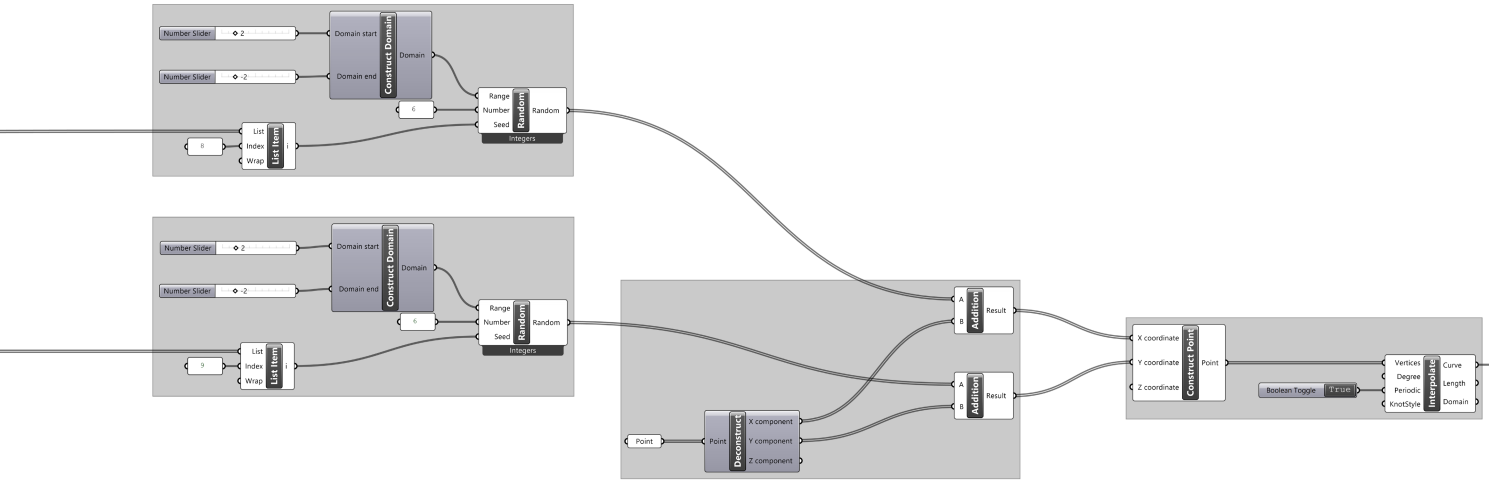



Fig. 4.4.10 - Zoom sur la génération d'un "**blob**".

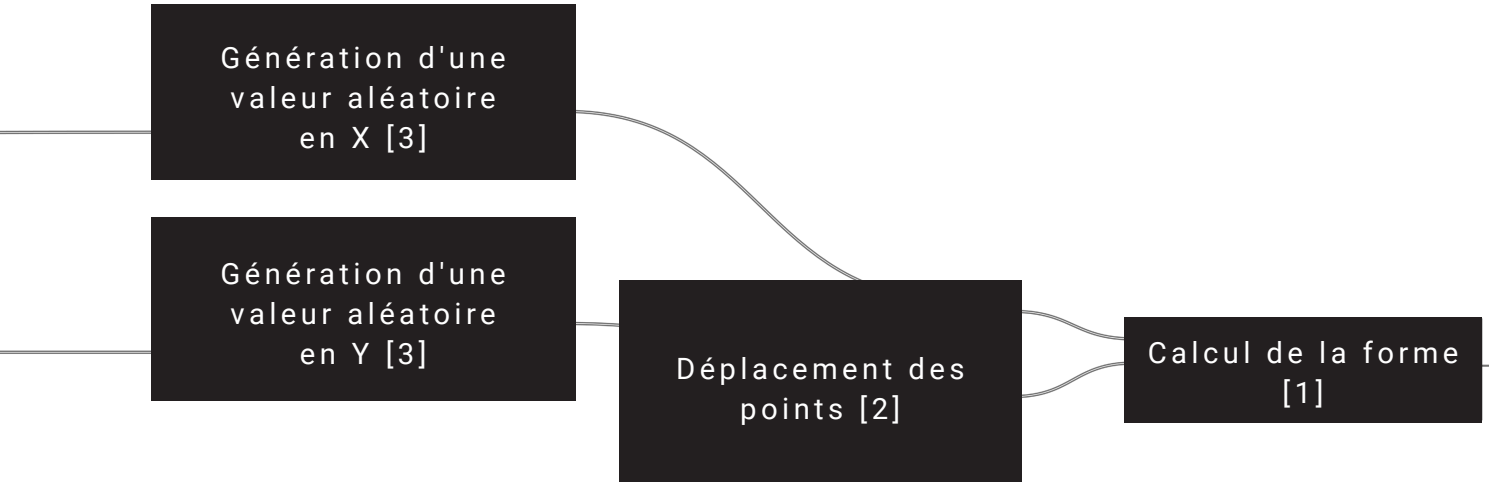



Fig. 4.4.11 - Schématisation de la figure précédente.

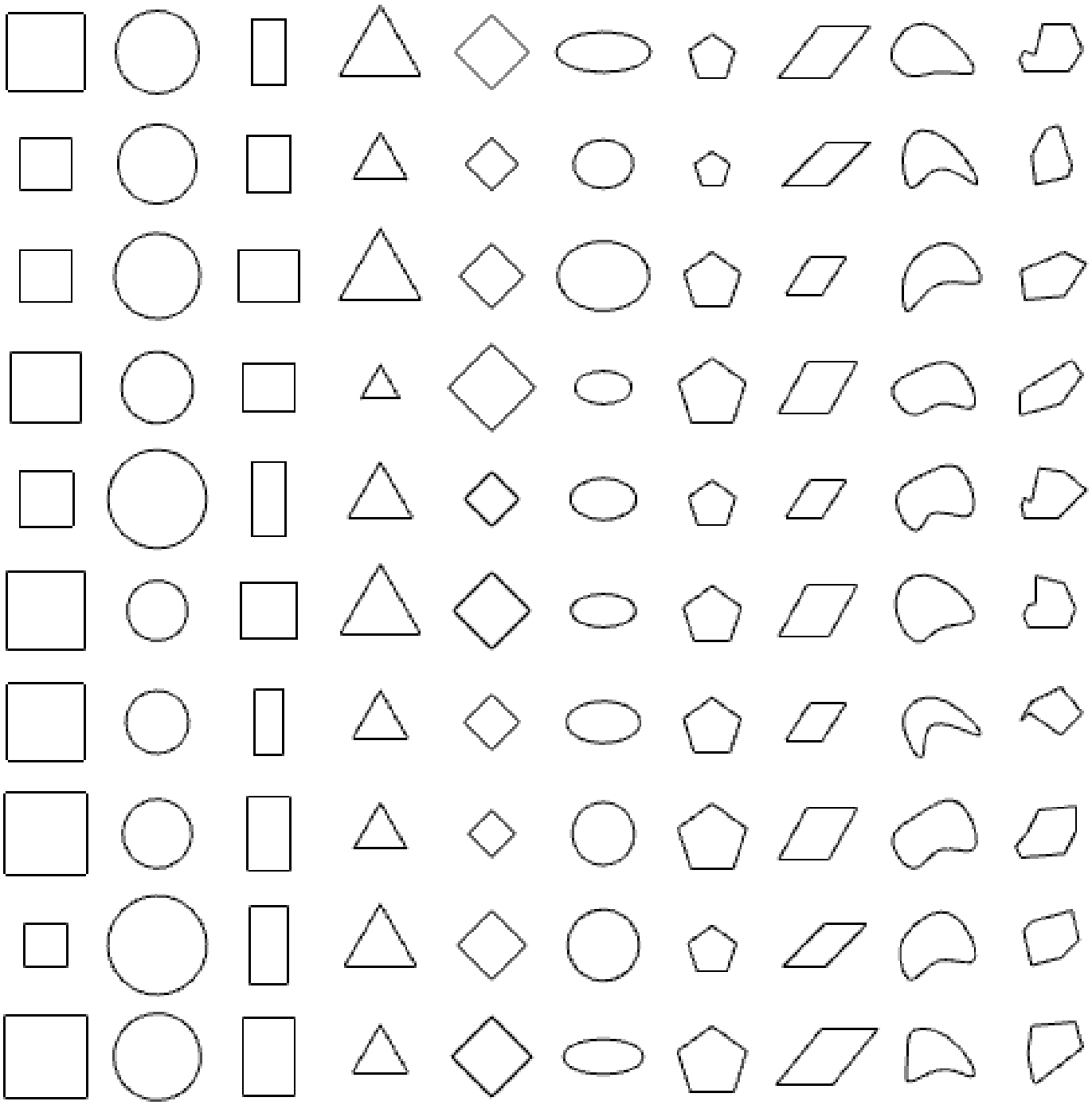


Fig. 4.4.12 - Fragment des formes générées par le modèle paramétrique.

J'observe certes des variations moins surprenantes que celles des chiffres, mais je comprends la dynamique de chaque forme, sa logique, et par conséquent le **GAN** serait en capacité de l'apprendre également.

Comme pour la base de données de chiffres, je génère 7000 images en noir et blanc et de même résolution, disposées dans la même hiérarchie de fichiers afin de les convertir en **ubytes** lisibles par le réseau de neurones.

*4.4.c Le réseau de neurones*

Le **GAN** reste identique, tant dans sa forme que dans ses paramètres. Je garde ainsi le même nombre d'**itérations** (700) pour une base de données paramétrique de même taille (7000 images). Le seul changement réside dans les fichiers **ubytes** que le **GAN** va aller chercher sur le disque dur : ils ne contiennent plus des images de chiffres compressés, mais nos images de formes. Passons directement aux résultats !

*4.4.d Exploitation des résultats*

Lors de l'étude des résultats de génération des chiffres j'ai abordé plusieurs sélections de résultats pour en observer l'évolution. Sur tous les essais que j'ai pu faire avec les formes je retrouve la même logique : des creux d'apprentissages, des résultats meilleurs au fil du temps mais variant d'un essai à l'autre, et une stabilisation de l'apprentissage au cours des dernières **itérations**. Aussi je ne montrerai ici que la dernière **itération**, toutes les autres se trouvant bien sûr en annexes.

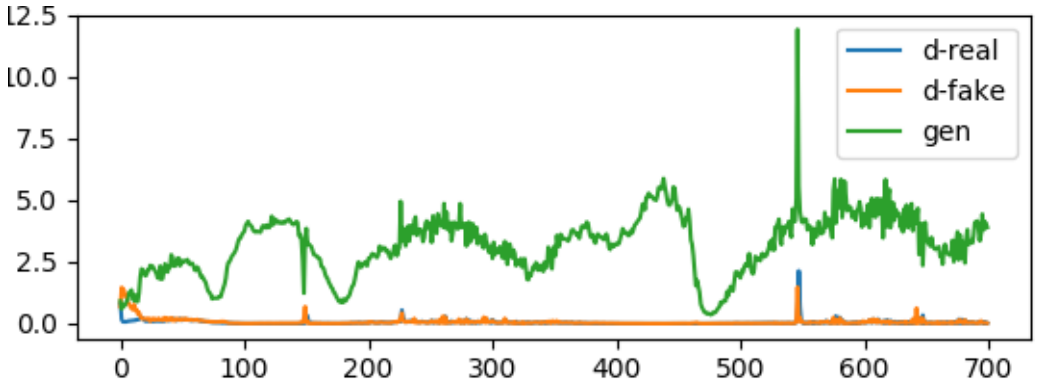
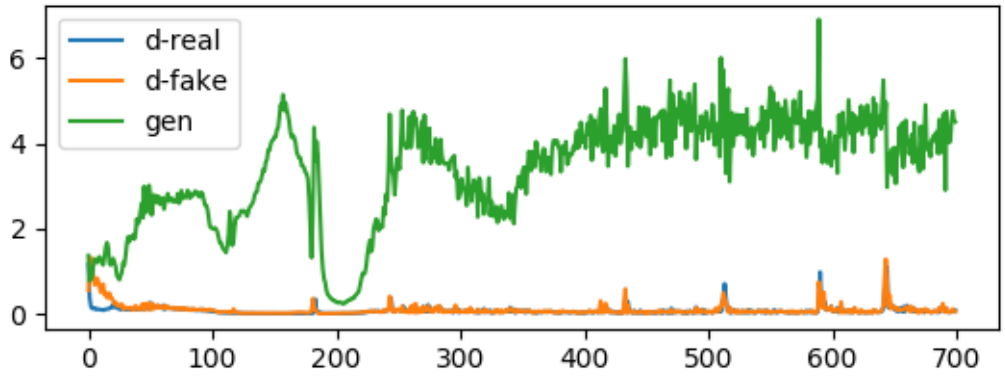



Fig. 4.4.13 - Statistiques de l'essai A (dessus) et B (dessous).



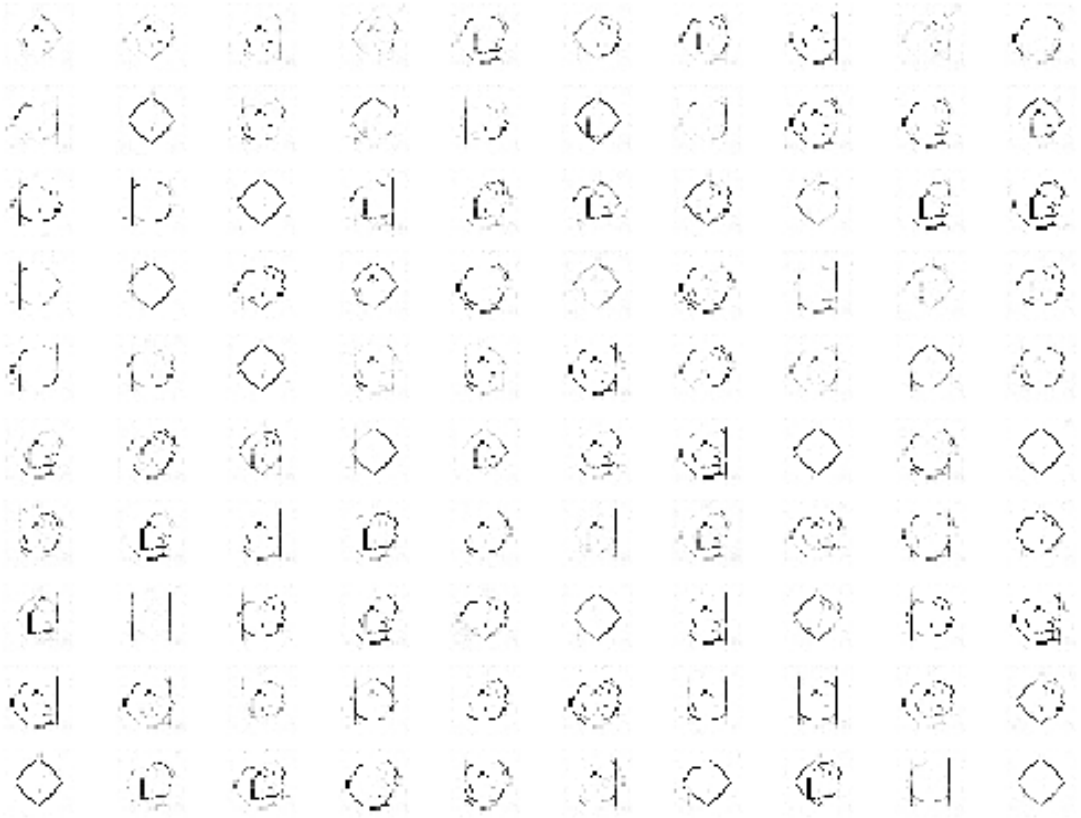


Fig. 4.4.14 - Essai A, noir sur blanc, 700ème **itération**.

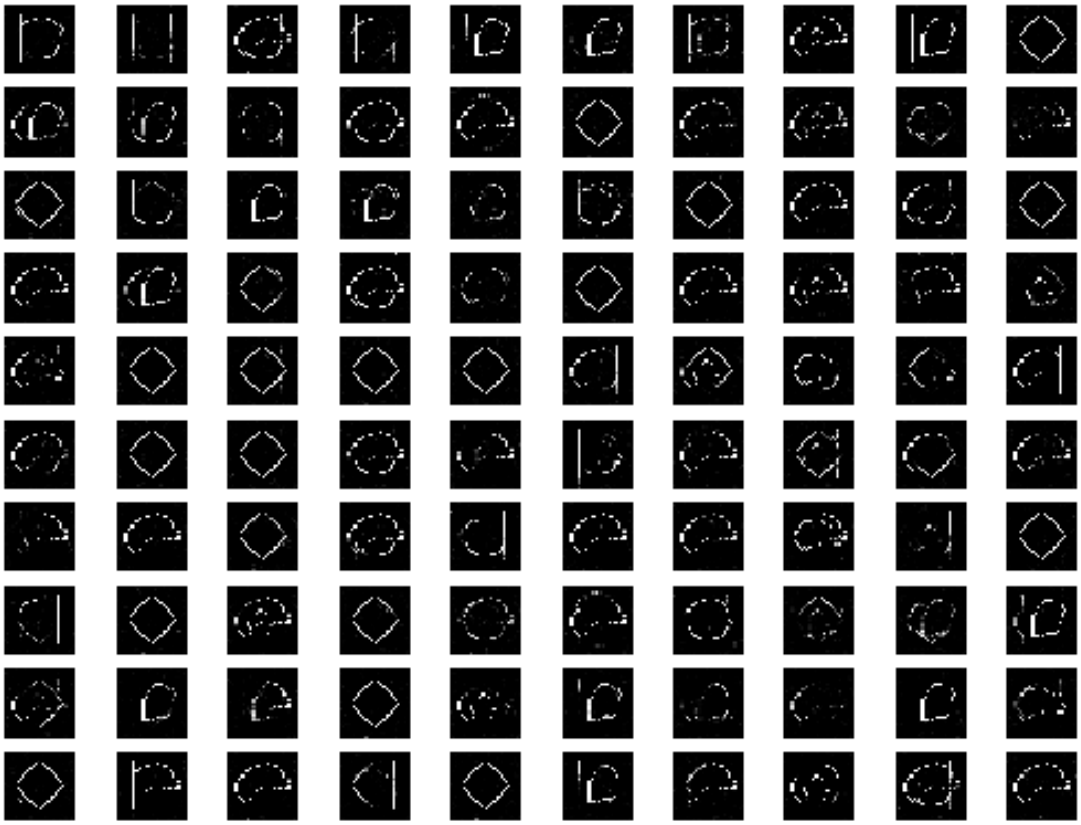


Fig. 4.4.15 - Essai B, blanc sur noir, 700ème **itération**.

Malgré les courbes d'apprentissages différentes, les essais A et B en figures 4.4.14 et 4.4.15 sont relativement similaires. J'arrive clairement à percevoir des formes se dessiner. Cependant je note une très grande présence de ce qui ressemble à un cercle, et relativement peu des autres formes. C'est là l'un des premiers biais de l'adaptation d'un réseau de neurones sur des données différentes.
Ce problème reste tout de même très léger, car je pense en avoir trouvé la source. Pour l'expliquer correctement, je replace ci-dessous le morceau de code du **GAN** générant les images de formes.

```
from keras.models import load_model
from numpy.random import randn
from matplotlib import pyplot
import matplotlib.pyplot as plt

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
        # generate points in the latent space
        x_input = randn(latent_dim * n_samples)
        # reshape into a batch of inputs for the network
        x_input = x_input.reshape(n_samples, latent_dim)
        return x_input

# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, n):
        # plot images
        for i in range(n * n):
                # define subplot
                pyplot.subplot(n, n, 1 + i)
                # turn off axis
                pyplot.axis('off')
                # plot raw pixel data
                pyplot.imshow(examples[i, :, :, 0], cmap='gray')
                pyplot.savefig('plot_300dpi.png', dpi=300)
        pyplot.show()

# load model
model = load_model('results_baseline/model_700.h5')
# generate images
latent_points = generate_latent_points(50, 100)
# generate images
X = model.predict(latent_points)
# plot the result
save_plot(X, 10)
```

Fig. 4.4.16 - Rappel du code du **GAN** générant les formes.

Pour rappel, cette section génère des points aléatoires dans l'**espace latent**, puis en déduit des images. Or, cette génération de points n'est pas parfaitement aléatoire : la fonction "*randn*" de la ligne indiquée va fournir des valeurs entre -1 et 1, et majoritairement autour de 0. Je me situe dans une zone précise de l'**espace latent**, et je constate que cette zone particulière correspond majoritairement à la forme du cercle. Je ne suis donc pas face à un disfonctionnement du système, mais plutôt à une de ces spécificités mises en lumière par un changement de données.

Il est légitime de se demander pourquoi je n'observais pas ce problème de génération du "même chiffre" avec l'expérience précédente. Pour être honnête, ça l'était car il y avait visiblement une plus grande présence de 8 et de 6 que de 2 par exemple.

Quand je regarde le graphique d'apprentissage en figure 4.4.13, je me rends compte que le **générateur** se stabilise plus vite que pour les chiffres. Je suppose que l'apprentissage a été plus rapide, et donc plus simple, et par conséquent un **espace latent** moins dense en solutions radicalement différentes.

Malgré la domination de cercles, je repère tout de même quelques étrangetés et des chimères d'angles et de traits droits. Je peux donc avancer que cette seconde expérience est une réussite, et les aléas ont permis de confirmer le processus expérimental. En effet, en gardant le réseau de neurones intact et en ne faisant varier que la base de données, cela permet de mettre en exergue certains aspects du fonctionnement du **GAN**, d'y réfléchir et proposer des hypothèses. Ce travail d'investigation et de compréhension du phénomène toujours assez abstrait qu'est l'apprentissage machine est très riche, mais m'éloigne un peu de la problématique générale. L'enrichissement est-il accompli ?

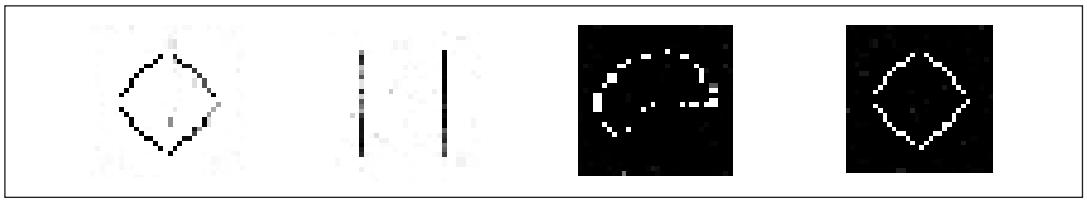


Fig. 4.4.17 - Selection de formes fidèles.

Bien que je puisse clairement distinguer la présence de cercles, je peux également discerner un losange parmi ces cercles, bien qu'ils semblent un peu mélangés tout de même. En piochant dans l'essai en blanc sur noir, je distingue un beau rond, ainsi que le fameux « **blob** ». Je peux ainsi valider le facteur « formes fidèles », malgré quelques aléas et une répartition inégale.

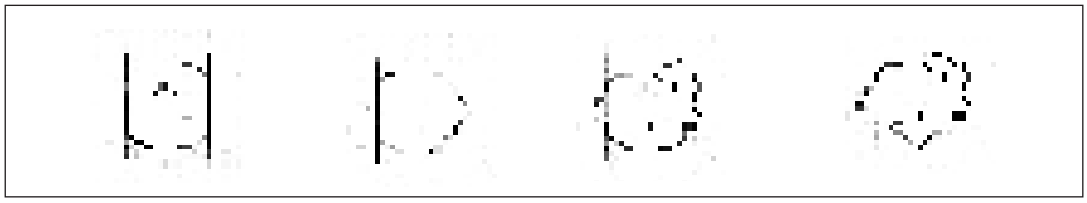


Fig. 4.4.18 - Selection de chimères.

Les chimères sont quant à elles beaucoup plus courantes et variées. J'identifie ainsi des mélanges entre rond et rectangle, parfois un **blob** s'invite, et en dernière image de la sélection, un intéressant triple mélange entre rond, **blob** et une base de losange.

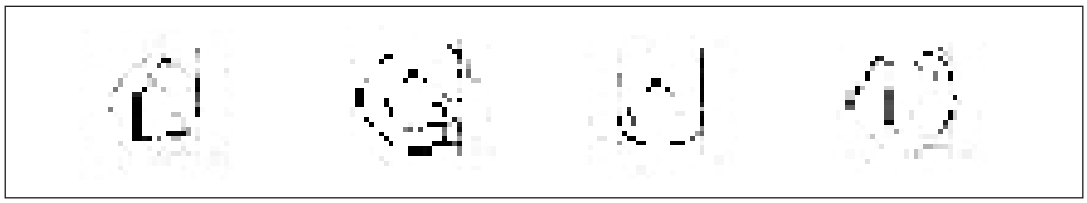


Fig. 4.4.19 - Selection d'étrangetés.

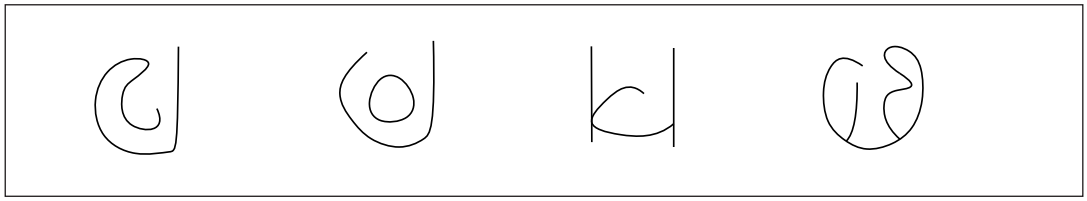


Fig. 4.4.20 - Retraçage des étrangetés.

Enfin, les étrangetés. Là aussi assez fréquentes, elles sont d'autant plus étranges qu'elles se basent sur des formes, dont le potentiel d'abstraction est plus grand que celui des chiffres. Je me suis encore une fois amusé à les retracer en vectoriel, et les résultats sont assez surprenants. J'ai toujours cette impression d'être face à un alphabet que je ne connais pas, ou des écritures anciennes gravées dans la roche. Au final, cette impression renforce l'idée que ces formes et caractères pourraient exister, et leur essence est proche des symboles que nous utilisons tous les jours. Le **GAN** a encore une fois validé un apprentissage cohérent, sur les 3 critères clés du tableau ci-contre.

| Critère / Expérience | Fidélité | Chimères | Etrangetés |
|---|---|---|---|
| Partie 1 "chiffres" | ✓ | ✓ | ✓ |
| Partie 2 "formes" | ✓ | ✓ | ✓ |
| Partie 3 "dispositions" | | | |

Tab. 4.3.1 - Tableau bilan mis à jour 1.

Tant de réussite en est presque émouvant, et je peux me permettre de passer à la dernière expérience, la disposition.

## 4.5 Partie 3 : les dispositions

Tout comme j'ai pu le faire avec l'expérience précédente, il est ici question de tester un nouveau critère : la disposition des éléments. Jusque-là, les formes comme les chiffres étaient centrés, au centre de chaque image produite constituant la base de données. L'objectif de cette troisième et dernière expérimentation (car il faut s'arrêter un jour) est donc de tester si un changement de disposition de l'élément sur une image constitue un changement pour le **GAN**, et s'il est capable d'apprendre tout aussi bien qu'avec les chiffres ou les formes.

### *4.5.a Mise en place de la stratégie*

Etant donné que je vais m'intéresser principalement à l'emplacement des éléments sur l'image, autant diminuer les variables possibles. Aussi je passerai à des formes encore plus simples que dans la partie précédente, soit uniquement des variations de rectangle. Ces rectangles vont donc, en plus de varier légèrement en dimensions, varier également dans leur position dans l'image. Certains seront en haut à gauche, en bas à droit, au centre…

Dans la première expérience je travaillais sur 10 chiffres différents. Dans la seconde sur 10 formes différentes. Il y aura en toute logique ici 10 dispositions différentes à l'origine de la base de données : dans chaque coin, chaque centre de chaque côté et au centre de l'image. Le schéma ci-dessous montre les différentes possibilités de positions qui s'offrent à moi :

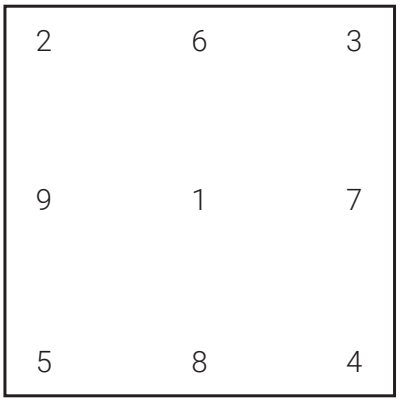



Fig. 4.5.1 - Schéma des positions possibles.

Cette approche du positionnement des éléments semble dès lors plus proche de l'architecture. Bien que ce lien sera encore plus flagrant pendant la présentation des résultats, la stratégie proposée s'apparente déjà à un procédé urbanistique. Certes dans sa conception la plus simple, mais tout de même. Etudier la position d'une forme dans une image, un rectangle d'autant plus, revient à placer un bâtiment dans une parcelle. Comme ce mémoire se concentre sur l'enrichissement, la stratégie de conception de la base de données paramétrique sera focalisée sur la simplicité de reconnaissance, des dispositions claires et explicites.

Cependant j'imagine facilement la possibilité de générer des positions paramétriques prenant en compte, pourquoi, pas l'apport solaire, les bâtiments voisins, ou même des éléments courant du **PLU** comme la surface à bâtir, les retraits ou alignements, les vues… Le potentiel est alors lié à la complexité du modèle : plus j'en ajoute, plus les résultats seront exploitables. Cependant il faut commencer par s'assurer que ce modèle tienne la route, et donc commencer par des données simples et vérifier que notre **GAN** fonctionne correctement.

*4.4.b Le modèle paramétrique*

Le modèle paramétrique se base sur la même logique que celui des chiffres, mais a nécessité quelques modifications. Même si les formes produites sont plus simples que les modèles précédents, la logique de fonctionnent est assez différente car il ne s'agit plus que de formes, mais d'un travail sur leur emplacement.

Malgré tout, on retrouve dans le schéma en figure 4.5.3 des sections inchangées des modèles précédents, que j'ai déjà abordé : la capture de chaque **itération** [E] ou encore la création de variables aléatoires [C]. La section [B] « *déplacement aléatoire des points d'ancrage* » est identique à ce qui avait été fait pour le cas des chiffres (voir figure 4.3.7 et explications en lien). Un détail change : je ne fais plus varier les points d'ancrage pour des chiffres mais pour des centres de rectangles, il y a donc moins de points à faire varier.

Le fonctionnement global de ce modèle reste assez simple : je crée des points à chaque angle et chaque côté comme l'illustrait le schéma en figure 4.5.1, et je fais varier légèrement leurs coordonnées aléatoirement. Il faut ensuite construire un rectangle sur ces points, rectangles dont les dimensions varient également aléatoirement. Voyons cela de plus près !

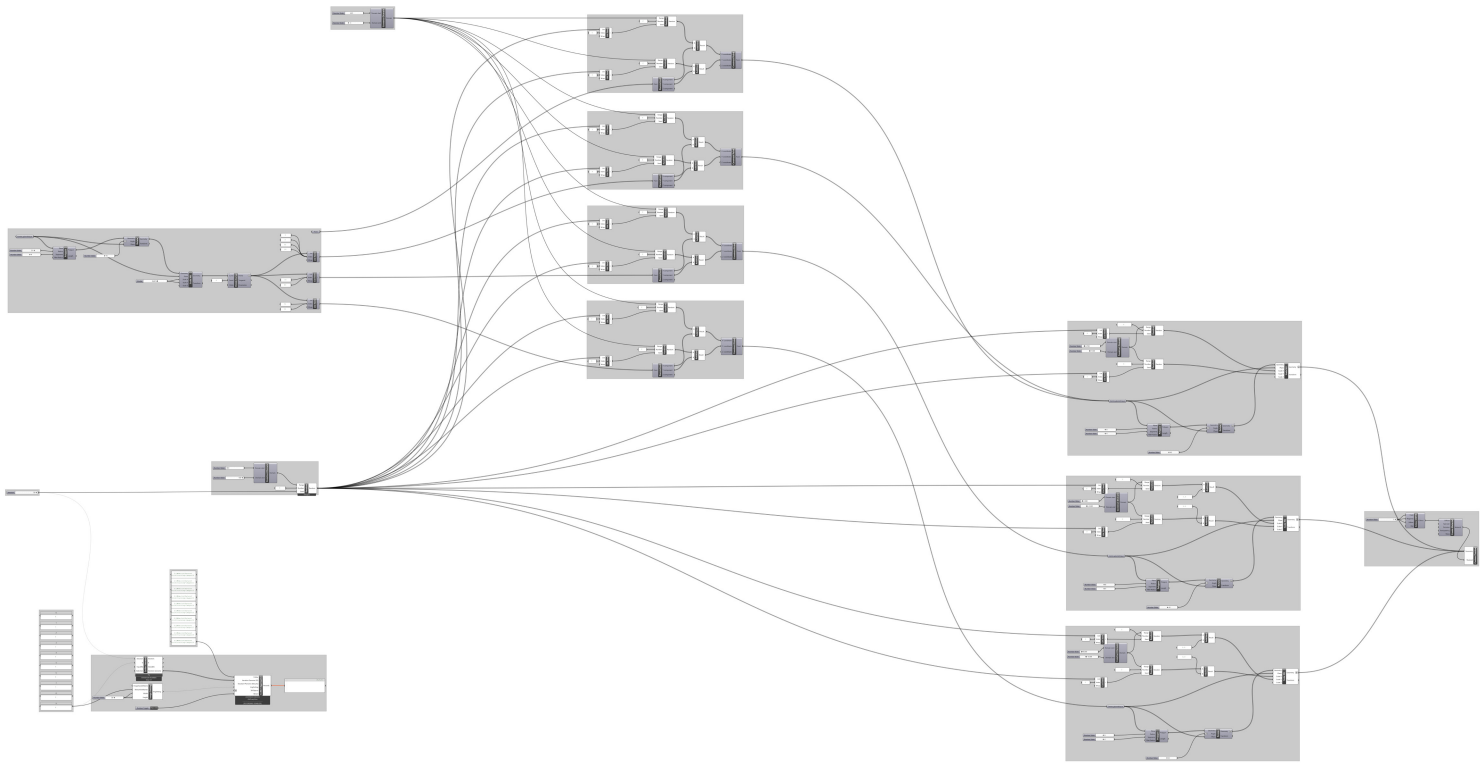


Fig. 4.5.2 - Modèle paramétrique générant les dispositions.

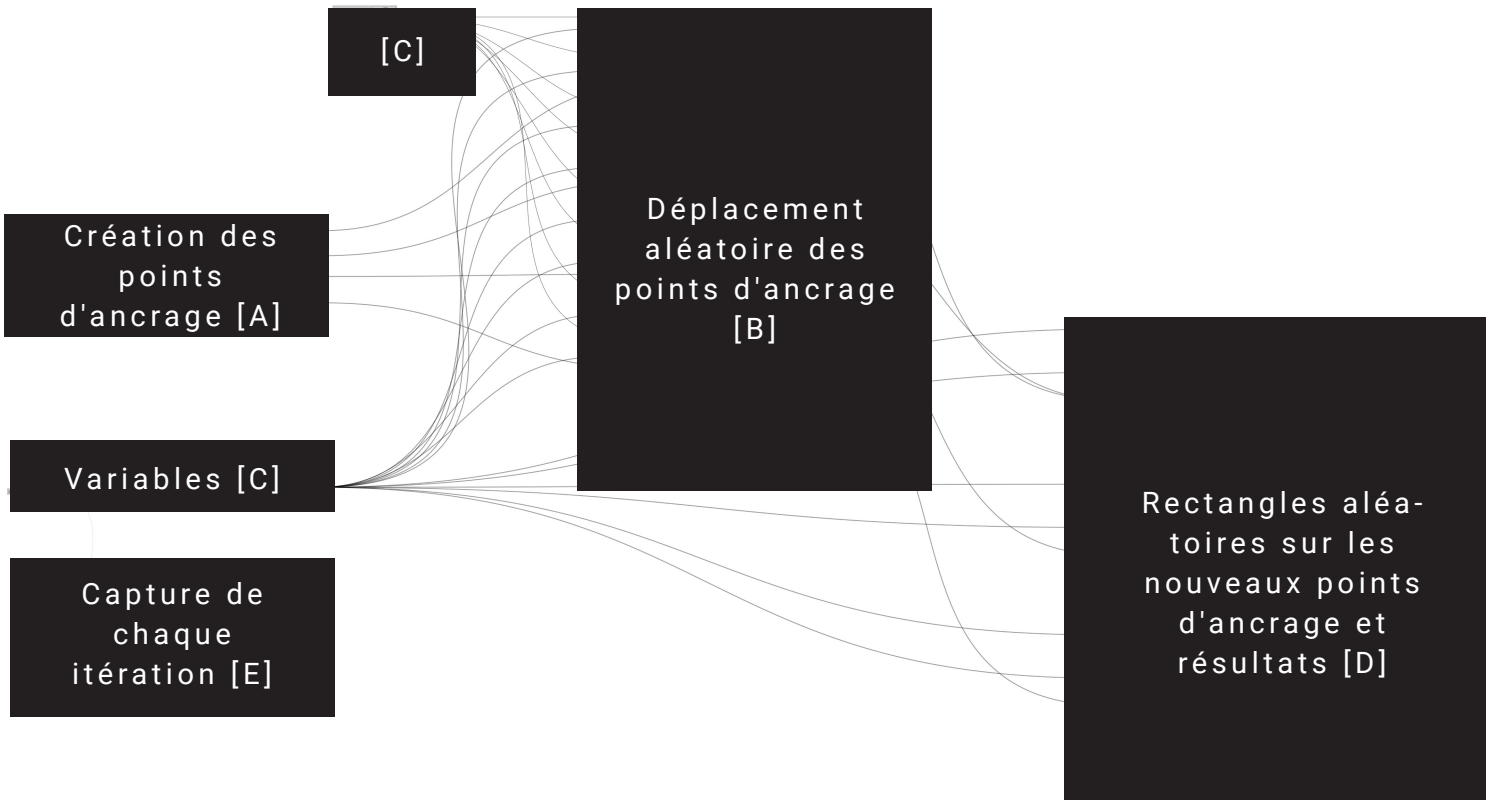



Fig. 4.5.3 - Schéma de la figure précédente.

[A]
Fig 4.5.3

La première section [A], détaillée ci-contre, est donc dédiée à la création des points d'ancrage. Pour y parvenir 3 sous-étapes sont nécessaires comme on peut le voir sur le schéma en figure 4.3.11.

Je commence par créer le contour de chaque future image [1], soit la parcelle dans laquelle se situera prochainement le rectangle. Les carrés obtenus correspondant à la première ligne du schéma en figure 4.5.4.

Ensuite, en réduisant la taille du carré de base, j'obtiens un carré interne (seconde ligne du schéma), dont je vais extraire les angles et les centres des côtés (troisième ligne) : j'ai ainsi la totalité des points d'ancrage possibles.

Enfin, je sélectionne les points d'ancrage un à un grâce à la section [3] du schéma en figure 4.5.6, afin de trier ceux qui sont dans les angles, les côtés horizontaux, verticaux, et j'y rajoute les centres. J'ai ainsi tous les points d'ancrage (le schéma en figure 4.5.4 ne montre que les 6 premiers), il faut maintenant les transformer en rectangles.

[D]
Fig 4.5.3

Dernière étape de ce modèle paramétrique, la section [D] du schéma global vient créer les rectangles et les afficher. Pour cela j'utilise le même procédé utilisé dans la création des rectangles durant l'expérience basée sur les formes. Pour rappel, je crée un polygone à 4 côtés (un carré donc) que je vais aléatoirement étirer selon les axes horizontaux et verticaux pour donner un rectangle. Ces rectangles ont cependant la particularité d'avoir des points d'ancrage différents, et j'obtiens ce que l'on peut voir en cinquième et dernière ligne du schéma ci-dessous.

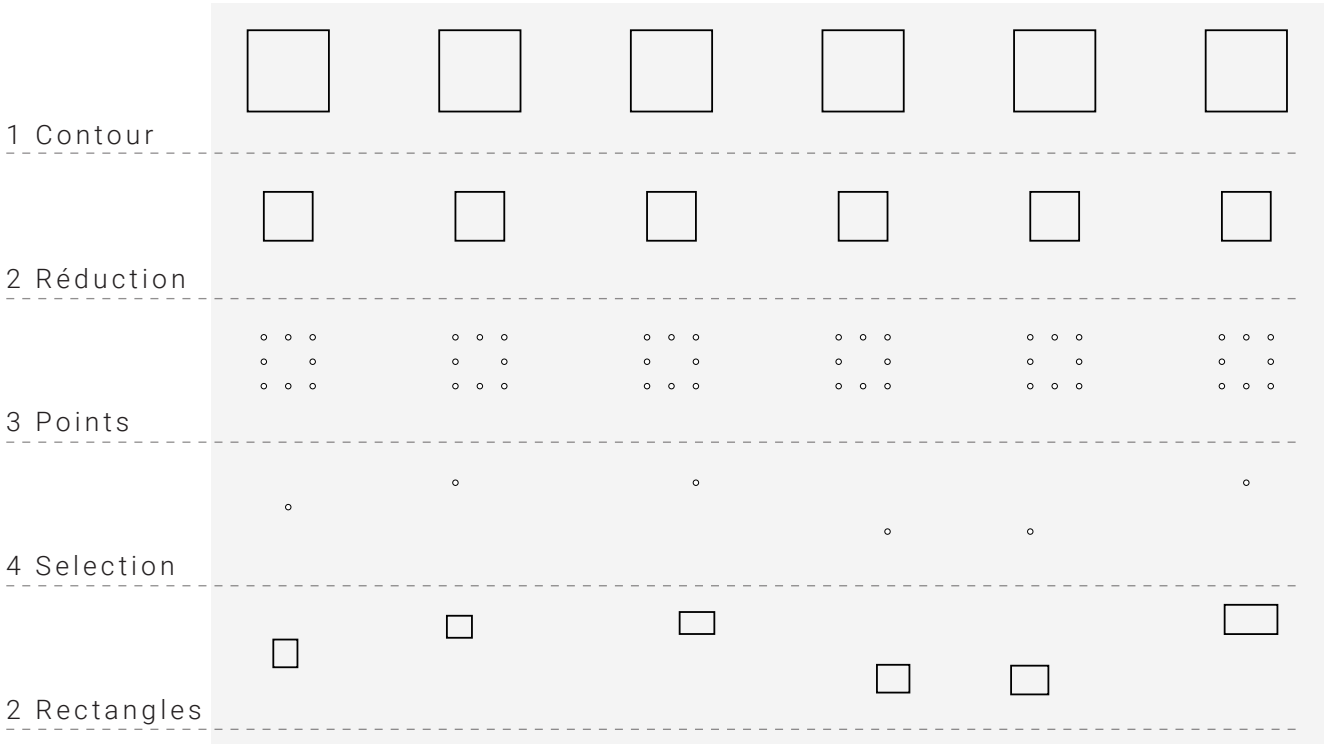



Fig. 4.5.4 - Aperçu des résultats par étape.

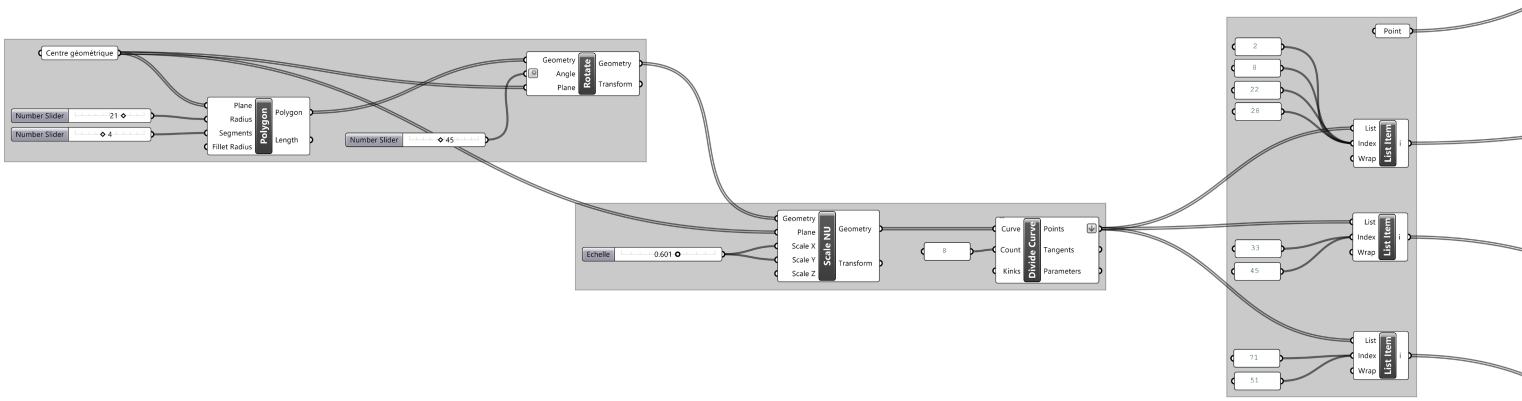



Fig. 4.5.5 - Zoom sur la section [A] du schéma global, la création des points d'ancrage.


Fig. 4.5.6 - Schéma de la figure précédente.

Voici donc un fragment de la base de données, constituée de la variation de dispositions de rectangles dans une image délimitée. Pour les regards les plus aguerris, force est de constater qu'il manque une colonne : en effet, il n'y a pas 10 dispositions mais 9, car en prenant les angles, les centres des côtés et le centre d'un carré, j'aboutis à 9 possibilités et non 10. En l'espèce cela ne change pas grand-chose, car un bug dans le code fait que seules 9 classes sur 10 sont apprises par le **GAN**.

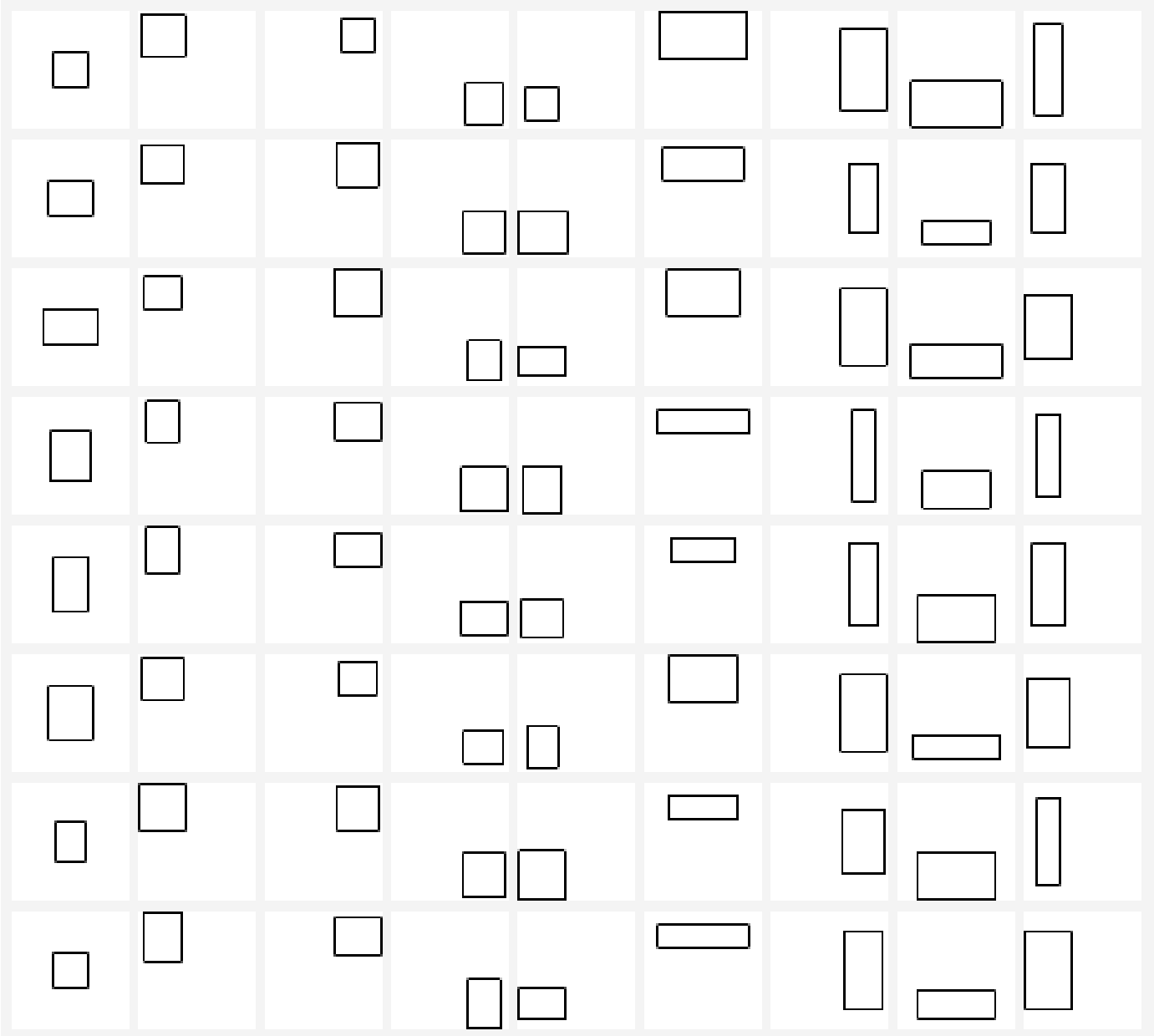


Fig. 4.5.7 - Exemples de dispositions vides générées par le modèle.

Pour rapprocher ces rectangles de l'idée que j'ai d'une "empreinte de bâtiment", je reprends le même modèle et je rajoute quelques composants venant remplir ces rectangles. Cela permet de créer un changement radical d'aspect des données, et pour un minimum d'efforts (encore une fois, j'observe cette tendance à vouloir rentabiliser l'investissement d'autant de temps dans la confection du modèle paramétrique). Me voici donc avec 2 bases de données pour le prix d'une (plus d'exemples encore en annexes). Le modèle **Grasshopper** et ses **graines** restant identiques, les dispositions sont donc les mêmes dans les deux jeux de données, seul le remplissage change. La comparaison en sera d'autant plus pertinente.

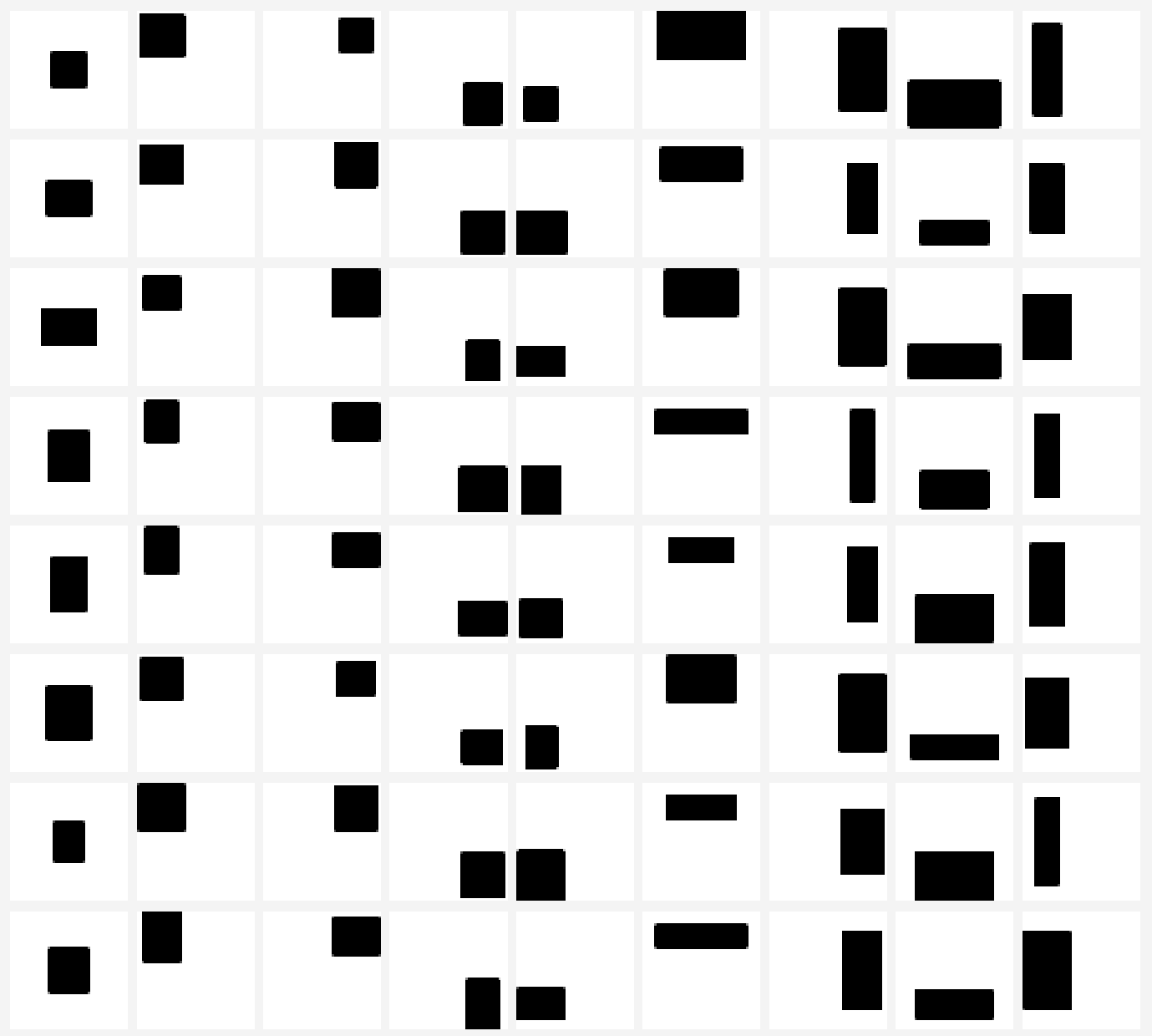


Fig. 4.5.8 - Exemples de dispositions pleines générées par le modèle.

*4.5.c Le réseau de neurones*

A l'instar de l'expérience précédente, je garde le **GAN** identique ainsi que ses paramètres, avec autant d'**itérations** et une base de données de même envergure.

*4.5.d Exploitation des résultats*

L'analyse des résultats est ici un peu différente, car je n'ai pas seulement deux essais à observer (noir sur blanc et blanc sur noir), mais 4 grâce aux essais de dispositions vides et pleines.

Pour avoir une approche plus méthodique, et pour faire durer le suspens une page de plus, je peux commencer par comparer les différentes statistiques avant les résultats graphiques. Ci-contre se trouve donc l'évolution de l'apprentissage du **générateur**, pour les 4 essais A, B, C et D.

Sans entrer dans le détail, les dispositions en noir sur blanc (B et D) semblent avoir une tendance croissante et régulière, ce qui laisse à penser qu'avec plus d'**itérations** les résultats seraient meilleurs. Les dispositions pleines témoignent également de plus de régularité dans l'apprentissage que les formes vides.

Même si plusieurs tentatives de chaque essai auraient été nécessaires pour confirmer ces hypothèses, ce n'est pas vraiment le cœur de notre problématique. Cela permet néanmoins, au sein d'une même expérience, de se concentrer sur les « meilleurs » résultats et ainsi d'avoir une analyse plus cohérente vis-à-vis de la problématique du mémoire : l'enrichissement des données.

Ainsi, je peux considérer que les dispositions pleines et noir sur blanc paraissent les plus intéressantes, statistiquement parlant bien sûr. Voyons si cela est le cas graphiquement.

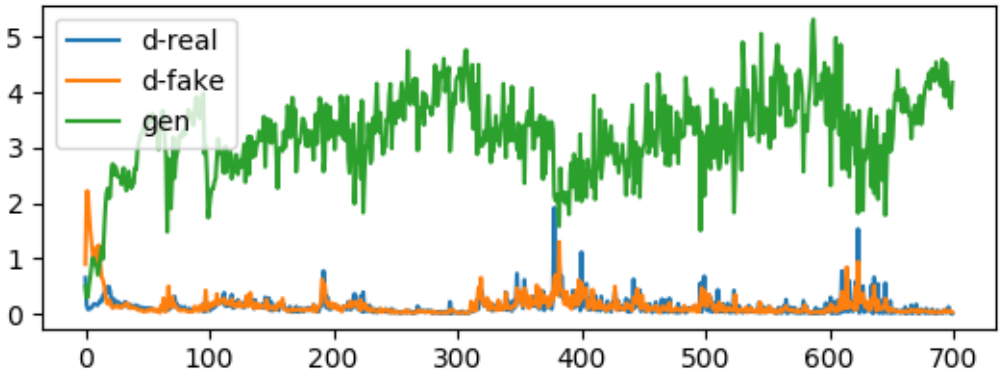



Fig. 4.5.9 - Essai A, dispositions vides blanc sur noir.

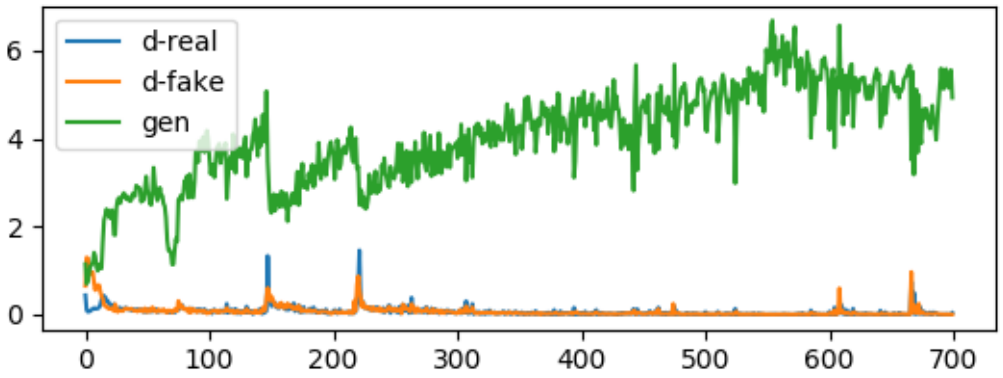



Fig. 4.5.10 - Essai B, dispositions vides noir sur blanc.

VIDES

PLEINES

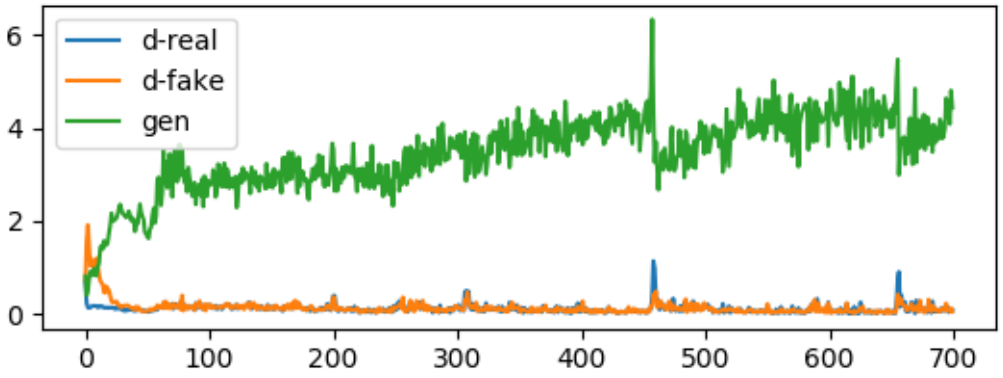



Fig. 4.5.11 - Essai C, dispositions pleines blanc sur noir.

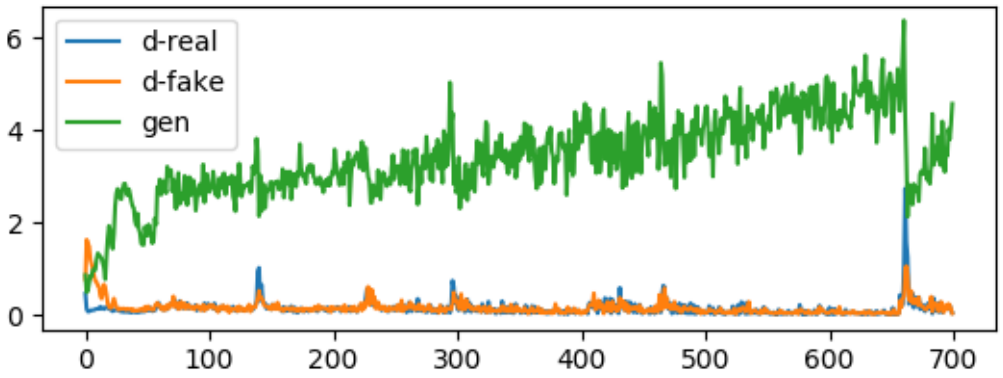



Fig. 4.5.12 - Essai D, dispositions pleines noir sur blanc.

## VIDES

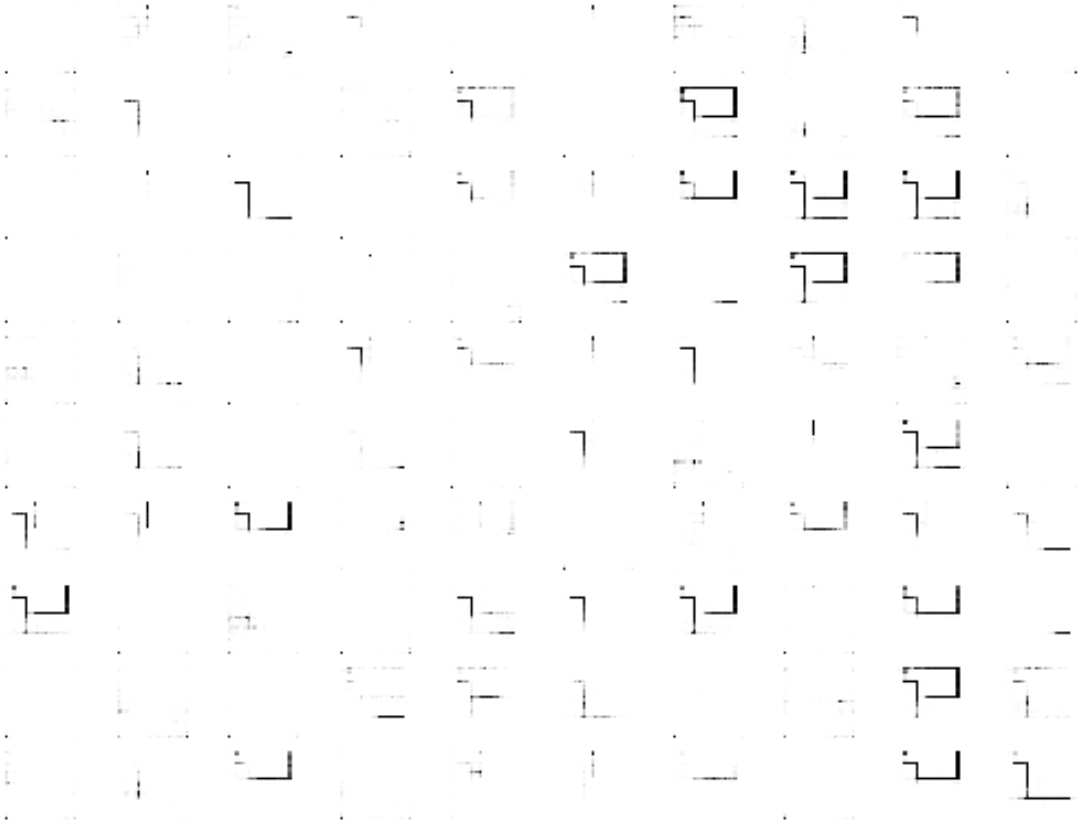


Fig. 4.5.13 - Essai A, noir sur blanc, 700ème **itération**.

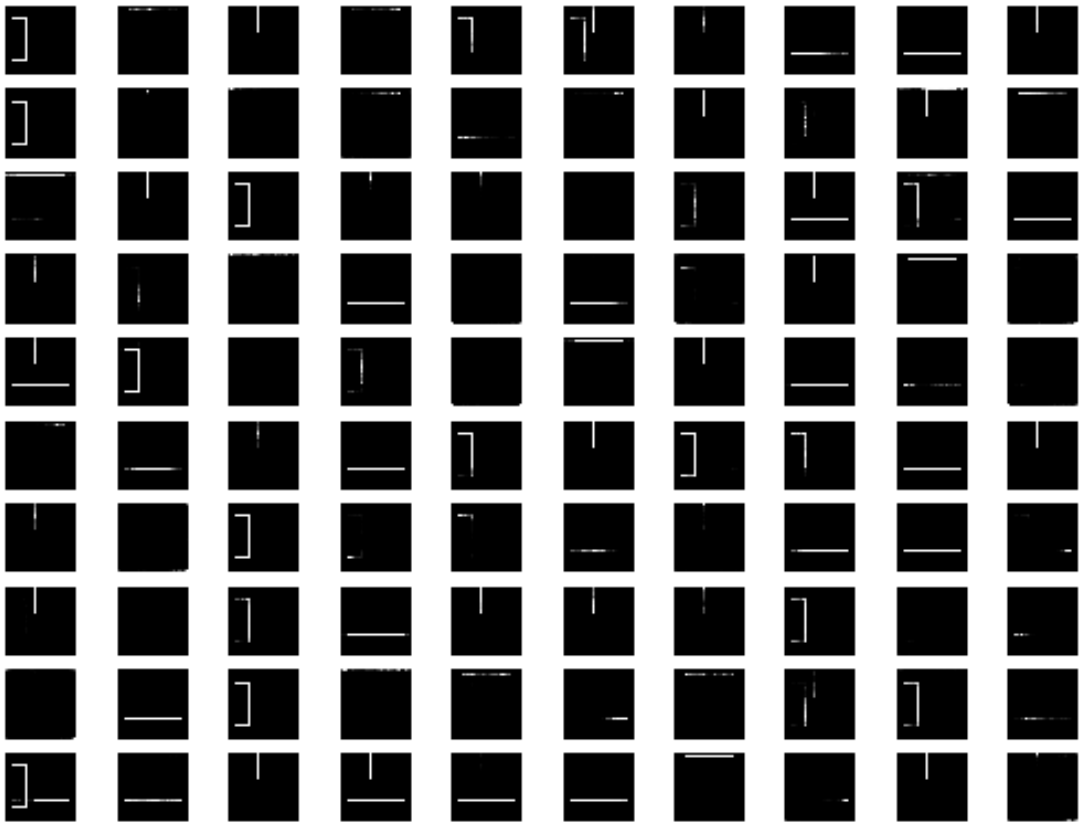


Fig. 4.5.14 - Essai B, blanc sur noir, 700ème **itération**.

## PLEINES

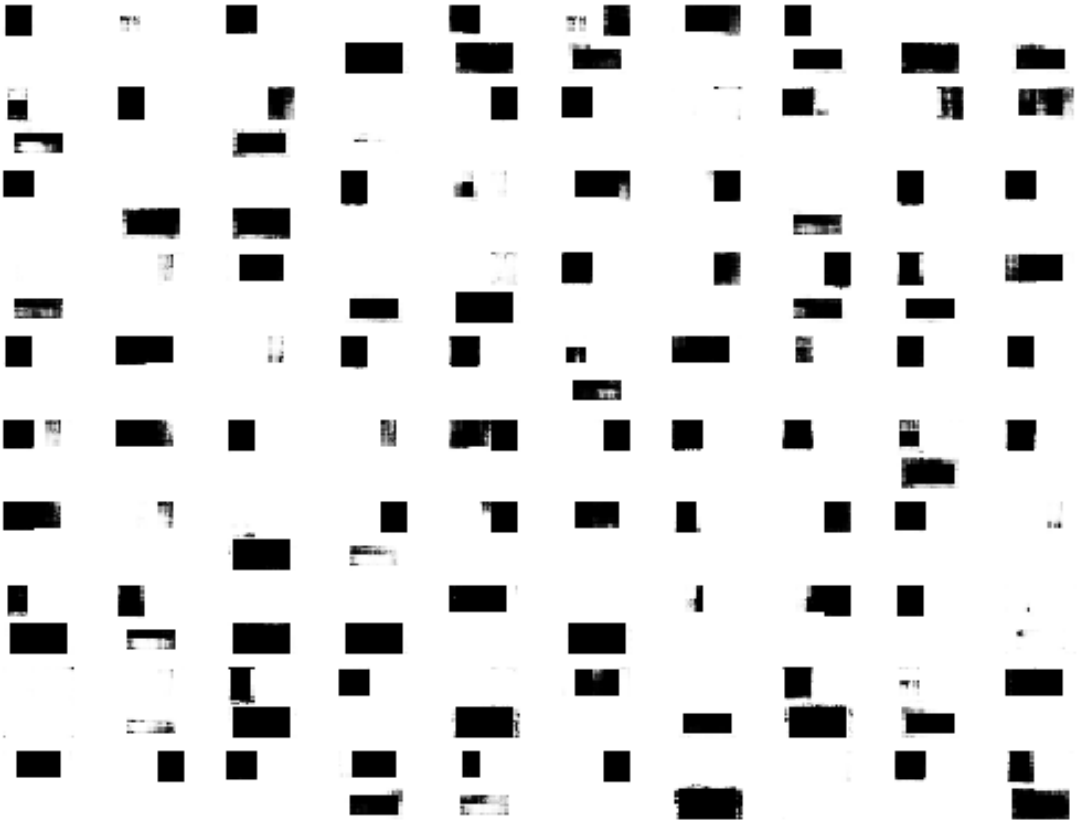


Fig. 4.5.15 - Essai C, noir sur blanc, 700ème **itération**.

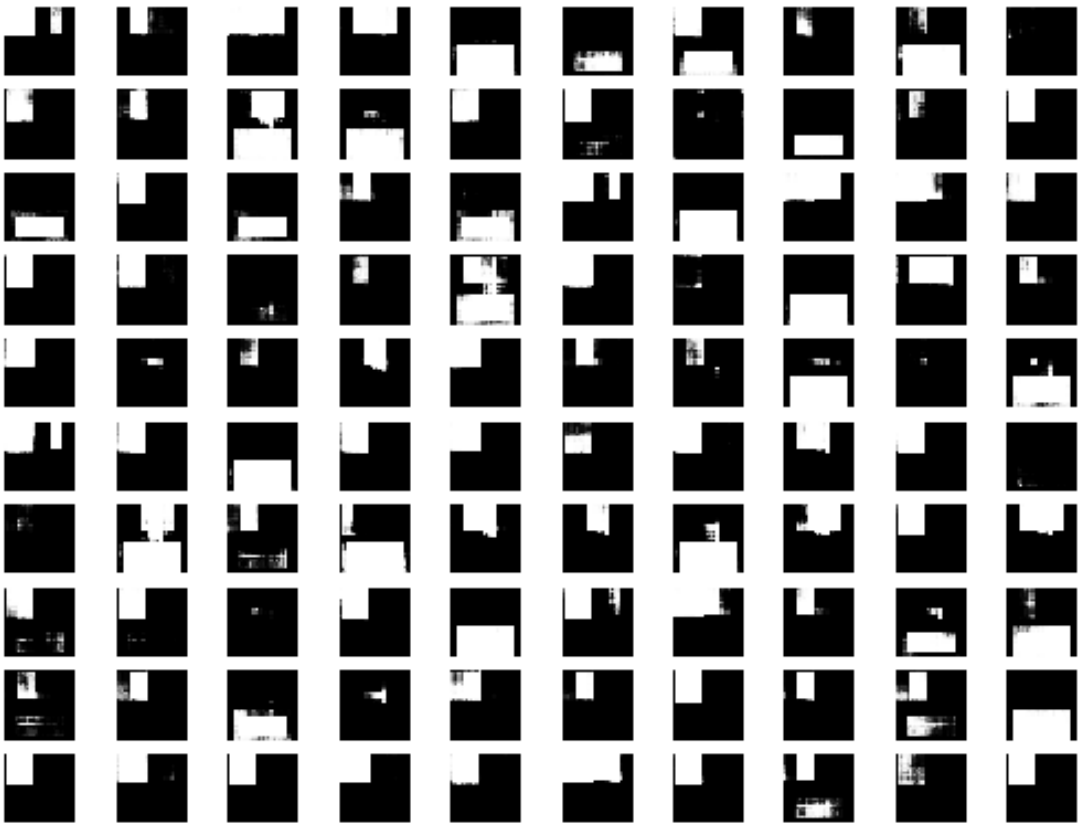


Fig. 4.5.16 - Essai D, blanc sur noir, 700ème **itération**.

Comme on l'apercevait dans les statistiques, les résultats en noir sur blanc semblent plus prometteurs, tout comme des dispositions pleines. Cependant, cette première analyse se base sur la reconnaissance rapide de dispositions que je connais. Parfois l'enrichissement se déclare de manière inattendue, sujet que j'aborderai un peu plus tard.

Je vais donc me concentrer premièrement sur les résultats en noir sur blanc, car leur lisibilité permettra d'identifier les critères habituels (fidélité, chimères et étrangetés) validant ou non la problématique.

Les sélections présentées ci-dessous ne proviennent pas toutes de la 700ème et dernière **itération** que l'on a en figure 4.4.13, mais également des **itérations** précédentes. En effet le modèle se stabilisant rapidement, je trouve des résultats intéressants très vite. Toutes les **itérations** sont comme à l'habitude présentes en annexes.

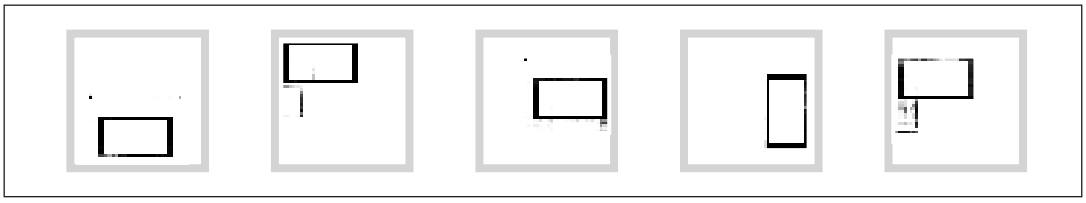


Fig. 4.5.17 - Selection de dispositions fidèles.

Les formes vides permettent d'identifier plusieurs dispositions : en haut, en bas, dans les angles… La disposition centrée sur le côté bas reste la plus représentée, tout comme j'observais une majorité de rond ou de 8 dans les expériences précédentes. Le fond de l'image étant blanc, il peut être difficile de comprendre où se trouvent les limites de la « parcelle » : un contour gris clair a été rajouté pour une meilleure visualisation.

Dès lors qu'il s'agit d'identifier des chimères, la recherche sur dispositions vides devient très difficile, et je serais tenté d'invalider ce critère. Mais miracle ! Quelqu'un a pensé à essayer une version « pleines », et la recherche est beaucoup plus probante.

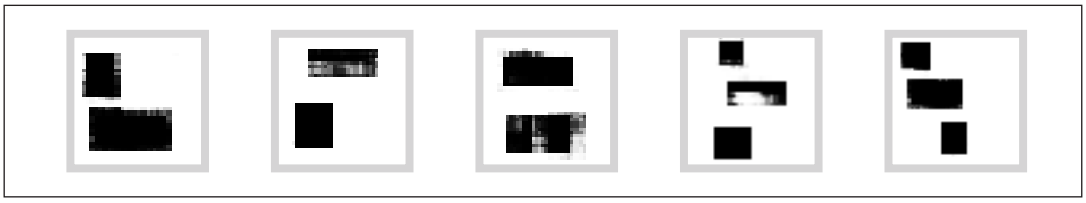


Fig. 4.5.18 - Selection de chimères.

Je retrouve bien de nombreux mélanges entre les dispositions existantes de la base de données paramétrique. Parfois à deux voir à 3 dispositions au sein d'une même solution. Ces chimères étaient assez évidentes, car très binaires dans leur construction. En effet soit j'ai une forme, soit je n'en ai pas, là où les résultats étaient plus flous, incertains mais d'autant plus créatifs pour les formes ou les chiffres.

En effet, en « remplissant » les dispositions, le **GAN** à compris le principe du rectangle bien tracé, de l'aspect tout noir de la forme et tout blanc du fond.

Je suis effectivement face à une situation particulière : le **GAN** a « trop » appris, il est considéré comme « surentrainé ». Comme un enfant qui apprendrait des années à dessiner un même objet, qui n'a plus que ça en tête et qui ne saurait pas dessiner autre chose. Il en va de même pour le réseau de neurones. Il sait générer des formes presque « parfaites » mais ce n'est pas exactement ce que je cherche, il faut des intermédiaires, des flous, des étrangetés, et ce plus tôt dans l'apprentissage.

Fig. 4.5.19 - Selection d'étrangetés.

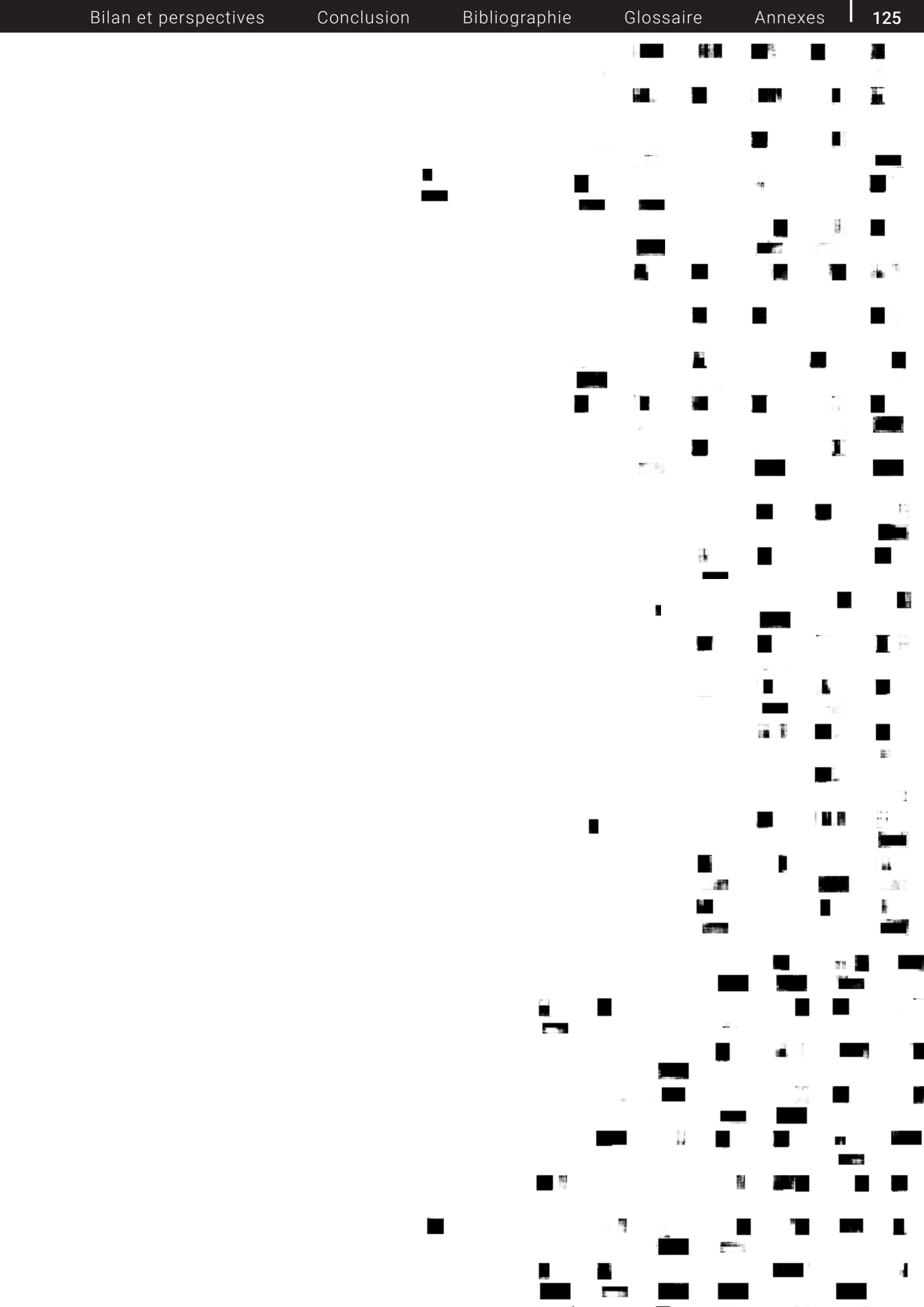
Ci-dessus je suis revenu à la 70ème **itération**, au tout début de l'entraînement. Je perçois des résultats plus riches, qui ne sont ni des fidélités, ni des chimères. De nouvelles dispositions prennent formes, telles que la diagonale. Je n'avais même pas pensé à mettre cette disposition dans le jeu de donnée, et je suis agréablement surpris de voir le **GAN** m'y faire penser.

Je suis parti d'une base de données aux dispositions simples et uniques, et j'obtiens une nouvelle base de données avec des possibilités intermédiaires, des croisements, des inventions quelquefois étranges, parfois intéressantes. J'ai donc bien affaire à un enrichissement de la base de données paramétrique, et la dernière ligne du tableau se complète.

| Critère / Expérience | Fidélité | Chimères | Etrangetés |
|---|---|---|---|
| Partie 1 "chiffres" | ✓ | ✓ | ✓ |
| Partie 2 "formes" | ✓ | ✓ | ✓ |
| Partie 3 "dispositions" | ✓ | ✓ | ✓ |

Tab. 4.5.20 - Tableau bilan mis à jour 2.

Ce joli tableau teinté de positivité est un solide argument pour valider l'hypothèse de ce mémoire, selon laquelle le processus proposé permet bien d'enrichir un espace de solution paramétrique initial. Cette conclusion n'est cependant que partielle, et la partie suivante dédiée aux résultats en eux-mêmes permettra de donner plus de relief à cette réflexion.



# 5
# BILAN ET PERSPECTIVES

## 5.1 Analyse critique des résultats

L'analyse des résultats de la partie précédente a permis de valider l'hypothèse du mémoire. Cependant, cette analyse était celle de quelqu'un voulant à tout prix valider son hypothèse, et il est temps d'avoir une dimension un peu plus critique des résultats obtenus.
Tout d'abord, je peux commencer par croiser les différentes expériences entre elles, et pas uniquement entre essais au sein d'une même expérience. Autrement dit, comparer les résultats du **GAN** sur les chiffres, les formes et les dispositions. Un des premiers éléments qui m'a sauté aux yeux concerne les statistiques d'apprentissage, et plus précisément le temps nécessaire pour obtenir des résultats corrects.

Ci-contre on peut voir un bilan de la courbe d'apprentissage du **générateur**, parmi tous les essais qui ont été retenus et présentés dans ce mémoire (en excluant donc tous les échecs et erreurs). On retrouve dans l'ordre les essais des chiffres (1), formes (2), dispositions vides (3) et pleines (4).
Première remarque, que les essais soient en blanc sur noir ou noir sur blanc, les résultats sont globalement identiques. Le réel changement réside dans la lisibilité de ces résultats, car le blanc sur noir est plus proche de nos habitudes de déchiffrage. Par contre, j'observe des comportements assez similaires entre les chiffres et les formes, mais très différents entre les chiffres et les dispositions. Je peux ainsi avancer une hypothèse assez solide, selon laquelle l'aspect graphique des données paramétriques joue un rôle important. En effet, des données similaires (je pense aux formes qui étaient comme les chiffres, des éléments filaires complexes et variant beaucoup) vont être apprises globalement de la même manière. Un changement visuel radical, comme avec les dispositions simples et pleines, va complètement modifier le schéma d'apprentissage, mais fonctionner quand même.

Je constate ainsi une remarquable adaptabilité du **GAN**. En effet il est coutume de dire qu'un réseau de neurones codé pour apprendre des chiffres le fera très bien, mais aura beaucoup plus de mal pour d'autre données. Tout comme un bâtiment d'architecture excelle dans le domaine pour lequel l'architecte l'a conçu, son adaptation peut fonctionner mais colle

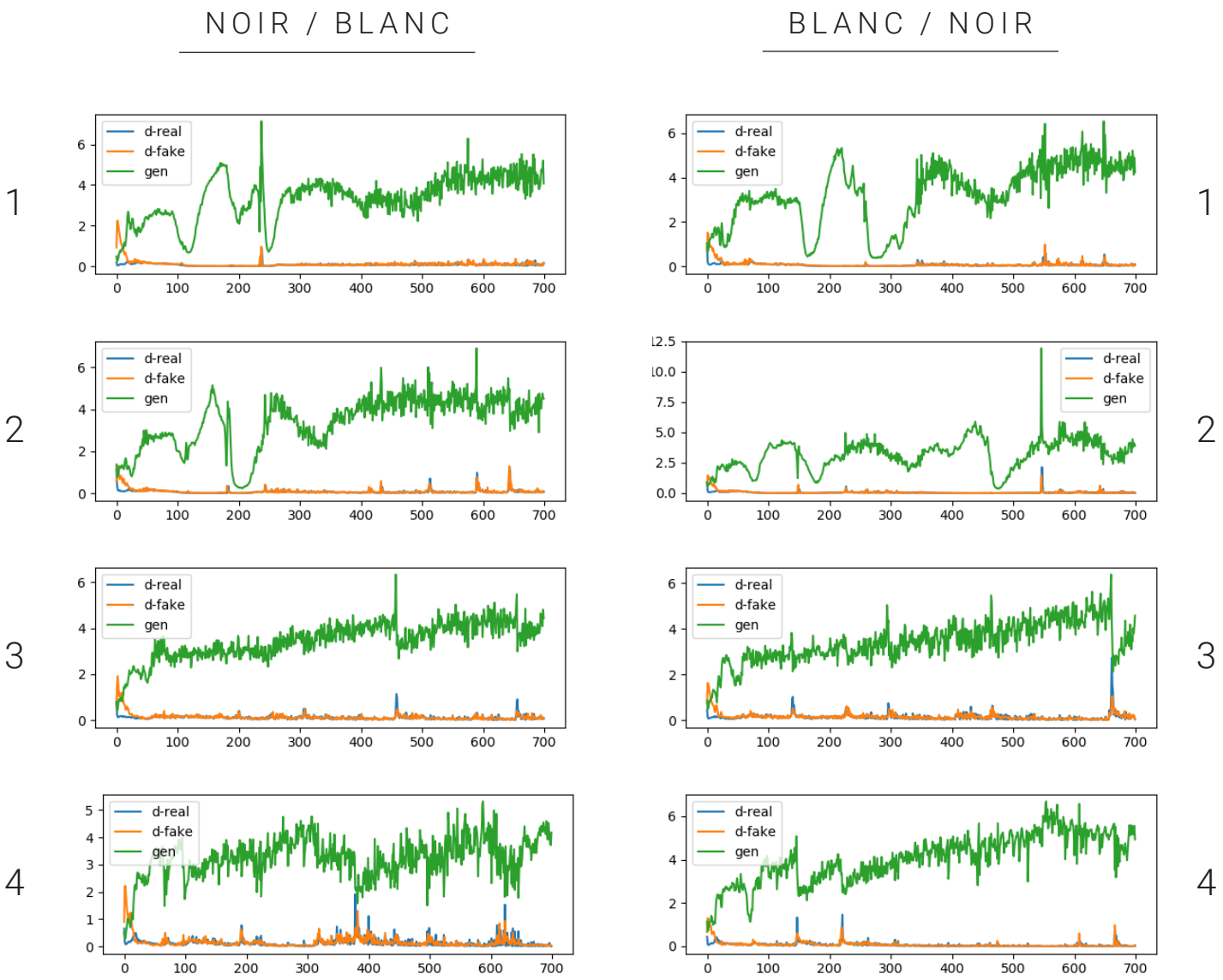



Fig. 5.1.1 - Statistiques du **générateur** pour chaque essai.

moins bien à la structure. Ce n'est pas pour rien que je parle « d'architecture informatique », car un logiciel a une fonction et un architecte.

Le **GAN** s'est donc révélé plutôt fonctionnel sur les nouvelles données qui lui ont été proposées, bien qu'il faille garder à l'esprit qu'elles restent relativement proches, avec une même résolution et les mêmes couleurs noir et blanc.

Je peux donc imaginer assez naïvement que si je sais plutôt bien adapter des réseaux de neurones à des données différentes, il sera possible dans un futur proche de s'accaparer des systèmes beaucoup plus puissants, développés pour les secteurs souvent en avance technologiquement, tels que l'industrie ou la santé.

Au-delà de la variation du temps d'apprentissage et bien sûr des résultats, le changement de base de données semble également jouer sur un paramètre plus complexe, que j'appellerai ici le « potentiel de création ».

Je me suis rendu compte de cet aspect au moment de chercher dans les résultats les fameuses « étrangetés ». Dans le cadre des chiffres, elles étaient abondantes et assez novatrices. Pour les formes, j'étais capable d'en trouver également, mais l'investigation était plus difficile. La plupart des résultats relevaient de la chimère ou de l'image fidèle. Enfin, pour les dispositions, les étrangetés ont été très rares, voir presque inexistantes si je considère une catégorie dédiée aux « images fidèles ratées ».

Une première hypothèse viserait à parler de complexité graphique. En effet, les chiffres sont plus complexes sur le papier que les formes, qui elles-mêmes le sont plus que les dispositions. Une plus grande complexité rimerait donc avec plus de contenu à hybrider, faire varier, et en tirer un fort potentiel créatif.

Cependant, cette hypothèse n'est valable qu'à un temps T de l'apprentissage. Les résultats que j'étudie ne sont pas seulement ceux de la dernière **itération**, mais sur l'ensemble de l'apprentissage. Là où les chiffres avaient besoin des 700 **itérations** pour révéler leur potentiel, les dispositions y arrivaient en seulement 70. Tout comme un moteur ne délivre pas sa puissance de la même façon selon le régime, chaque base de données affiche ses meilleurs résultats à des moments différents. Ainsi, si je sais regarder au bon moment dans l'apprentissage, chaque expérience sait proposer un potentiel de création convenable.

Ce potentiel n'est néanmoins pas limité à la capacité de créer des étrangetés, et prend parfois des formes inattendues. Par exemple, lors de la dernière expérience, j'ai été surpris par une « étrangeté » très réussie, la diagonale. Plus qu'une simple bizarrerie, il s'agissait vraiment d'une solution potentielle, à laquelle je n'avais même pas pensé.

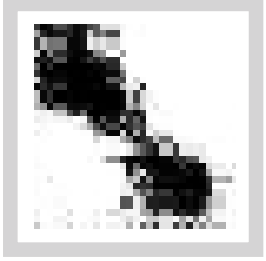


Fig. 5.1.2 - Résultat d'une disposition "étrangeté" intéressante.

Plus encore, cette dernière expérience sur les dispositions a fourni des résultats très intéressants en blanc sur noir. Lors de la présentation des résultats dans le chapitre précédent, je me suis surtout attardé sur les résultats en noir sur blanc, car leur lisibilité et leur proximité avec les images de référence permettait d'observer facilement les critères de la grille de validation. Les résultats en négatif, donc des dispositions pleines et blanches sur fond noir, mènent à une toute autre lecture. Comme on peut le voir ci-dessous, ce ne sont plus des rectangles dans une parcelle que j'interprète mais plutôt directement des empreintes de bâtiments, comme des typologies de pavillons. Ce résultat fait penser aux premières étapes de la thèse de Stanislas Chaillou, dont le travail a été grandement initiateur de ce mémoire. Cet aboutissement relève purement du hasard et n'a à aucun moment été intentionnel, et c'est pour cette raison que je n'ai pas insisté dessus dans le chapitre dédié à l'expérimentation.

Néanmoins, ces résultats aussi surprenants qu'intéressants prouvent encore une fois que le potentiel créatif se trouve souvent dans les erreurs. Celles-ci ne sont ce pas disséminées dans les autres résultats au sein de l'**espace latent**, mais sous la forme d'une légère variation de représentation de ces mêmes résultats.

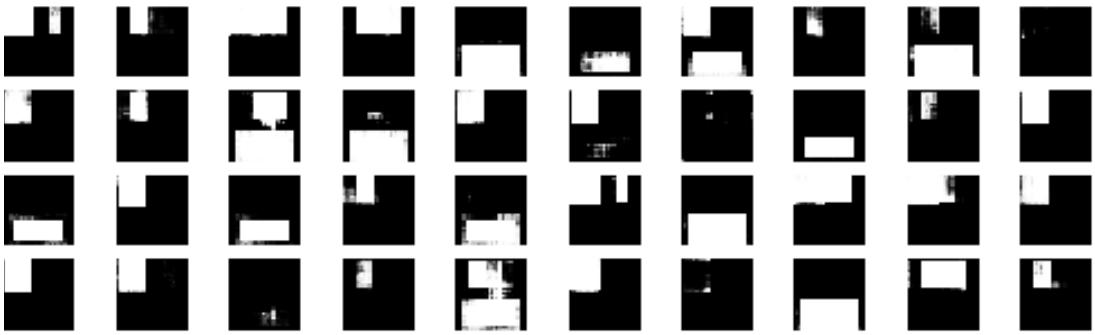


Fig. 5.1.3 - Fragment des résultats de l'expérience 3, essai des dispositions pleines en blanc sur noir.

L'expérience à la complexité graphique la plus simple aura donc été capable de proposer les solutions les plus riches, mais sous des formes plus inattendues que les expériences précédentes, qui n'en restent pas moins créatives.

L'enrichissement de la base de données paramétrique est donc réel, autant dans les résultats bruts, que dans ses variations inopinées.

## 5.2 Pistes d'amélioration

Comme tout travail, et particulièrement lorsqu'il s'agit d'informatique, de nombreuses améliorations sont envisageables, tant dans le détail que dans l'aspect général.
Ce chapitre concerne donc des pistes à petite échelle, que j'aurais aimé explorer si j'avais eu plus de temps. Ces améliorations ne sont donc pas indispensables à la validation de l'hypothèse du mémoire, mais ne sont pas suffisantes non plus pour en justifier un nouveau. Les pistes d'évolution à plus grande échelle seront abordées en revanche dans le chapitre suivant.

Le premier degré d'amélioration est donc dans la continuité directe de ce qui a été fait. Bien que je me sois arrêté à 3 expériences basées sur 3 bases de données paramétriques, on imagine facilement continuer le processus avec de nouveaux critères à tester avec un degré de complexité plus élevé. En effet, jusque-là les données générées paramétriquement n'étaient que des variations aléatoires d'une référence fixe. Une amélioration singulière consisterait à « optimiser » ces variations aléatoires selon un critère, appartenant au domaine architectural par exemple. Ainsi, au lieu de faire varier aléatoirement des empreintes de bâtiment sur une parcelle (l'expérience 3), ces variations pourraient prendre en compte l'orientation par rapport au soleil, la disposition selon le contexte urbain, les vues, les vis-à-vis, des éléments du **PLU**… Ce degré d'amélioration assez simple pourrait déjà constituer des éléments très intéressants à analyser, surtout après le passage du **GAN** : saurait-il retrouver les critères d'optimisation imposés ? Saurait-il croiser les critères par le biais des chimères ? Que signifierait alors les étrangetés ?

Une seconde phase d'amélioration viserait à se concentrer sur la capacité de calcul, autrement dit la puissance brute informatique déployée pour l'ensemble du processus. Cela permettrait de travailler avec plus de données, plus complexes et plus lourdes, et d'aboutir à des résultats plus nets et intéressants. L'amélioration est cependant plus du domaine de la quantité que de la qualité.

Plusieurs échelles d'intervention existent : de la simple amélioration de l'ordinateur (dans mon cas, c'était le **processeur ou CPU** qui montrait ses limites), à l'intégration de nouvelles technologies. Je pense particulièrement à **Nvidia**, entreprise américaine spécialisée dans les composants informatiques, qui développe une compatibilité entre ses composants et **Tensorflow**, et plus largement une intégration du **Machine Learning**. Ces nouvelles technologies permettraient d'exploiter au maximum le matériel utilisé, et à coût modéré.

Dans la même lancée, **Nvidia** développe également une série de micro-ordinateurs, nommés **Nvidia Jetson**, qui sont spécialisés dans le secteur de l'**IA**. Plus de souris, d'écran, de clavier, ni même de boîtier, l'ordinateur en est réduit à ce qui est nécessaire pour le calcul. Cela permet donc, en plus du gain de place, un énorme gain de moyens qui va permettre d'injecter beaucoup de puissance de calcul, et au bon endroit. Bien pratique pour entraîner un **GAN** ou miner de la crypto-monnaie, ces outils ne permettent cependant pas de lire ses mails.

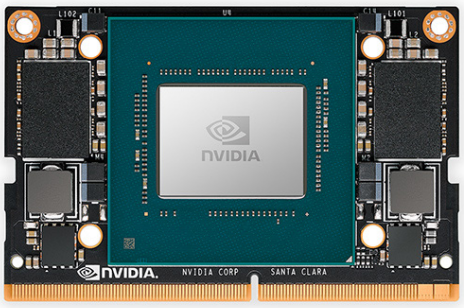



Fig. 5.2.1 - Exemple d'un des élément de la série **Nvidia** Jetson, le Jetson Xavier NX (source : Site Web Nvidia).

Enfin, pour sortir de l'échelle locale et atteindre le summum de la puissance de calcul, il est possible de s'orienter vers le **cloud**, et les fermes de calcul. Il s'agit plus ou moins du même principe qu'au paragraphe précédent, soit des micro-ordinateurs super performants, mais en très grande quantité et travaillant tous ensemble. De nombreuses entreprises proposent leurs services de location de « puissance de calcul », mais il faut y mettre le prix.

Une augmentation nette de la puissance de calcul permettrait dans un premier temps une amélioration des résultats obtenus, mais ouvrirait également la voix à ces nombreuses évolutions du processus et donc du mémoire.

## 5.3 Pistes d'évolution

Voyons maintenant quelles suites peut-on faire à ce travail, tant à court qu'à long terme.

Une des principales limites de l'expérience résidait dans la résolution des images. En effet, bien que le potentiel créatif se trouvait surtout dans les images les moins nettes, elles étaient tout de même très simples, et contraintes par le faible nombre de pixels. Avec une plus grande puissance de calcul théorique, ou des **algorithmes** mieux optimisés, il serait possible de travailler sur des sujets plus complexes et plus détaillés, et par conséquent plus riches à étudier. L'idée de plans d'architecture est évidente, mais cela pourrait s'étendre aux photos d'architecture, voir même aux représentations tridimensionnelles.

Face à ces nouveaux sujets d'étude beaucoup plus complexes, l'architecture des réseaux de neurones est également à traiter. J'ai ici utilisé un **GAN** en particulier, mais sa structure est entièrement questionnable. Même si les réseaux de neurones antagonistes génératifs sont très en vogue actuellement dans le secteur de la génération de données, une autre piste réside dans les réseaux de neurones auto-organisateurs. Le sujet est encore tout frais dans les papiers de recherche[1], mais suggère une structure du réseau de neurone variable, qui s'enrichirait et s'adapterait à chaque **itération** et donc excellerait en efficacité.

En complément de réseaux de neurones plus complexes et d'une puissance de calcul accrue, une variable manque à l'appel et constitue un vaste champ d'évolution : les bases de données. En effet peu importe le domaine ou le type de données à générer, une quantité faramineuse de données initiales est nécessaire. Ce mémoire présentait une façon d'y remédier en générant soit même ces données de façon paramétrique, mais d'autres alternatives existent. Certaines sources sont capables de fournir des données de façon presque infinie (par exemple, la toiture de chaque bâtiment sur Terre se trouve sur Google Maps, et les cartes sont mises à jour chaque semaine).

---

[1] **Michaël Aupetit,** « Approximation de variétés par réseaux de neurones auto-organisés », Thèse de doctorat en Génie industriel, Grenoble INPG.

Encore à titre d'exemple, les images révélées par l'entreprise californienne Capella Space semblent être très prometteuses. Cette firme privée spécialisée dans les images satellites a exploité une technologie jusque-là encore réservée à l'industrie militaire : le **SAR**, ou **radar à synthèse d'ouverture**. Sans entrer dans les détails, cela permet d'obtenir directement depuis l'espace (les images Google Maps sont réalisées depuis des avions) des vues satellites d'une qualité exceptionnelle, de jour comme de nuit, avec ou sans nuages, et pouvant « voir » au travers des bâtiments. Comme on peut le voir sur l'image ci-dessous, aperçu proposé par l'entreprise en fin d'année 2020, il est possible de distinguer les différents bâtiments de cette zone industrielle à Singapour, mais également leur structure grossière.

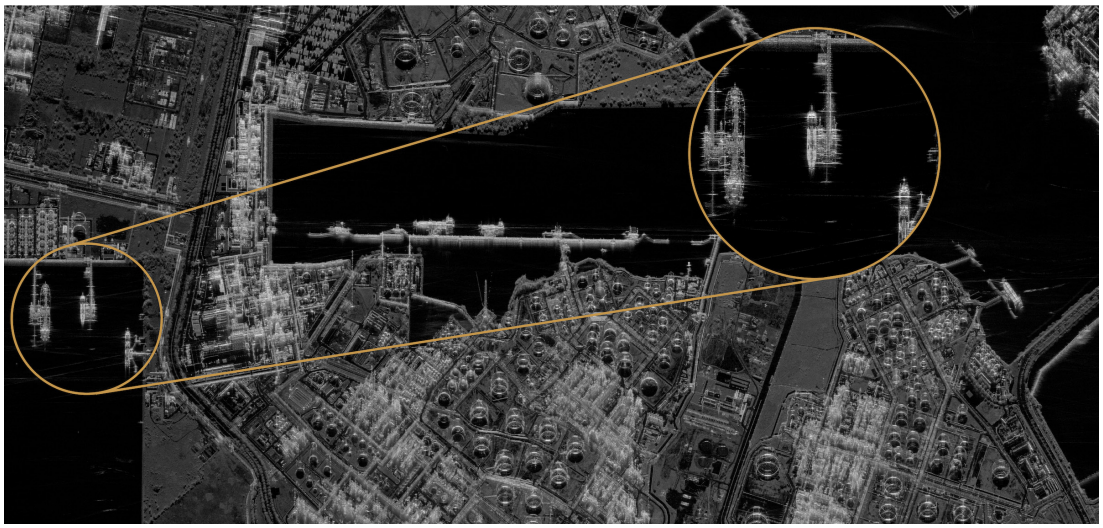


Fig. 5.3.1 - Image satellite de l'entreprise Capella Space, vue de la zone portuaire industrielle de Singapour, 2020 (source : Capella Space).

Quel en est l'apport pour les **GAN** et les pistes d'évolution ? Il s'agit là de données à but commercial et donc exploitables, et ce à l'échelle de la planète. La structure de chaque bâtiment existant est identifiable en blanc sur noir, ce qui constitue une formidable base de données pour un réseau de neurones. Même si la qualité est encore un peu légère et les bâtiments vagues, la technologie est là et quelques années suffiront à rendre ces images aussi nettes que des plans masse.
Je trouve intéressant ce renversement de position : là où les recherches dont celle de Stanislas Chaillou se concentrent sur des plans existants faits par l'Homme, l'idée est ici de chercher après la réalisation et d'analyser l'existant.

Autre piste intéressante quant à l'évolution des problématiques de ce mémoire, la piste de l'**upscaling**. Il s'agit d'un procédé très courant dans les nouvelles télévisions et consoles de jeux, qui consiste à augmenter la qualité d'une image artificiellement. Comme représenté sur le schéma ci-dessous, l'idée est de partir d'une image de faible résolution, puis de multiplier la quantité de pixels tout en conservant leur pertinence par rapport à l'image initiale. Ce processus peut avoir recours à de nombreux types d'**algorithmes**, mais les plus aboutis aujourd'hui restent ceux basés sur le **Machine Learning**. L'apprentissage sur des images aussi nombreuses que variées leur permettra ainsi de réaliser l'**upscaling** de n'importe quelle image initiale.

L'intérêt de ce procédé est simple : faire moins d'efforts dans la création des données, mais procéder à un traitement de celles-ci en dernière étape. Pour une console de jeu cela permet d'afficher une image en résolution 4K (3840x2160 pixels), alors qu'elle a été calculée en Haute Définition (1920x1080 pixels).

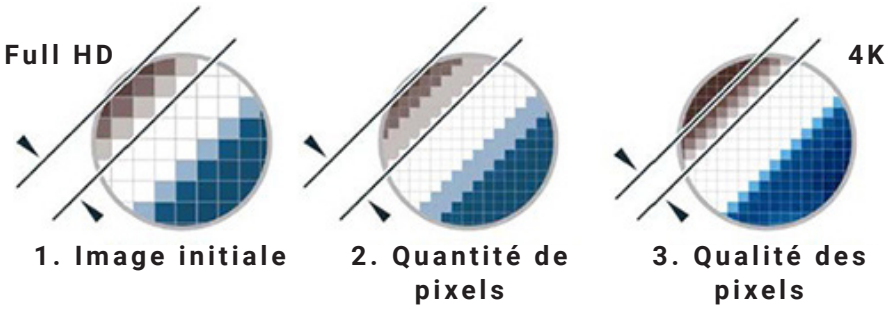



Fig. 5.3.2 - Schéma du principe de l'**upscaling** d'une image (source : Philips).

Dans le cadre du processus utilisé dans ce mémoire, l'objectif est le même : compléter le travail du réseau de neurones par un autre **algorithme**, qui va apporter une meilleure lisibilité des résultats.

Enfin, toujours dans cette optique de traitement des résultats, il est possible d'imaginer un système où j'appliquerai un autre réseau de neurones sur les résultats du premier. Le plus pertinent serait un **algorithme** de classification, qui irait de lui-même chercher les résultats les plus pertinents, et regrouper ce qui tient de la chimère, du fidèle ou de l'étrangeté, accompagné des statistiques de ressemblances. Il existe de nombreuses manières de s'y prendre, et là Jason Bronwlee est d'une très grande aide avec des pistes pour chaque méthode de classement[2]. Ce sont d'ailleurs ces mêmes méthodes de classement qui sont utilisées pour réaliser les célèbres images de « transition » d'une classe à une autre, en l'occurrence pour le jeu de données **MNIST** (voir image ci-dessous) le passage d'un chiffre à un autre avec tous les intermédiaires.

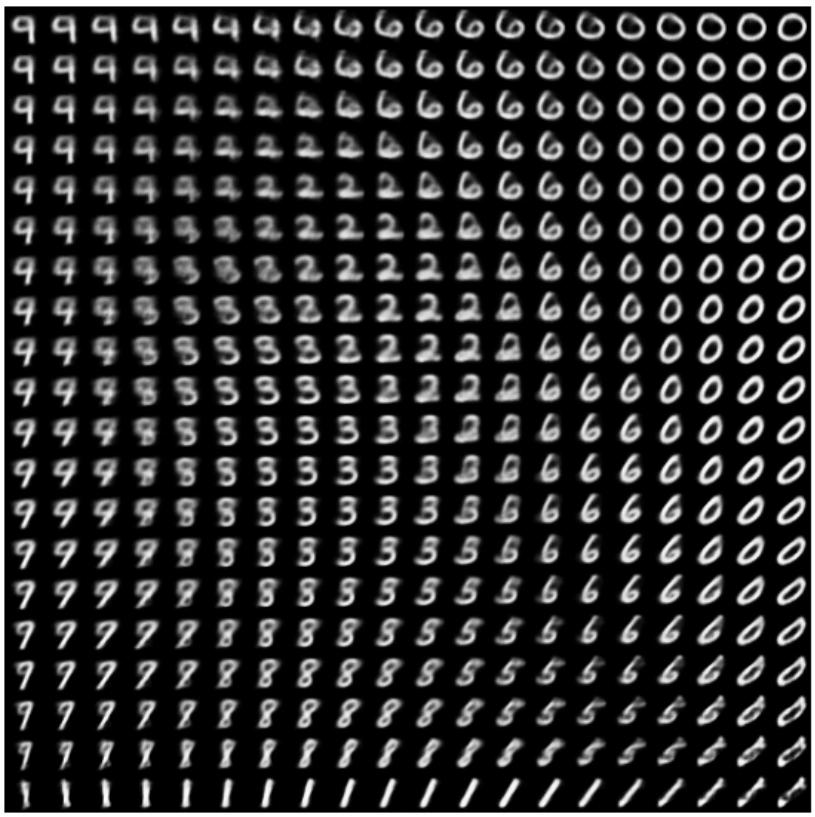


Fig. 5.3.3 - Résultat obtenu après le travail d'un réseau de neurones de classification (Autoencodeur Variationel Convolutif) sur une base de données **MNIST** (source : site Web TensorFlow).

[2] **Jason Bronwlee**, « 4 Types of Classification Tasks in Machine Learning ", site Web Machine Learning Mastery, section Python Machine Learning, Avril 2020, consulté le 26/12/20. Disponible sur : https://machinelearningmastery.com/types-of-classification-in-machine-learning.

Cette évolution tient plus de la présentation des résultats que des résultats en eux-mêmes, mais permet vraiment de se projeter dans un système d'aide à la création, où l'œil peut glisser d'un résultat à un autre en y repérant les déclinaisons qui l'intéresse.

Ces **algorithmes** de classement pourraient également être branchés directement aux coordonnées des résultats dans l'**espace latent**, plutôt qu'aux images produites. En effet, comme j'avais pu l'observer au fil de mes expériences, les données produites par un **GAN** ont une logique de placement dans l'**espace latent**. Cette logique est cependant trop complexe à imaginer, car l'**espace latent** est souvent en plus de 3 dimensions. Les **algorithmes** de classement pourraient donc faire émerger cette logique, et peut-être en trouver de nouvelles dans les choix d'images que nous faisons. Peut-être que nos résultats préférés produis par le **GAN** coïncident avec des logiques particulières, ce qui permettrait à l'**algorithme** de classement de proposer de manière personnalisée des résultats sur la base de ceux que l'on a aimé, comme le font déjà de nombreux programmes sur Internet.

Ces logiques ainsi dégagées seraient alors l'inverse même d'une référence architecturale. Lorsque l'on s'approprie une référence, on extrait d'une œuvre existante des caractéristiques qui nous intéressent, que l'on souhaite exploiter. Avec l'approche du paragraphe précédent, on extrait des caractéristiques intéressantes directement de la conception même, de nos propres choix vis-à-vis des propositions de la machine. Cette mécanique complexe permettrait cependant d'exploiter au mieux les résultats produits, et de fournir un enrichissement aussi qualitatif qu'exploitable de la base de données initiale.

L'ensemble des améliorations et évolutions proposées précédemment peuvent être rassemblées au sein d'un même schéma, centré sur les résultats et dont les extrémités matérialisent les trois pôles essentiels d'un réseau de neurones :

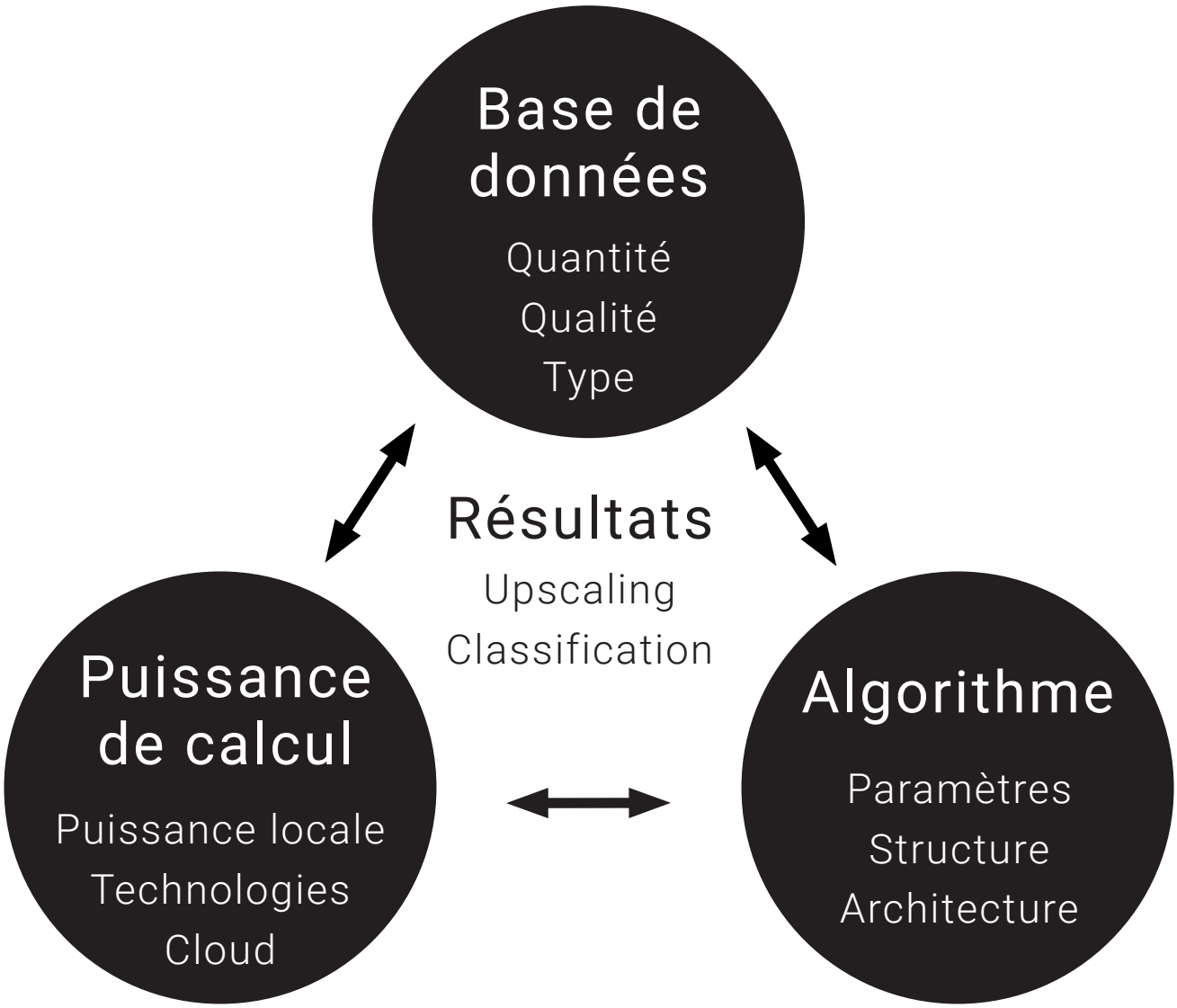



Fig. 5.3.4 - Schéma récapitulatif des pôles d'amélioration et d'évolution du mémoire.

# 6

# CONCLUSION

Dans un contexte de déploiement de technologies de plus en plus lourdes dans le milieu architectural, j'ai décidé d'aborder l'une des plus énigmatiques et magiques d'entre elles, le **Machine Learning**. Au travers de ses aspects les moins froids, les erreurs et décalages entraînés par l'apprentissage, j'ai questionné ce qu'il en était d'un enrichissement potentiel de données existantes.

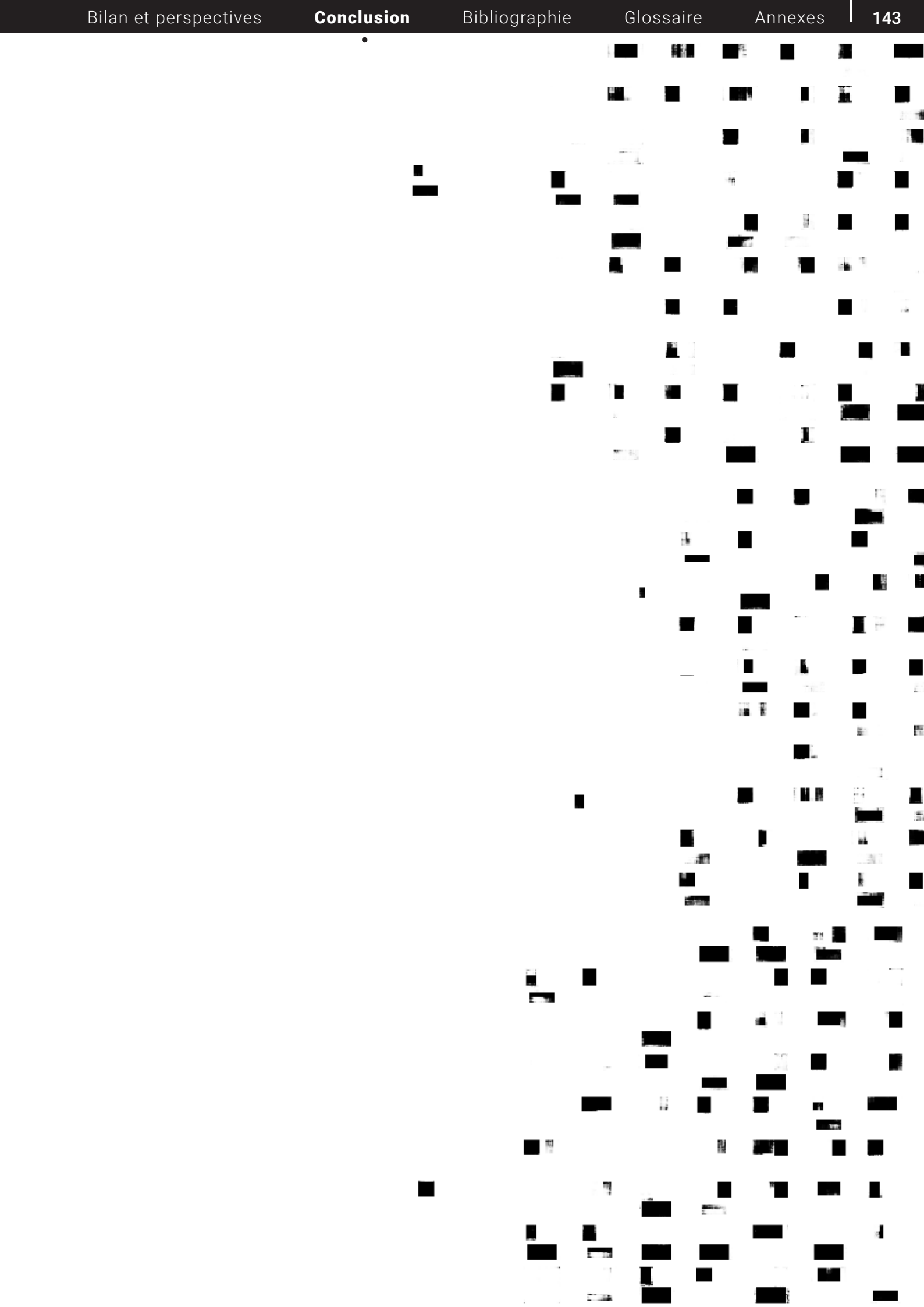
L'expérimentation a permis de révéler l'efficacité des réseaux de neurones antagonistes génératifs et ce à plusieurs niveaux. Par le biais de plusieurs essais, et des bases de données d'images paramétriques différentes, l'**algorithme** a su produire des images à son tour. Dans chaque essai j'ai retrouvé 3 éléments clés témoins d'un enrichissement, à savoir des images fidèles aux images de la base de données initiale, des chimères entre images et des « bizarreries ». Ces dernières, pouvant être qualifiés pragmatiquement « d'erreurs », permettent cependant de sortir de la sobriété des images initiales, et servir de base de réflexion pour un concepteur aguerri. Ces « étrangetés » ne résultent pas du hasard, mais d'un processus d'apprentissage acharné (et machinique). Elles ont donc une cohérence, une essence, une source de richesse.

Ce travail de mémoire permet donc de proposer un processus, afin d'enrichir un espace de solution paramétrique grâce aux réseaux antagonistes génératifs, et les alternatives sont nombreuses. Le système présenté est une base très simple, ouverte à une infinité d'améliorations relevant tant des données, de la manière de les exploiter, et de la façon de se servir des résultats.

La révolution du **Machine Learning** ne se fera pas sans bruit, et s'y préparer est la meilleure façon de conserver la richesse du métier d'architecte.

« *Une pinte de sueur économise un gallon de sang.* »[1]

[1] **George S. Patton,** Général de l'Armée de terre américaine.



# 7
# BIBLIOGRAPHIE

- Travaux de recherche :

**Michaël Aupetit**, « Approximation de variétés par réseaux de neurones auto-organisés », Thèse de doctorat en Génie industriel, Grenoble INPG.

**Ehsan Bazafkan**, « Assessment of Usability and Usefulness of New Building Performance Simulation Tools in the Architectural Design Process », Thèse de doctorat à l'Université technique de Vienne, Architektur + raumplanung, 2017.

**Stanislas Chaillou**, « AI + Architecture », Thèse de doctorant à l'Université de Harward, Graduate School of Design, Cambridge, 2019.

**Clément Chatelain**, « Extraction de séquences numériques dans des documents manuscrits quelconques », Thèse de doctorat à l'Université de Rouen, UFR de sciences et techniques, discipline informatique, 2006.

**Manon Dampfhoffer**, « Synthèse Initiative Deep Learning du MAP », 2019.

**Sophie Oros**, « L'apprentissage machine au service de la conception architecturale ? Une application : extraire des informations à partir de plans », Mémoire de master dans le cadre du séminaire Activités et Instrumentation de la Conception, ENSA de Paris la Villette, 2019.

- Ouvrages et conférences :

**Aurélien Géron**, « Deep Learning avec TensorFlow, mise en oeuvre et cas concrets », Dunod, 2017.

**O. Guilhem, R. Gelin**, « L'IA et nous », Editions Le Pommier, 2019.

**Ecole Normale Supérieure**, Département de physique, Guide « La Démarche scientifique ».

**Ajit Joakar**, « Computer Science for Schools from Concepts to Code », CreateSpace Independent Publishing Platform, 2016.

**Yann LeCun**, « L'apprentissage profond », conférences au Collège de France, 2015-2016.

**Stéphane Lutard**, « Architecture et intelligence artificielle », Cahiers de la profession n°70, 2e trimestre 2020.

**John Peterson**, « How to use Knot Vectors », Albert Technical Memo, Juin 1990.

**Christophe Pradal**, « Courbes et Surfaces NURBS », Cours Institut national de recherche en sciences et technologies du numérique.

**Caroline Quentin**, « The World's Most Extraordinary Homes, Sous-terre », Saison 1 Episode 4, BBC Two, 2017.

**Gérard Swinnen**, « Apprendre à programmer avec Python 3 », Eyrolles, 2017.

**Arturo Tedeshi**, « AAD Algorithms-Aided Design : Parametrix Streategies using Grasshopper », Le Penseur, 2014.

- Ressources en ligne :

**Afnan Amin Ali**, « GAN (Generative Adversarial Network) », Site Medium, The Startup, 17 Mai 2020, consulté le 05/02/20. Disponible sur : https://medium.com/swlh/gan-generative-adversarial-network-3706ebfef77e.

**Will Badr**, « Auto-Encoder: What Is It? And What Is It Used For? (Part 1) », Site Web towards data science, 22 Avril 2019, consulté le 12/08/20. Disponible sur : https://towardsdatascience.com/auto-encoder-what-is-it-and-what-is-it-used-for-part-1-3e5c6f017726.

**Aurélie De Boissieu**, François Guéna « Grasshopper et la programmation sur Rhinocéros 4 : une introduction », DNArchi, Design for Numerical Architecture, consulté le 28/08/20. Disponible sur : http://dnarchi.fr/pedagogies/grasshopper-et-la-programmation-sur-rhinoceros-4-une-introduction/#:~:text=Grasshopper%20(GH)%20est%20un%20plugin,%C3%A0%20de%20la%20programmation%20visuelle.

**Jason Bronwlee**, « How to Develop a GAN for Generating MNIST Handwritten Digits », Site Web Machine Learning Mastery, 14 Juin 2019, consulté le 22/12/20. Disponible sur : https://machinelearningmastery.com/impressive-applications-of-generative-adversarial-networks/.

**Jason Bronwlee**, « 18 Impressive Applications of Generative Adversarial Networks (GANs) », Site Web Machine Learning Mastery, 18 Juin 2019, consulté le 19/06/20. Disponible sur : https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-an-mnist-handwritten-digits-from-scratch-in-keras.

**Jason Bronwlee**, « 4 Types of Classification Tasks in Machine Learning », site Web Machine Learning Mastery, section Python Machine Learning, Avril 2020, consulté le 26/12/20. Disponible sur : https://machinelearningmastery.com/types-of-classification-in-machine-learning.

**Jason Bronwlee**, « How to Manually Scale Image Pixel Data for Deep Learning », site Web Machine Learning Mastery, section Python Machine Learning, 5 Juillet 2019, consulté le 11/09/20. Disponible sur : https://machinelearningmastery.com/how-to-manually-scale-image-pixel-data-for-deep-learning.

**Jason Bronwlee**, « How to Identify Unstable Models When Training Generative Adversarial Networks », site Web Machine Learning Mastery, section Generative Adversarial Networks, 17 Août 2020, consulté le 11/10/20. Disponible sur : https://machinelearningmastery.com/practical-guide-to-gan-failure-modes.

**Capella Space**, « Capella Unveils World's Highest Resolution Commercial SAR Imagery », Site Web de Capella Space, 16 Décembre 2020, consulté le 18/12/20. Disponible sur : https://www.capellaspace.com/capella-unveils-worlds-highest-resolution-commercial-sar-imagery/.

**Stanislas Chaillou**, « L'IA en Architecture, Séquence d'une Alliance », Article du site Medium, 5 Octobre 2019, consulté le 03/03/2020. Disponible sur : https://medium.com/@sfjchaillou/lia-en-architecture-c805651d7f42.

**Stanislas Chaillou**, « IA & Architecture, Une Perspective Expérimentale », Article du site Medium, 26 Février 2019, consulté le 30/10/2020. Disponible sur : https://medium.com/@sfjchaillou/ia-architecture-475d3282f6e6.

**Gaël G**, « 3 Algorithmes de Deep Learning expliqués en Langage Humain », Site Web DATAKEEN, 25 Février 2018, consulté le 20/12/20. Disponible sur : https://datakeen.co/3-deep-learning-architectures-explained-in-human-language/.

**Gregory S. Kielan**, « JPG-PNG-to-MNIST-NN-Format », Site Web GitHub, consulté le 15/08/20. Disponible sur : https://github.com/gskielian/JPG-PNG-to-MNIST-NN-Format.

**Yann LeCun, Corinna Cortes, Christopher Burges**, « The MNIST database of handwritten digits », consulté le 13/03/20. Disponible sur : http://yann.lecun.com/exdb/mnist.

**Nvidia**, « Systèmes embedded pour machines autonomes de nouvelle génération », Site Web Nvidia, consulté le 25/08/20. Disponible sur : https://www.nvidia.com/fr-fr/autonomous-machines/embedded-systems/.

**Gavril Ognjanovski**, « Everything you need to know about Neural Networks and Backpropagation — Machine Learning Easy and Fun », Site Web towards data science, 14 Janvier 2019, consulté le 12/08/20. Disponible sur : https://towardsdatascience.com/everything-you-need-to-know-about-neural-networks-and-backpropagation-machine-learning-made-easy-e5285bc2be3a.

**Madhu Ramiah**, « How I increased the accuracy of MNIST prediction from 84% to 99.41% », Article du site Medium, 18 Juin 2019, consulté le 29/12/2020. Disponible sur : https://medium.com/@madhuramiah/how-i-increased-the-accuracy-of-mnist-prediction-from-84-to-99-41-63ebd90cc8a0.

**Léa S**, « Quelle différence entre Machine Learning et Deep Learning ? », Site Web DataScientest, Section Deep Learning, Juillet 2020, consulté le 28/12/20. Disponible sur : https://datascientest.com/quelle-difference-entre-le-machine-learning-et-deep-learning.

**Thilo Spinner, Jonas Körner, Jochen Görtler, Olivier Deussen**, « Towards an Interpretable Latent Space - An Intuitive Comparison of Autoencoders with Variational Autoencoders », Université de Konstanz, 2018, consulté le 25/12/20. Disponible sur : https://thilospinner.com/towards-an-interpretable-latent-space.

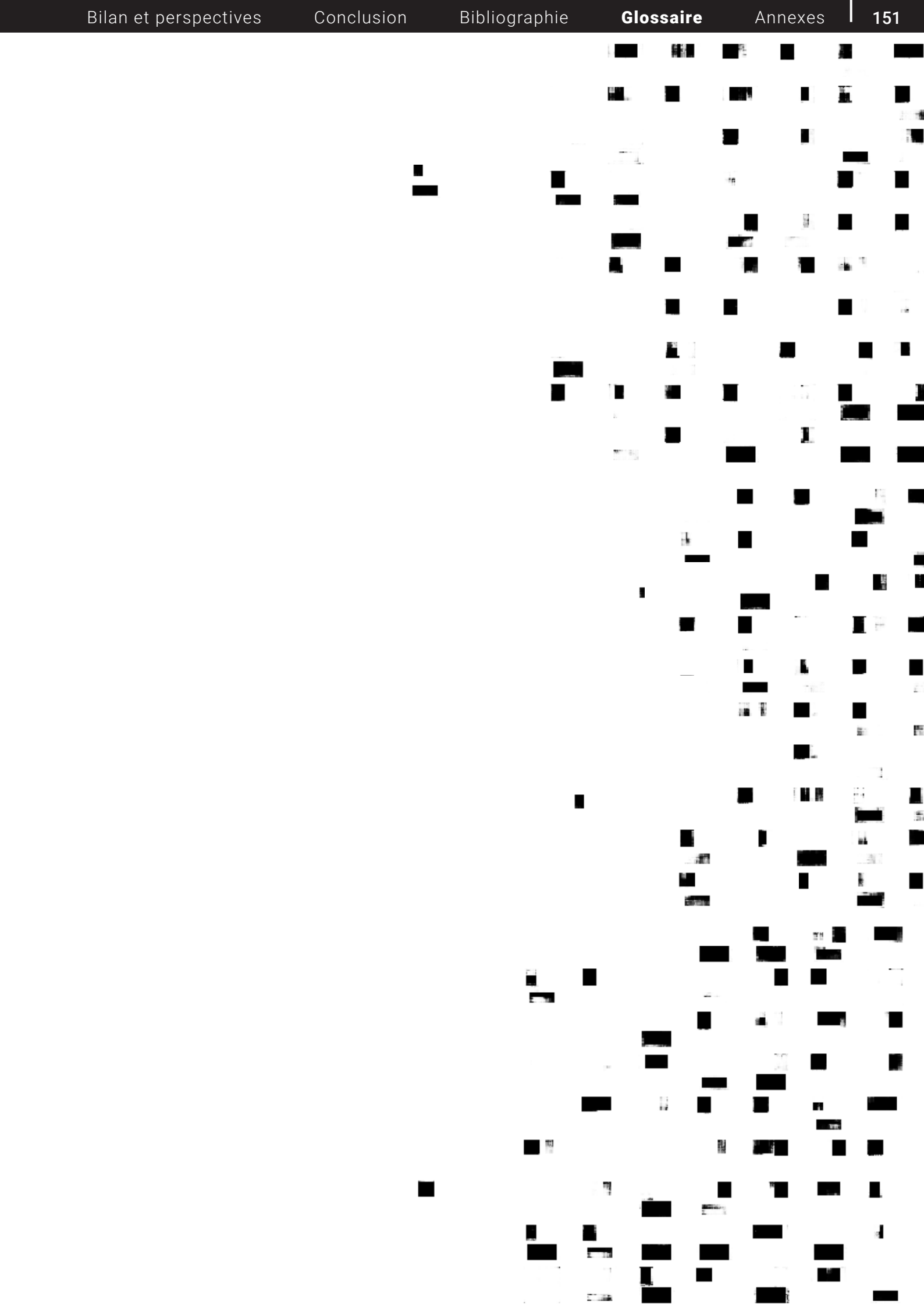
- Guides et tutoriels :

**TensorFlow :** https://www.tensorflow.org/tutorials.

**GitHub :** https://github.com.

**Python :** https://www.python.org.

**Machine Learning Mastery (Jason Bronwlee) :** https://machinelearningmastery.com.



# 8

# GLOSSAIRE

## 8.1 Termes courants

**Algorithme :** en informatique, c'est une suite finie de calculs permettant d'arriver à un résultat, souvent pour résoudre un problème.

**Anaconda Navigator :** interface permettant de créer, gérer et supprimer des environnements virtuels et ce qui y est installé (langages de programmation, **libraries** et leurs versions).

**Auto-encodeur :** architecture de réseau de neurones qui va encoder puis décoder les données. Elle est principalement utilisée dans la détection d'anomalie ou le débruitage des photos.

**BIM :** Building Information Modeling, soit un ensemble de méthodes de travail autour d'une maquette numérique 3D paramétrée permettant une plus grande collaboration des acteurs.

**Blob :** en architecture, le blob caractérise des formes molles, organiques. En géométrie 2D cela peut s'interpréter comme une sorte de tâche.

**Cloud :** service proposé par un fournisseur permettant d'accéder à des outils informatiques (stockage de données, calcul, logiciels…) à distance par le biais d'Internet.

**CNN :** Convolutional Neural Network, soit **réseau de neurones convolutifs** en français.

**Code couleur CMYK :** Cyan Magenta Yellow Black, code couleur utilisé pour le traitement des images et l'imprimerie. Equivalent du CMJN en français (Cyan Magenta Jaune Noir). L'intersection du Cyan Magenta et Jaune donne un noir, et non un blanc comme le système RGB plus courant (Red Green Blue).

**Computation :** anglicisme (mais qui tend à un emprunt définitif) pour calcul informatique.

**DAO :** Dessin Assisté par Ordinateur.

**Dataset :** terme anglais pour base de données.

**Démarche scientifique :** méthode qui guide la production de connaissances scientifiques et permet d'améliorer la compréhension du monde.

**Deep learning :** apprentissage profond en français, sous domaine du **Machine Learning** caractérisé par des réseaux de neurones organisés en couches permettant un apprentissage plus complexe et plus « intelligent ».

**Descente de gradient : algorithme** d'optimisation cherchant à minimiser une **fonction de coût**.

**Discriminateur :** dans un **GAN**, section qui "juge" les résultats produits.

**Environnement virtuel :** environnement comprenant une certaine version de **Python** et de ses **libraries**, permettant d'éviter les conflits de versions.

**Espace latent :** espace multidimensionnel comprenant toutes les solutions produites par un réseau de neurones, ou système de production neuronal de données.

**Fonction de coût :** équation mathématique à minimiser (**Machine Learning**).

**Grasshopper (GH) : plugin** de **Rhinocéros 3D** permettant de réaliser des modèles paramétriques en programmation visuelle.

**GAN :** Generative Adversarial Networks, **réseau de neurones antagonistes génératifs** en français.

**Générateur :** dans un **GAN**, section qui crée des résultats selon les notes attribuées par le discriminateur.

**GitHub :** plateforme web dédiée au partage de code en divers langages de programmation, de projets et de programmes plus complexes.

**Gradient :** variation d'une grandeur dans l'espace.

**Graine :** « seed » en anglais, valeur à l'origine de l'aléatoire d'une fonction ou d'un composant.

**Invite de commande Windows :** interface utilisateur permettant de dialoguer en lignes de commandes avec le système d'exploitation Windows.

**IA :** abréviation pour Intelligence Artificielle.

**Itérations :** répétitions successives d'un processus.

**Jupyter Notebook :** application web servant d'interface de programmation.

**Keras :** ensemble de **libraries** pour le **Machine Learning** compatible avec **Python**.

**Library (bibliothèque) :** ensemble de fonctions permettant d'implémenter de nouvelles capacités au langage de programmation.

**Machine Learning (ML) :** apprentissage automatique en français, c'est une technologie d'intelligence artificielle permettant aux ordinateurs « d'apprendre » et ainsi de répondre à des situations auxquelles ils n'ont pas été spécifiquement programmés.

**Matplotlib : library Python** spécialisée dans la création de graphiques 2D et 3D de données mathématiques.

**MLxtend :** ensemble de **libraries** pour le **Machine Learning** compatible avec **Python**.

**MNIST (base de données) :** base de données regroupant 60 000 images de chiffres de 0 à 9 écrits à la main, crée par Yann Lecun. Les images sont en noir et blanc, normalisées, centrées et de 28 pixels de côtés.

**NumPy : library Python** destinée à la création de matrices et tableaux.

**NURBS curve :** acronyme anglais pour Non-Uniform Rational Basis Spline, soit une courbe passant par des points de contrôle, avec une certaine force que l'on va définir (le poids).

**Nvidia :** entreprise américaine spécialisée dans la conception de composants informatiques.

**Nvidia Jetson :** série de micro-ordinateurs développés par **Nvidia** à basse consommation énergétique permettant d'accueillir des fonctionnalités liées au **Machine Learning**.

**PLU :** Plan Local d'Urbanisme, document regroupant les règles et documents graphiques quant à l'occupation du sol et la planification de l'urbanisme.

**Plugin :** outil permettant d'ajouter de nouvelles fonctionnalités en marge d'un logiciel auquel il est rattaché.

**Processeur (CPU) :** qualifié comme le cerveau de l'ordinateur, il réalise les calculs et coordonne les différents composants de la machine (mémoire vive, carte graphique, disque dur…). CPU est l'abréviation de Central Processing Unit.

**Python :** langage de programmation multiplateforme.

**Radar à synthèse d'ouverture (SAR) :** radar satellite spécialisé dans l'imagerie exploitant son déplacement afin de fournir des résultats plus fins.

**Réseau de neurones antagonistes génératifs (GAN) :** Generative Adversarial Networks en anglais, architecture de réseau de neurones permettant de générer des données après avoir appris d'autres données.

**Réseau de neurones convolutifs (CNN) :** Convolutional Neural Network en anglais, architecture de réseau de neurones principalement utilisée dans la reconnaissance vocale, la traduction automatique et l'analyse de séquences ADN.

**Réseau de neurones récurrents (RNN) :** Recurrent Neural Network en anglais, architecture de réseau de neurones principalement utilisée dans la reconnaissance d'image et de vidéos.

**Rhinocéros 3D – Rhino :** logiciel de Conception Assistée par Ordinateur (CAO) servant à la modélisation de formes et objets complexes.

**RNN :** Recurrent Neural Network, soit **réseau de neurones récurrents** en français.

**SAR :** Synthetic Aperture Radar, **radar à synthèse d'ouverture** en français.

**TensorFlow :** ensemble de **libraries** pour le **Machine Learning** compatible avec **Python**.

**Ubyte :** abréviation pour « .idx3-ubyte », une extension de fichier contenant une base de données d'images, classées et normalisées aux mêmes dimensions.

**Upscaling :** augmentation de la résolution d'une image ou d'une vidéo.

## 8.2 Termes liés à la programmation

Les termes présentés ici ne sont pas tous cités dans le mémoire, mais j'ai dû les définir de mon côté afin de pouvoir les comprendre et/ou les utiliser à mon tour. On trouve donc ici des termes appartenant aux commandes **Python**, mais aussi **Numpy**, **Tensorflow** et quelques éléments du terminal Windows.

**Numpy :**

**Array :** tableau multidimensionnel.

**np.random.rand( , ) :** génère un **array** de dimensions définies avec des valeurs aléatoires entre 0 et 1.

**np.random.randn( , ) :** génère un **array** de dimensions définies avec des valeurs aléatoires de moyenne 0 et de variance 1.

**np.random.seed( ) :** cale une valeur fixe au random (aléatoire).

**.T :** transpose l'**array**.

**.dot :** produit matriciel.

**X.mean :** moyenne de X.

**X.std :** Écart type de X.

**np.c_[ ] :** concatène (rassemble) 2 matrices.

**np.ceil :** plafonne une valeur à l'entier supérieur.

**np.random.randint :** renvoie des entiers.

**Python :**

**print( ) :** affiche la variable.

**“”” “”” :** ignore les symboles et mémorise le formatage d’un texte.

**[ ] :** détermine un caractère d’une chaîne.

**len( ) :** détermine la longueur d’une chaîne.

**int( ) :** convertit une chaîne en nombre entier.

**float( ) :** convertit une chaîne en nombre réel.

**import … as … :** importe une **library** (bibliothèque).

**== :** teste une égalité.

**!= :** teste une différence.

**del( ) :** supprime un élément d’une liste.

**.append( ) :** ajoute un élément en fin de liste.

**end = “ “ :** dans **print()** remplace le saut à la ligne par “ “.

**sep = “ “ :** dans **print()** remplace l’espace de séparation par “ “.

**input( ) :** l’utilisateur doit entrer des caractères.

**from math import * :** importe toutes les fonctions (*) du module math.

**for … in range ( ) :** répète des instructions, la variable s’incrémente de 1 jusqu’à la fin de la plage entrée.

**eval( ) :** évalue le contenu d’une chaîne comme une expression.

**list( ) :** convertit une séquence de caractères en liste.

**return :** retourne une valeur depuis une fonction définie.

**for … in range (start,end,step) :** établit un compteur.

**" {} {} ".format ( ) :** remplace les {} par les valeurs dans les ( ).

**Tensorflow :**

**Batch :** lot de données.

**Epoch :** époque ou période, que l'on peut qualifier dans les **GAN** par un regroupement d'**itérations** (**itération** d'**itérations**).

**tf.Variable( , name=" ") :** met en place une variable pour un graphe.

**tf.constant( , name=" ") :** met en place une constante pour un graphe.

**.shape :** définit les grandeurs d'un tableau.

**dtype= :** définit le type de donnée.

**tf.transpose( ) :** transpose une matrice.

**tf.matmul( ) :** multiplication matricielle.

**tf.matrix_inverse( ) :** inverse une matrice.

**with tf.Session() :** crée une session de calcul **Tensorflow**.

**with tf.Session() as sess :** crée une session de calcul **Tensorflow**.

**sess.run() :** effectue un calcul dans la session.

**tf.placeholder( ) :** produit une donnée précisée à l'exécution.

**feed_dict={A: } :** précise la valeur de la variable.

**saver = tf.train.saver() :** crée un nœud de sauvegarde (phase construction).

**save_path = saver.save (sess, "chemin") :** sauvegarde les données.

**saver.restore(sess, "chemin") :** restaure des variables enregistrées.

**tf.summary.scalar(name,data) :** crée un fichier binaire true/false (vrai/faux).

**tf.summary.FileWriter :** crée des fichiers de logs d'évènements.

**.data :** extrait les valeurs d'un jeu de données.

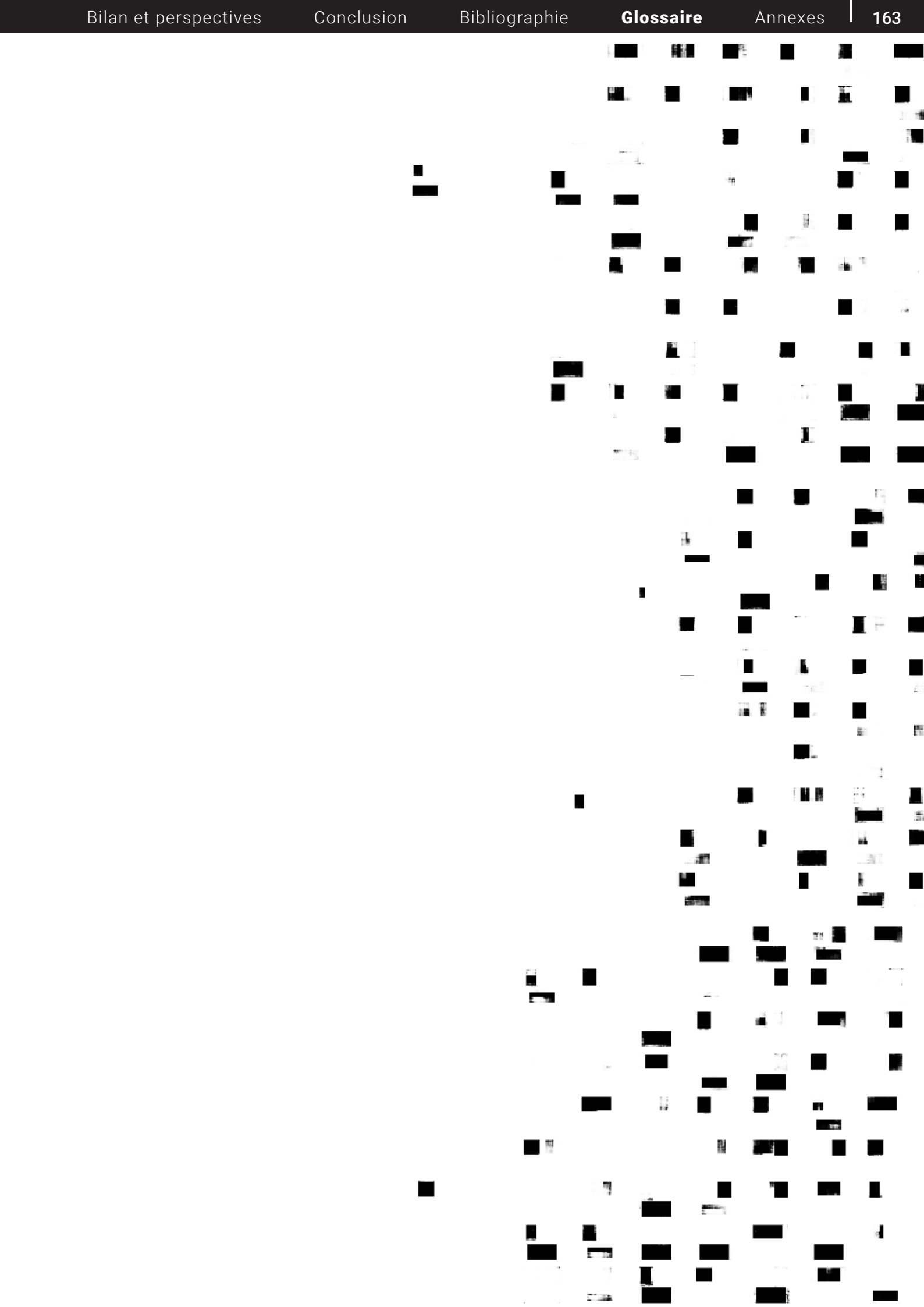
**Invite de commandes :**

**- :** lettre.

**-- :** option.

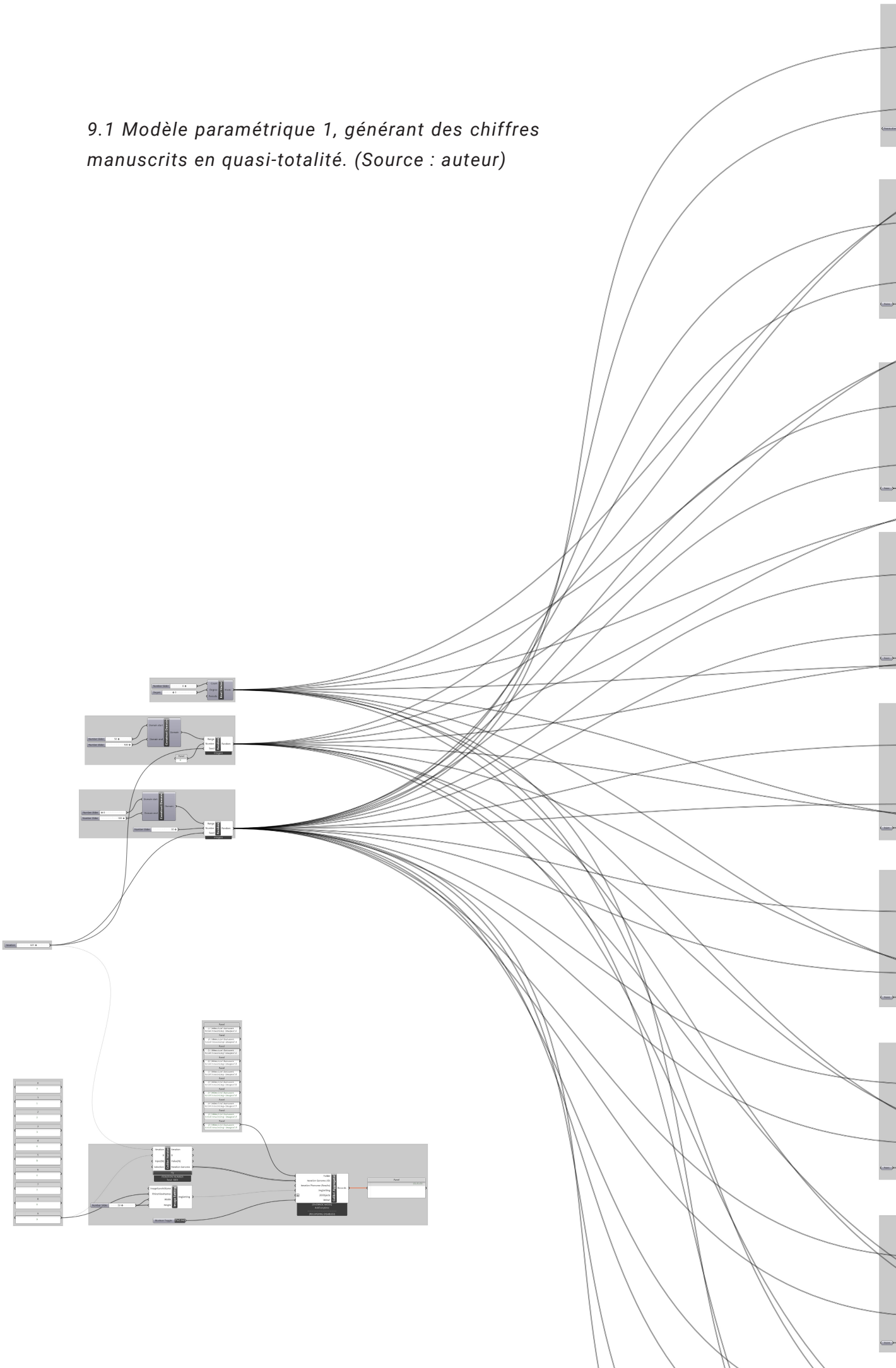
**mkdir :** make directory, crée un nouveau répertoire.

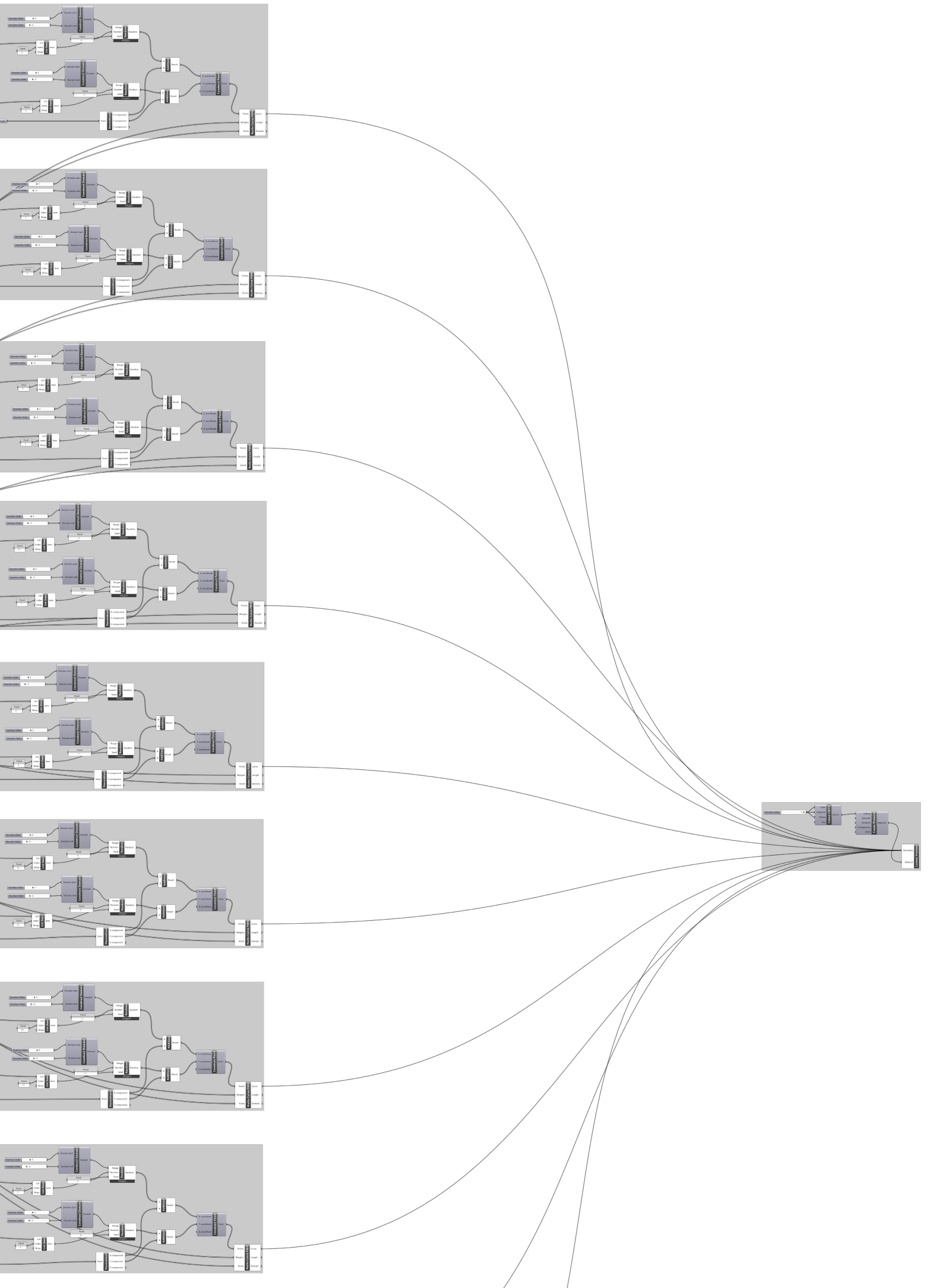
**cd :** change directory, changer de répertoire.



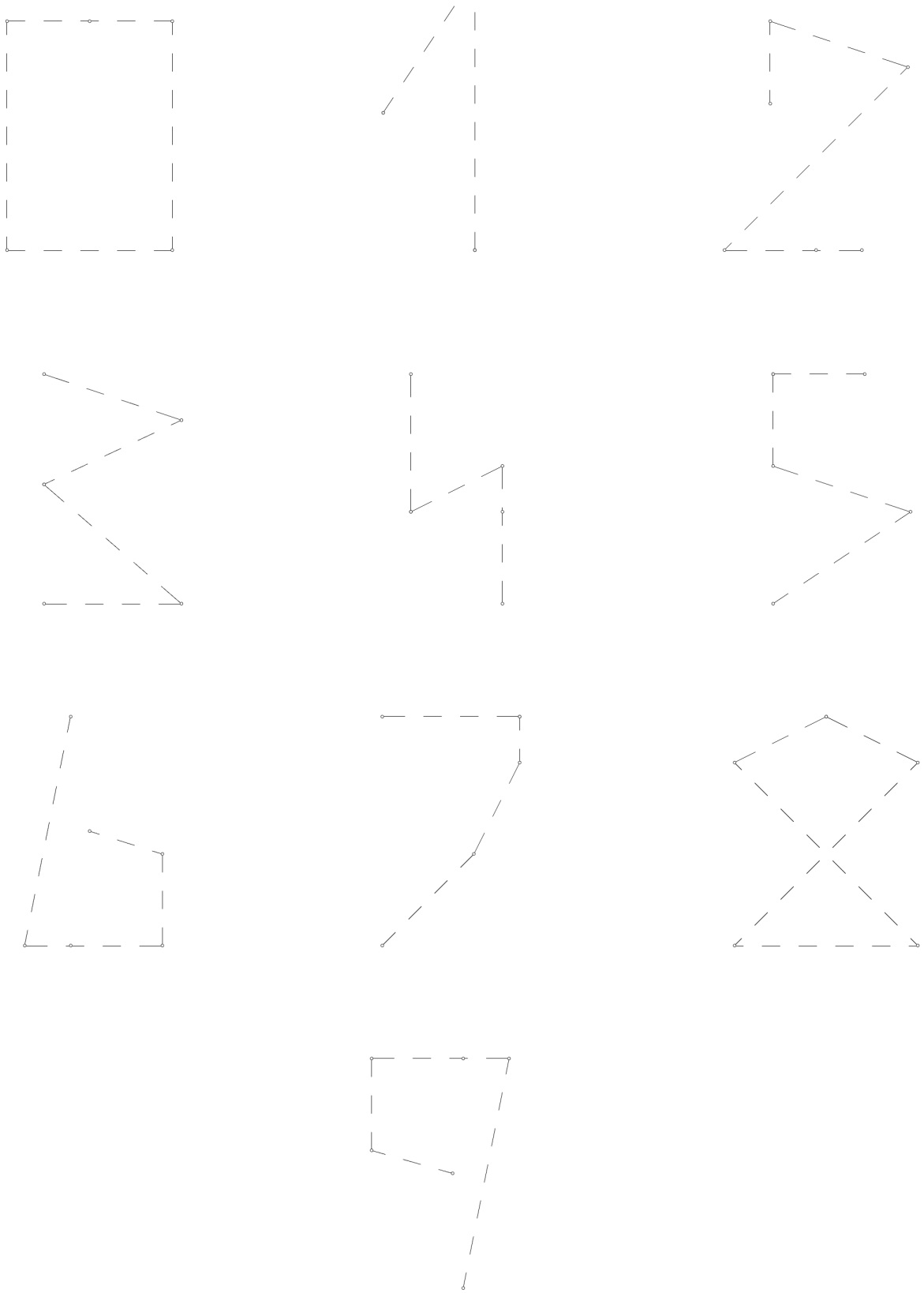
# 9
# ANNEXES

*9.1 Modèle paramétrique 1, générant des chiffres manuscrits en quasi-totalité. (Source : auteur)*

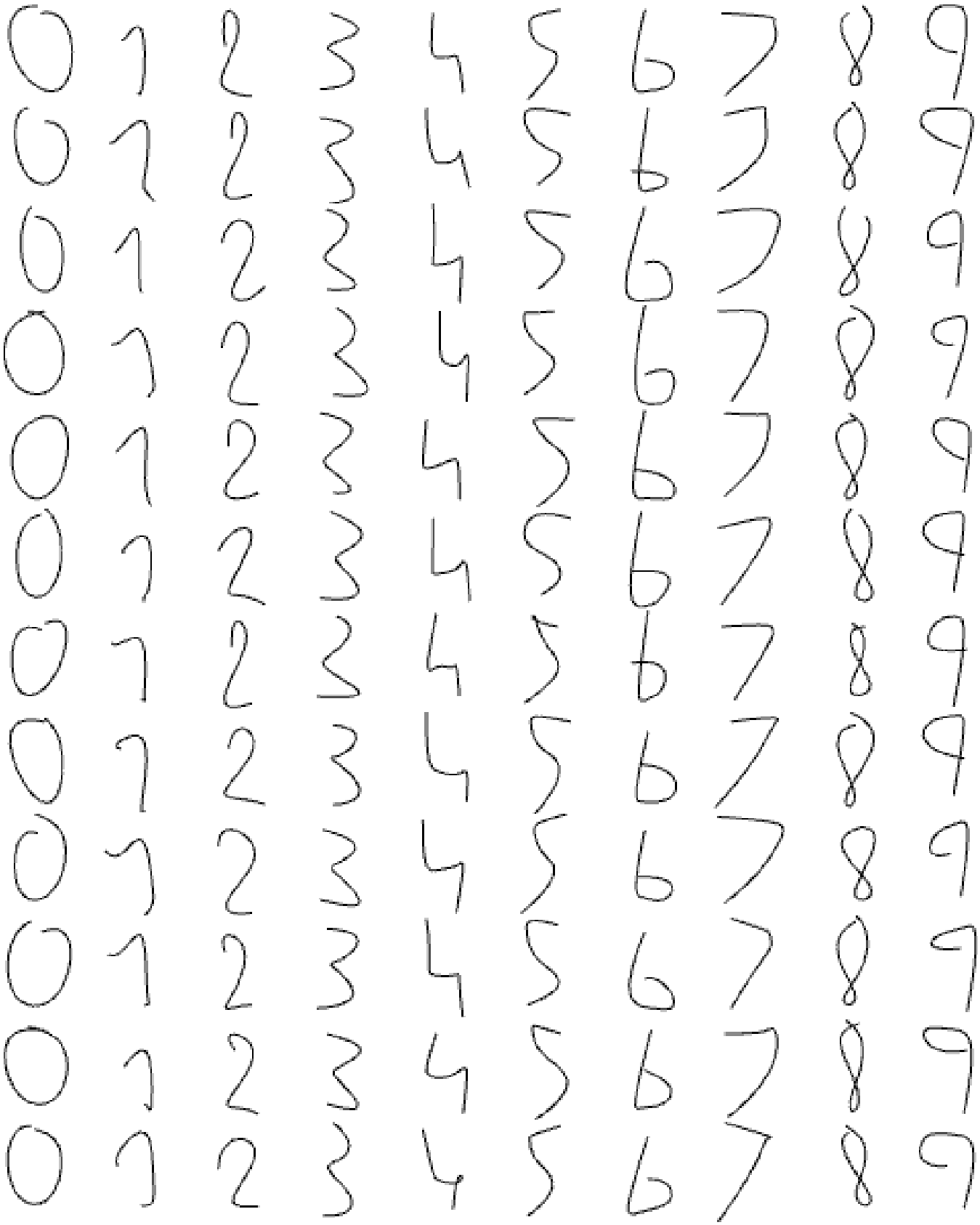




*9.2 Points d'ancrage de chaque chiffre, reliés par des pointillés.*
*(Source : auteur)*



*9.3 Résultats du modèle paramétrique générant des chiffres manuscrits. (Source : auteur)*



*9.4 Intégralité du code constituant le **GAN** de l'expérimentation. (Source : auteur)*

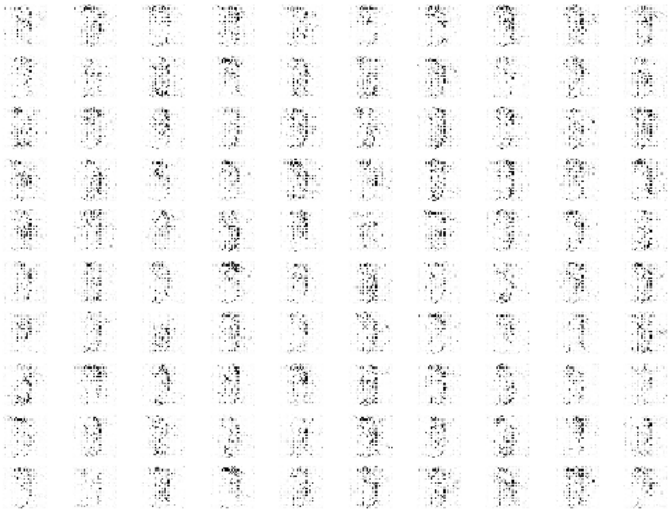
```
# example of training a stable gan for generating a handwritten digit
from os import makedirs
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import BatchNormalization
from keras.initializers import RandomNormal
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
	# weight initialization
	init = RandomNormal(stddev=0.02)
	# define model
	model = Sequential()
	# downsample to 14x14
	model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init, input_shape=in_shape))
	model.add(BatchNormalization())
	model.add(LeakyReLU(alpha=0.2))
	# downsample to 7x7
	model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
	model.add(BatchNormalization())
	model.add(LeakyReLU(alpha=0.2))
	# classifier
	model.add(Flatten())
	model.add(Dense(1, activation='sigmoid'))
	# compile model
	opt = Adam(lr=0.0002, beta_1=0.5)
	model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
	return model

# define the standalone generator model
def define_generator(latent_dim):
	# weight initialization
	init = RandomNormal(stddev=0.02)
	# define model
	model = Sequential()
	# foundation for 7x7 image
	n_nodes = 128 * 7 * 7
	model.add(Dense(n_nodes, kernel_initializer=init, input_dim=latent_dim))
	model.add(LeakyReLU(alpha=0.2))
	model.add(Reshape((7, 7, 128)))
	# upsample to 14x14
	model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
	model.add(BatchNormalization())
	model.add(LeakyReLU(alpha=0.2))
	# upsample to 28x28
	model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
	model.add(BatchNormalization())
	model.add(LeakyReLU(alpha=0.2))
	# output 28x28x1
	model.add(Conv2D(1, (7,7), activation='tanh', padding='same', kernel_initializer=init))
	return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
	# make weights in the discriminator not trainable
	discriminator.trainable = False
	# connect them
	model = Sequential()
	# add generator
	model.add(generator)
	# add the discriminator
	model.add(discriminator)
	# compile model
	opt = Adam(lr=0.0002, beta_1=0.5)
	model.compile(loss='binary_crossentropy', optimizer=opt)
	return model
```

```
import mlxtend
import numpy as np

from mlxtend.data import loadlocal_mnist

trainX, trainy = loadlocal_mnist(
        images_path='/Users/JR/Desktop/Ubytes copie/train-images-idx3-ubyte',
        labels_path='/Users/JR/Desktop/Ubytes copie/train-labels-idx1-ubyte')

print(trainX.shape[0], trainX.shape[1])
trainX=np.reshape(trainX,(trainX.shape[0],28,-1))
print(trainX.shape[0], trainX.shape[1], trainX.shape[2])


# load mnist images
def load_real_samples():
        # load dataset
        # expand to 3d, e.g. add channels


        X = expand_dims(trainX, axis=-1)
        # select all of the examples for a given class
        selected_ix = trainy
        X = X[selected_ix]
        # convert from ints to floats
        X = X.astype('float32')
        # scale from [0,255] to [-1,1]
        X = (X - 127.5) / 127.5
        return X

# select real samples
def generate_real_samples(dataset, n_samples):
        # choose random instances
        ix = randint(0, dataset.shape[0], n_samples)
        # select images
        X = dataset[ix]
        # generate class labels
        y = ones((n_samples, 1))
        return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
        # generate points in the latent space
        x_input = randn(latent_dim * n_samples)
        # reshape into a batch of inputs for the network
        x_input = x_input.reshape(n_samples, latent_dim)
        return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
        # generate points in latent space
        x_input = generate_latent_points(latent_dim, n_samples)
        # predict outputs
        X = generator.predict(x_input)
        # create class labels
        y = zeros((n_samples, 1))
        return X, y

# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, latent_dim, n_samples=100):
        # prepare fake examples
        X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
        # scale from [-1,1] to [0,1]
        X = (X + 1) / 2.0
        # plot images
        for i in range(10 * 10):
                # define subplot
                pyplot.subplot(10, 10, 1 + i)
                # turn off axis
                pyplot.axis('off')
                # plot raw pixel data
                pyplot.imshow(X[i, :, :, 0], cmap='gray')
        # save plot to file
        pyplot.savefig('results_baseline/generated_plot_%03d.png' % (step+1))
        pyplot.close()
        # save the generator model
        g_model.save('results_baseline/model_%03d.h5' % (step+1))
```

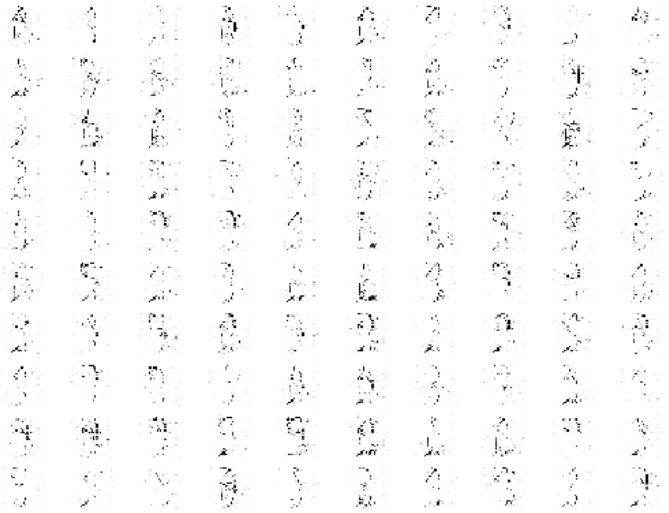
```
# create a line plot of loss for the gan and save to file
def plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist):
        # plot loss
        pyplot.subplot(2, 1, 1)
        pyplot.plot(d1_hist, label='d-real')
        pyplot.plot(d2_hist, label='d-fake')
        pyplot.plot(g_hist, label='gen')
        pyplot.legend()
        # plot discriminator accuracy
        pyplot.subplot(2, 1, 2)
        pyplot.plot(a1_hist, label='acc-real')
        pyplot.plot(a2_hist, label='acc-fake')
        pyplot.legend()
        # save plot to file
        pyplot.savefig('results_baseline/plot_line_plot_loss.png')
        pyplot.close()

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=10, n_batch=128):
        # calculate the number of batches per epoch
        bat_per_epo = int(dataset.shape[0] / n_batch)
        print(bat_per_epo)
        # calculate the total iterations based on batch and epoch
        n_steps = bat_per_epo * n_epochs
        # calculate the number of samples in half a batch
        half_batch = int(n_batch / 2)
        # prepare lists for storing stats each iteration
        d1_hist, d2_hist, g_hist, a1_hist, a2_hist = list(), list(), list(), list(), list()
        # manually enumerate epochs
        for i in range(n_steps):
                # get randomly selected 'real' samples
                X_real, y_real = generate_real_samples(dataset, half_batch)
                # update discriminator model weights
                d_loss1, d_acc1 = d_model.train_on_batch(X_real, y_real)
                # generate 'fake' examples
                X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
                # update discriminator model weights
                d_loss2, d_acc2 = d_model.train_on_batch(X_fake, y_fake)
                # prepare points in latent space as input for the generator
                X_gan = generate_latent_points(latent_dim, n_batch)
                # create inverted labels for the fake samples
                y_gan = ones((n_batch, 1))
                # update the generator via the discriminator's error
                g_loss = gan_model.train_on_batch(X_gan, y_gan)
                # summarize loss on this batch
                print('>%d, d1=%.3f, d2=%.3f g=%.3f, a1=%d, a2=%d' %
                        (i+1, d_loss1, d_loss2, g_loss, int(100*d_acc1), int(100*d_acc2)))
                # record history
                d1_hist.append(d_loss1)
                d2_hist.append(d_loss2)
                g_hist.append(g_loss)
                a1_hist.append(d_acc1)
                a2_hist.append(d_acc2)
                # evaluate the model performance every 'epoch'
                if (i+1) % bat_per_epo == 0:
                        summarize_performance(i, g_model, latent_dim)
        plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist)

# make folder for results
makedirs('results_baseline', exist_ok=True)
# size of the latent space
latent_dim = 50
# create the discriminator
discriminator = define_discriminator()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, discriminator)
# load image data
dataset = load_real_samples()
print(dataset.shape)
# train model
train(generator, discriminator, gan_model, dataset, latent_dim)
```

```python
# example of loading the generator model and generating images
from keras.models import load_model
from numpy.random import randn
from matplotlib import pyplot
import matplotlib.pyplot as plt

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
        # generate points in the latent space
        x_input = randn(latent_dim * n_samples)
        # reshape into a batch of inputs for the network
        x_input = x_input.reshape(n_samples, latent_dim)
        return x_input

# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, n):
        # plot images
        for i in range(n * n):
                # define subplot
                pyplot.subplot(n, n, 1 + i)
                # turn off axis
                pyplot.axis('off')
                # plot raw pixel data
                pyplot.imshow(examples[i, :, :, 0], cmap='gray')
                pyplot.savefig('plot_300dpi.png', dpi=300)
        pyplot.show()

# load model
model = load_model('results_baseline/model_700.h5')
# generate images
latent_points = generate_latent_points(50, 100)
# generate images
X = model.predict(latent_points)
# plot the result
save_plot(X, 10)
```

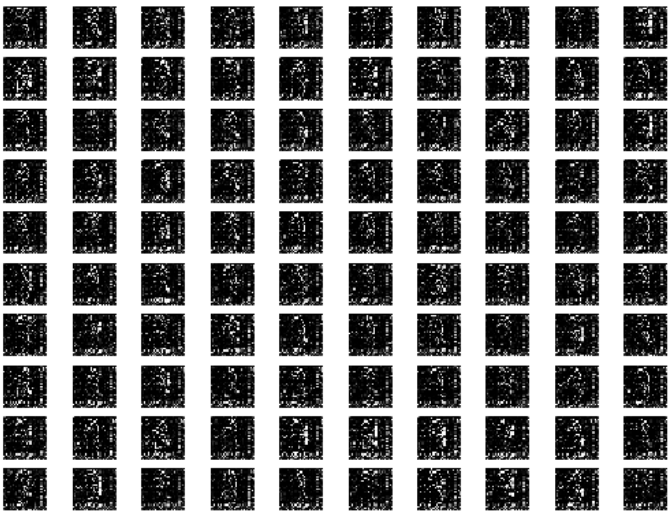
*9.5 Résultats du **GAN** sur les chiffres de 0 à 9, noir sur blanc (A).*
*(Source : auteur)*



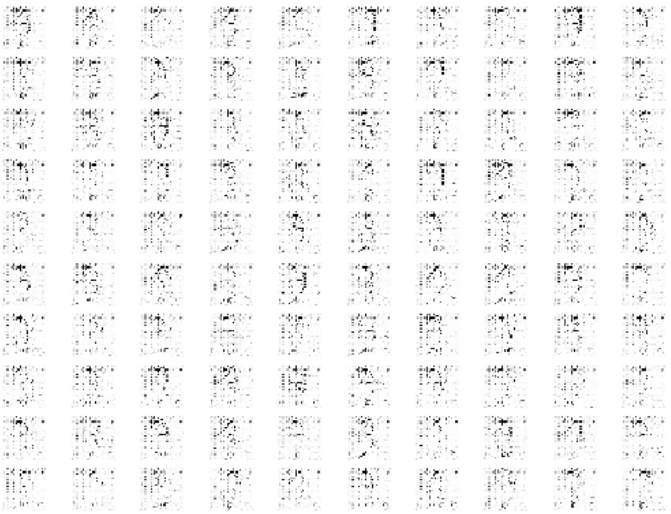
70ème itération



140ème itération

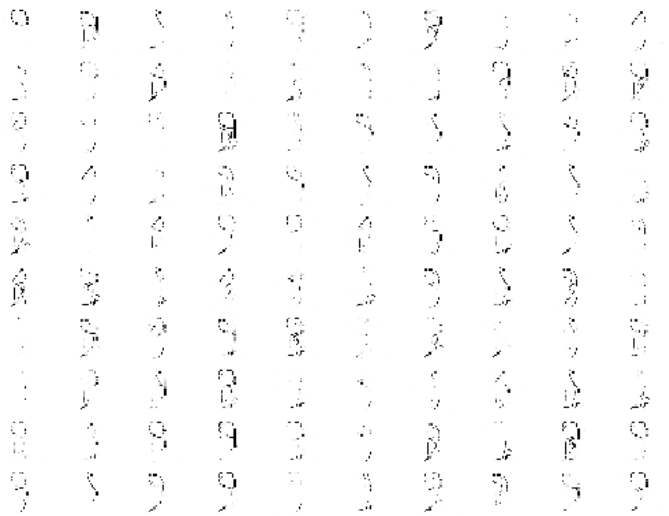


210ème itération

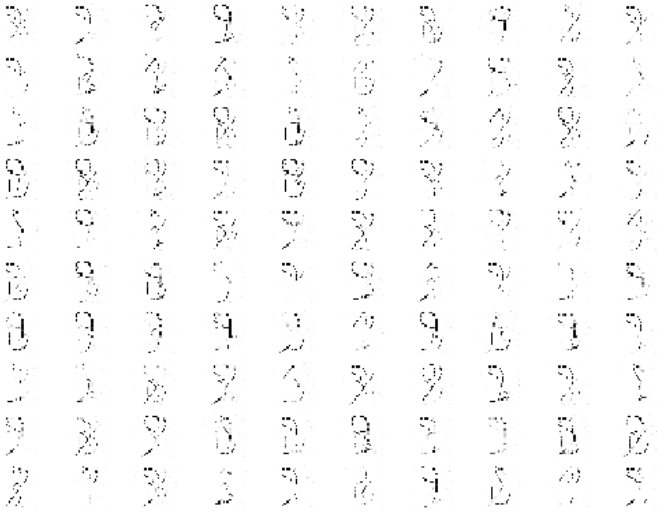


280ème itération

350ème itération

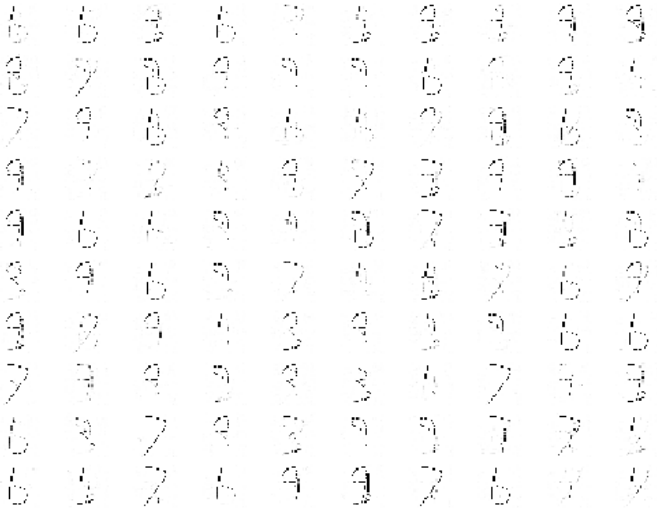


420ème itération

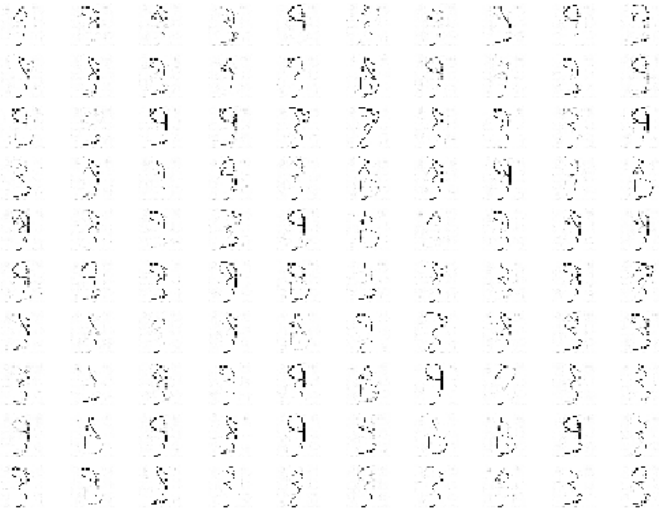


490ème itération

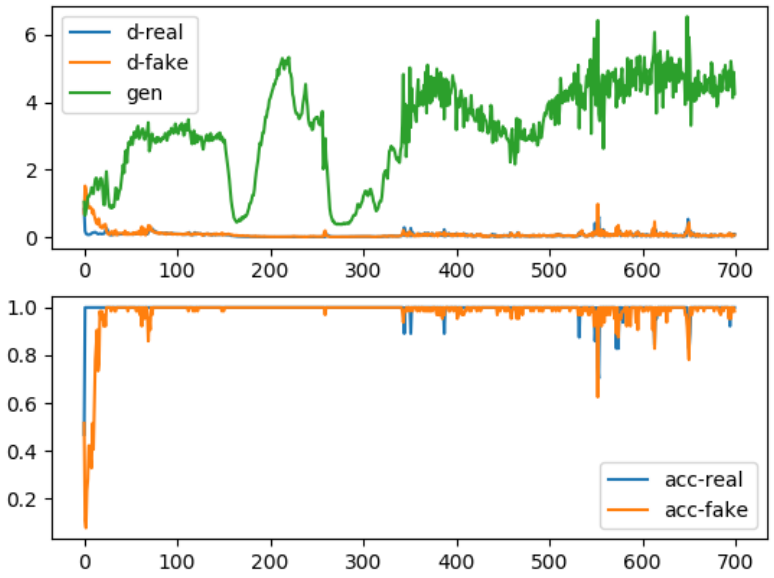
560ème itération



630ème itération

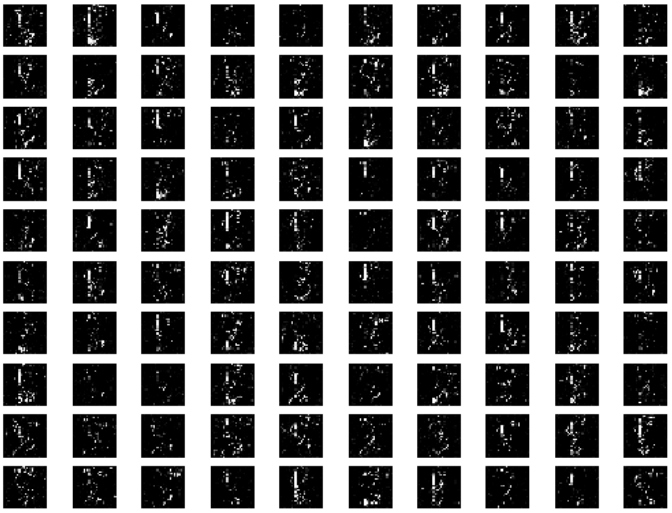


700ème itération

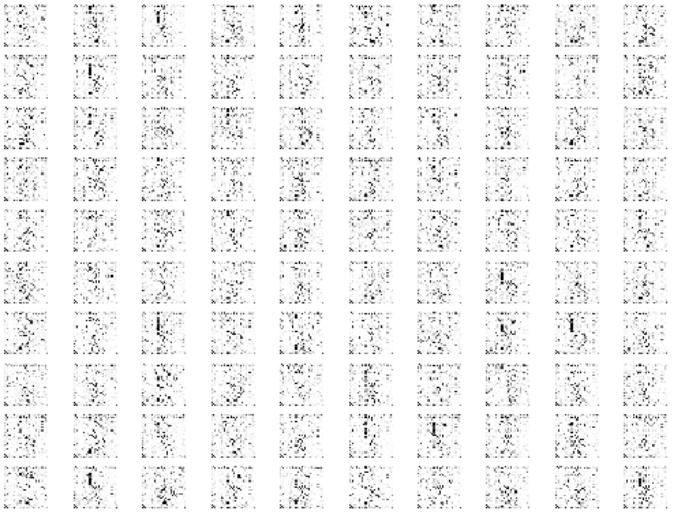



Courbes statistiques associées.

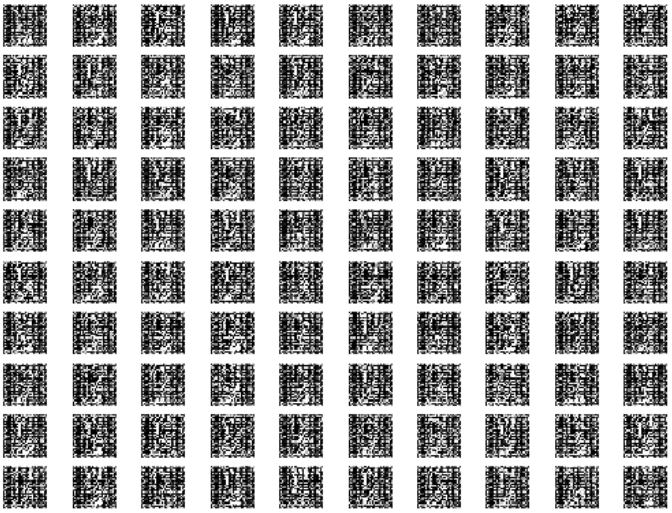
*9.6 Résultats du **GAN** sur les chiffres de 0 à 9, blanc sur noir (B).*
*(Source : auteur)*



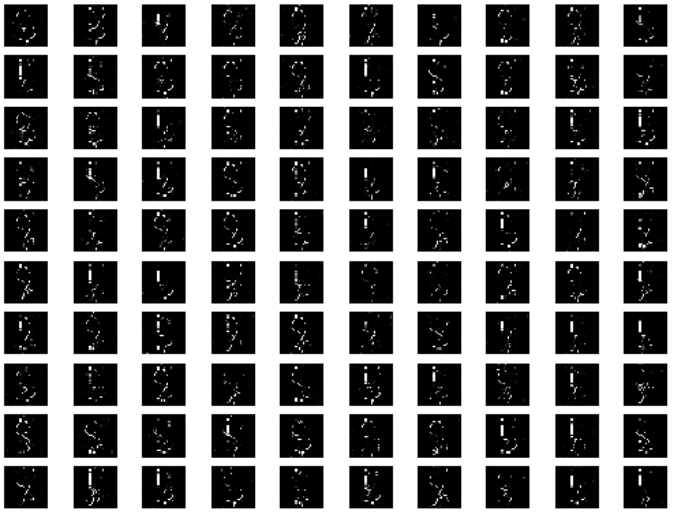
70ème itération



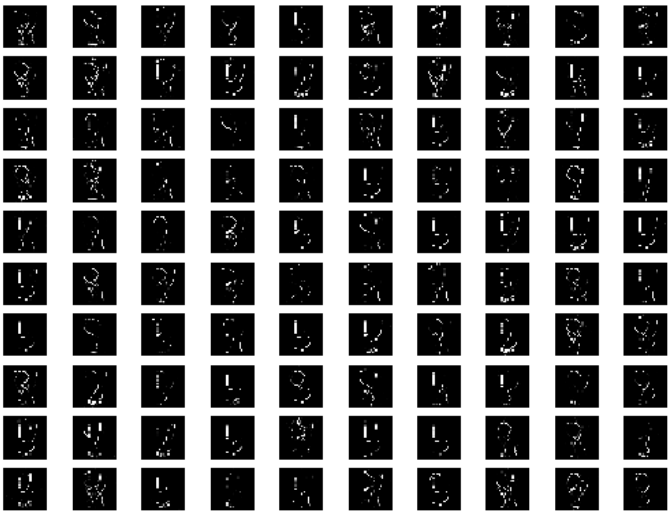
140ème itération



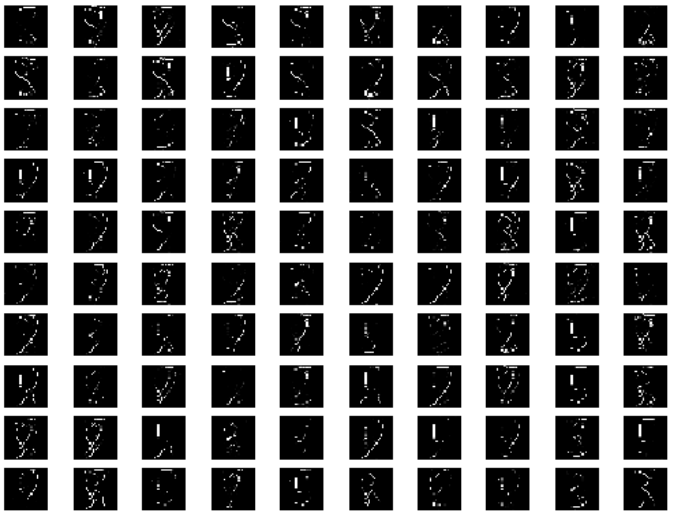
210ème itération



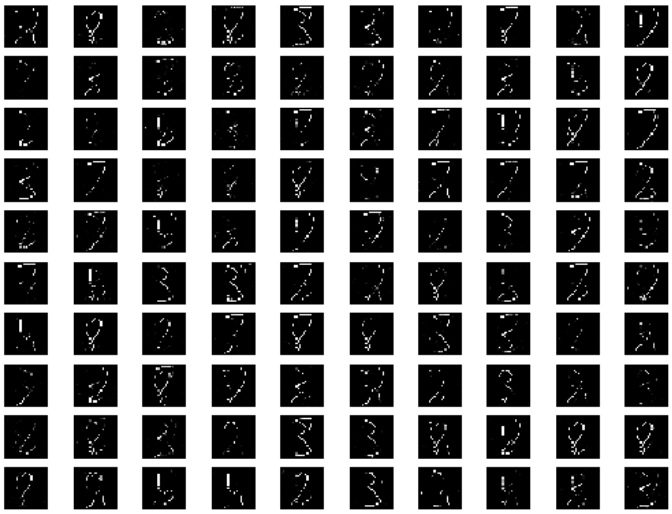
280ème itération



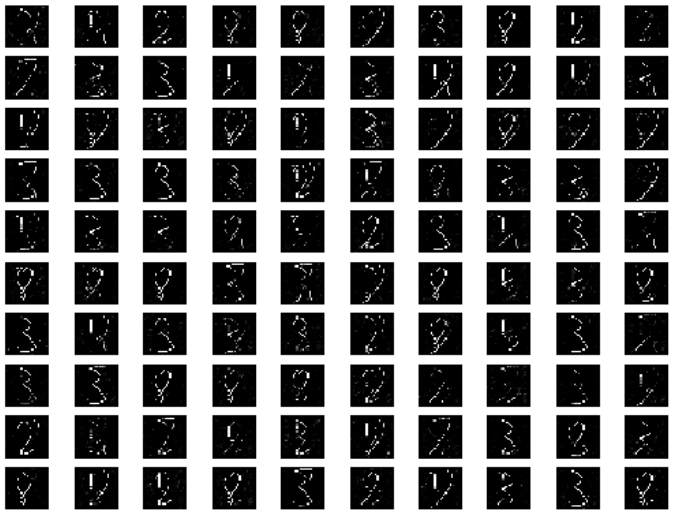
350ème itération



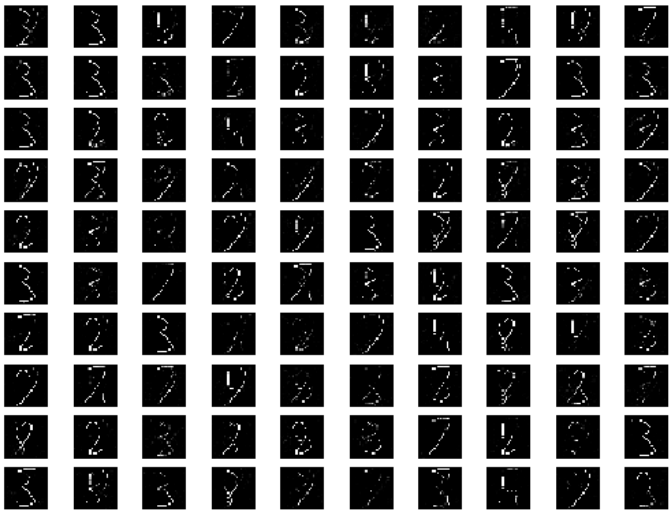
420ème itération



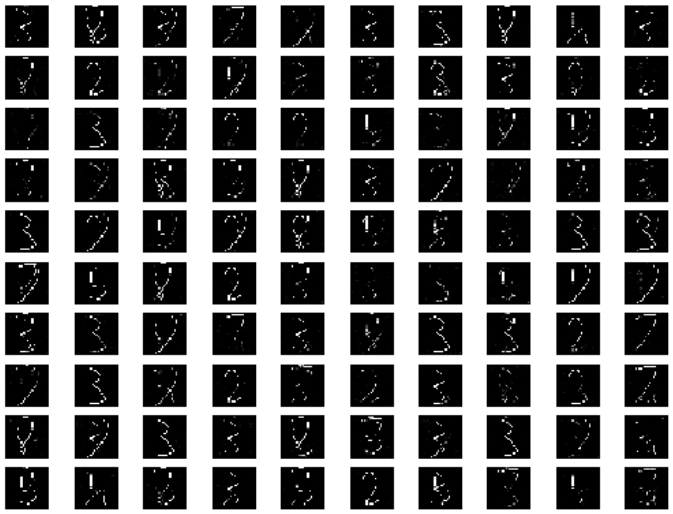
490ème itération



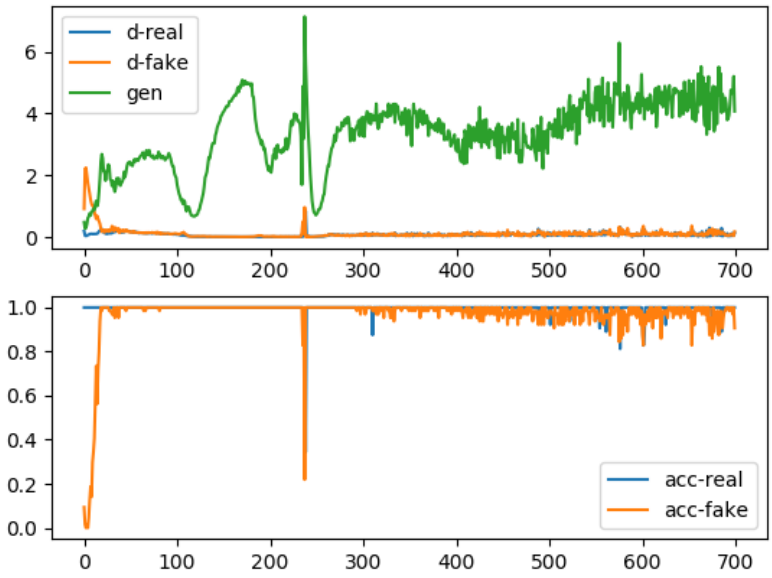
560ème itération



630ème itération

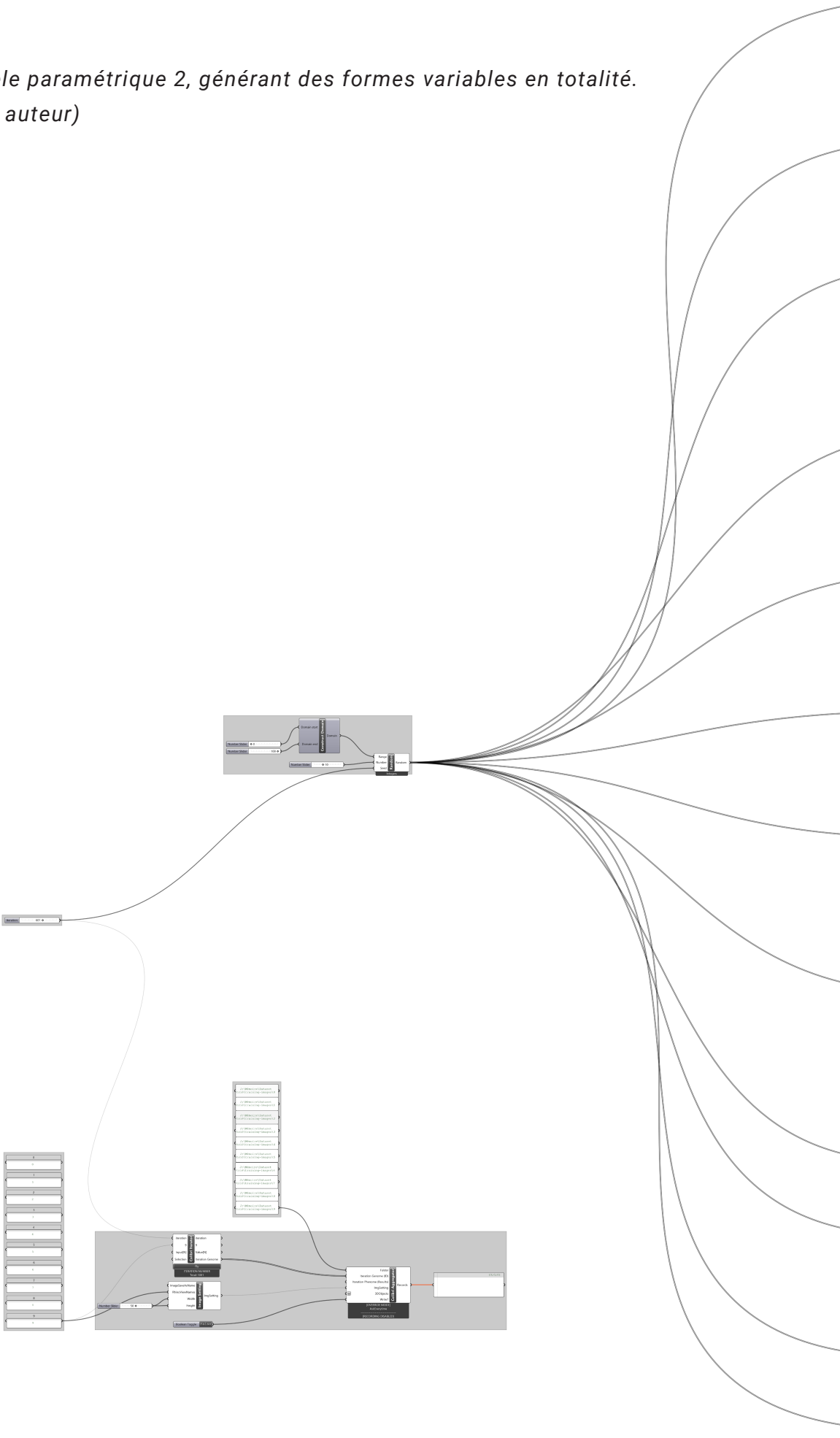


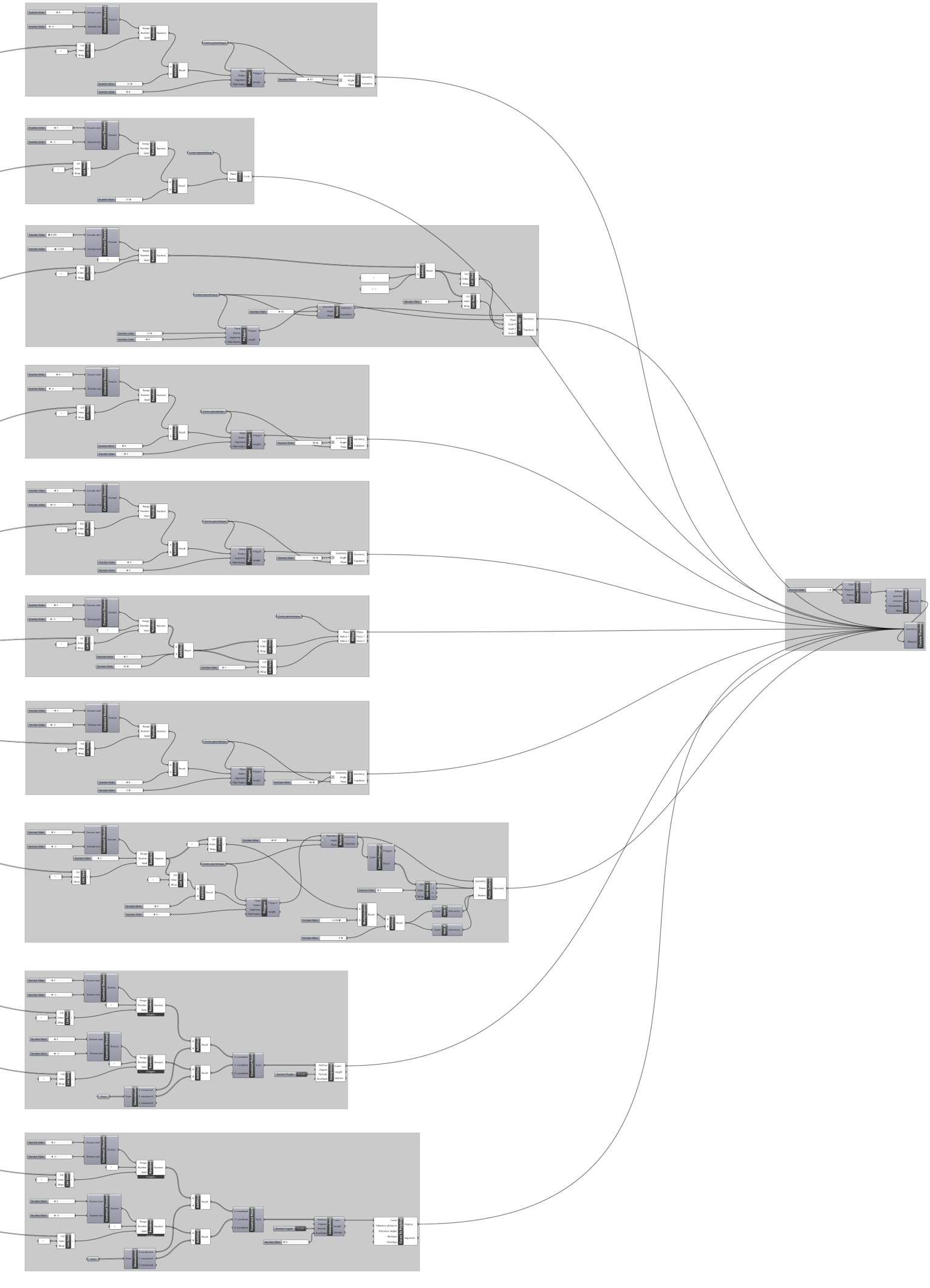
700ème itération



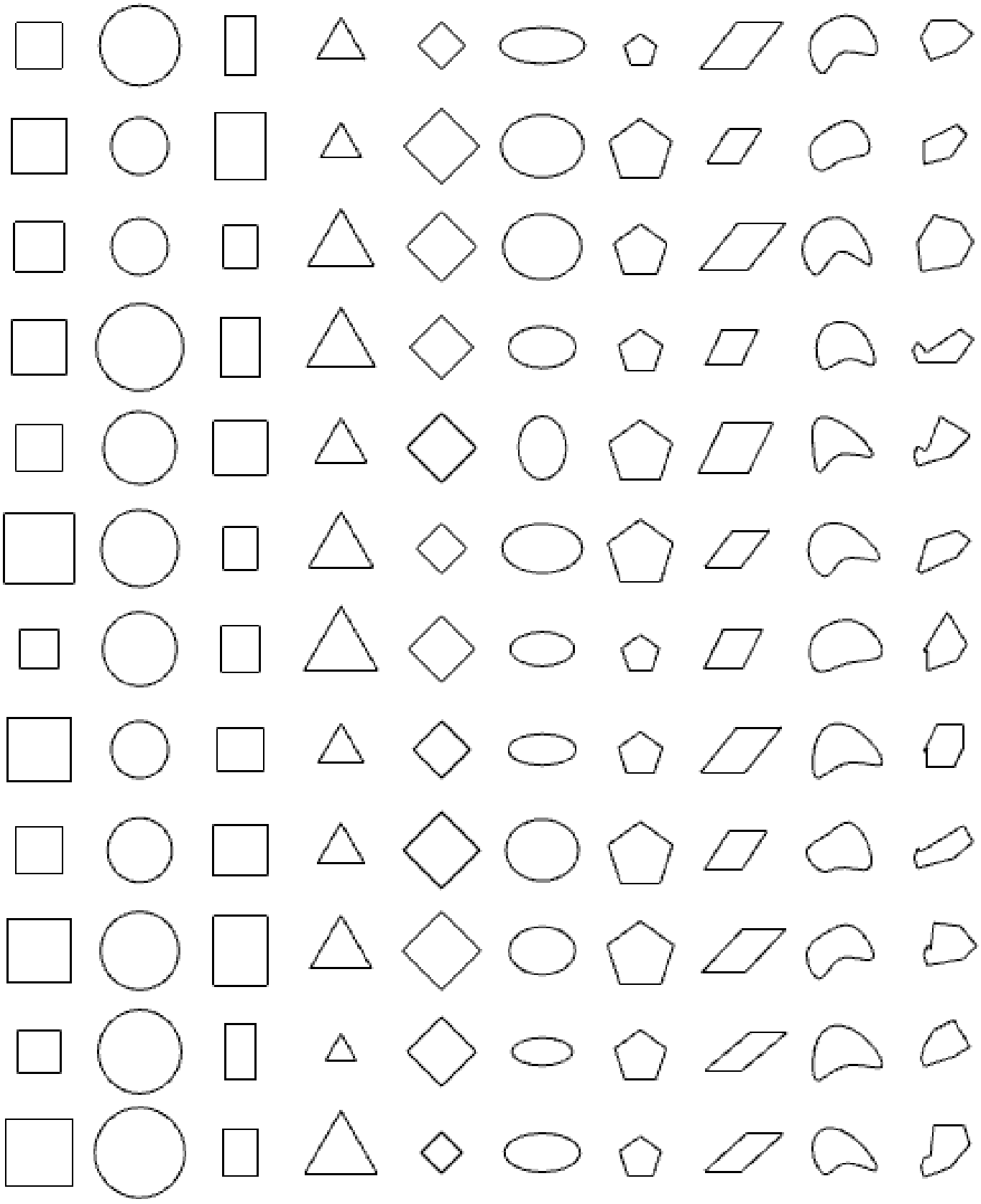

Courbes statistiques associées.

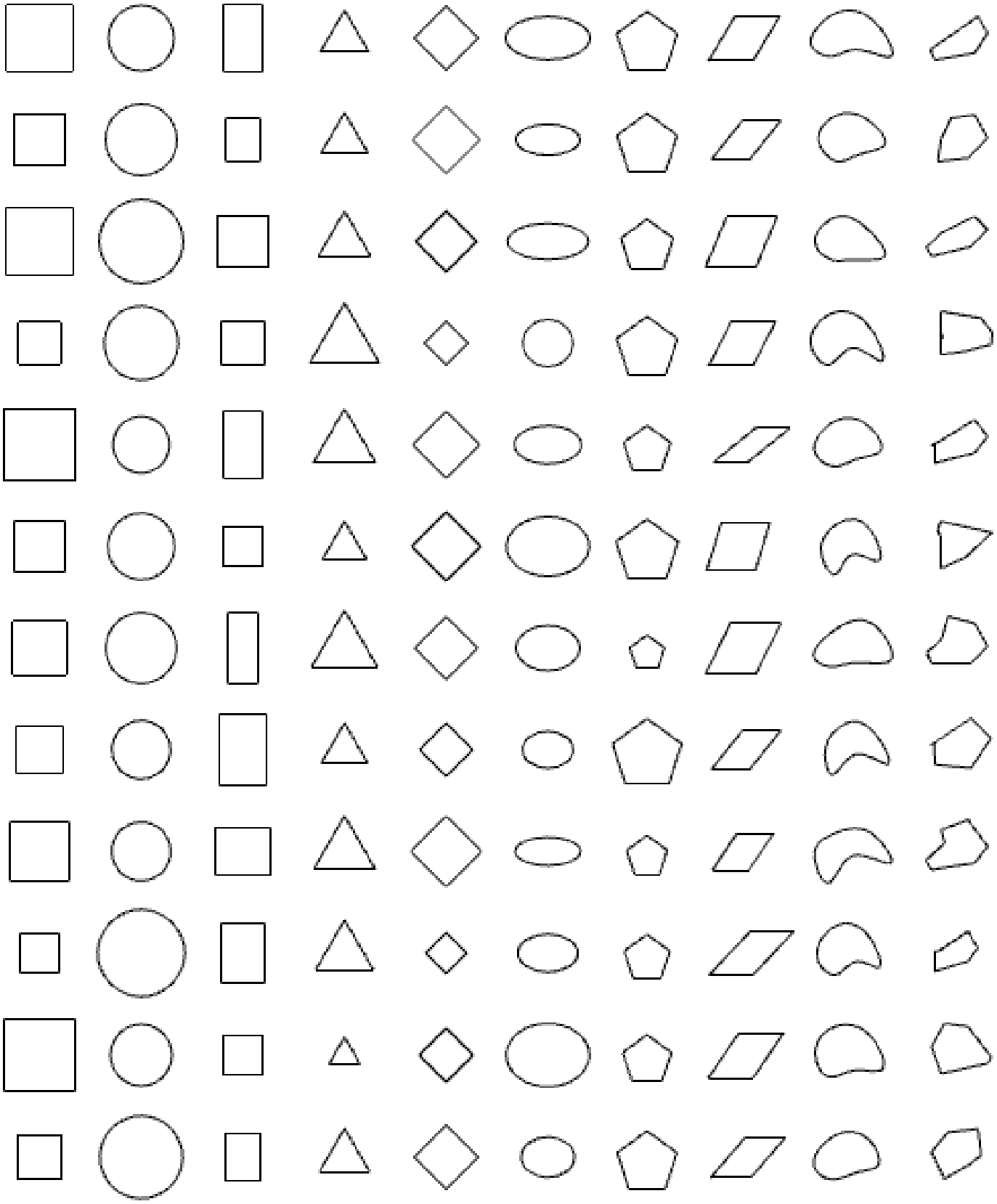
*9.7 Modèle paramétrique 2, générant des formes variables en totalité. (Source : auteur)*



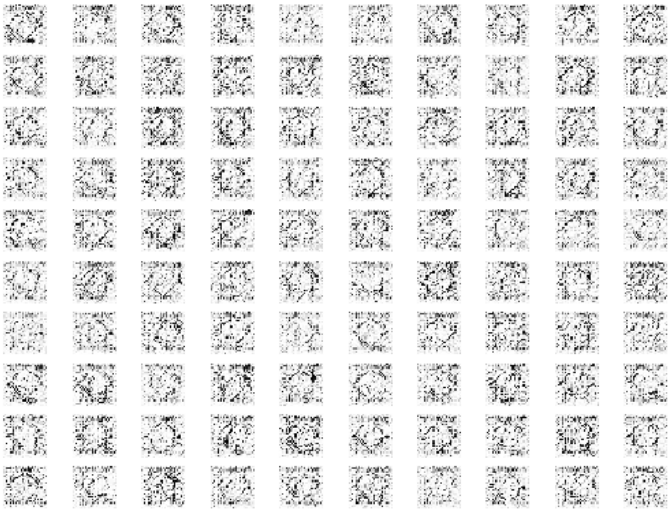


*9.8 Résultats du modèle paramétrique générant des formes variables. (Source : auteur)*

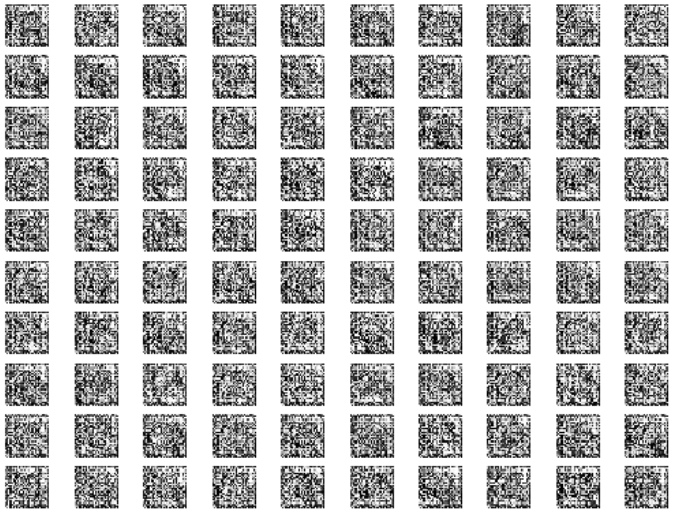




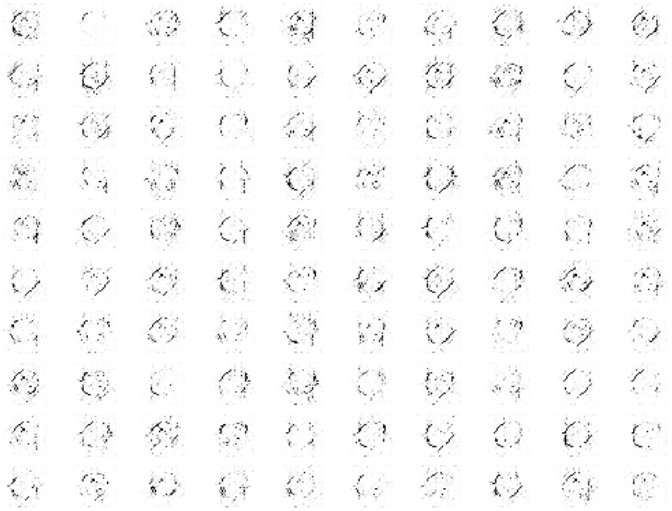
*9.9 Résultats du **GAN** sur les formes géométriques, noir sur blanc (A).*
*(Source : auteur)*



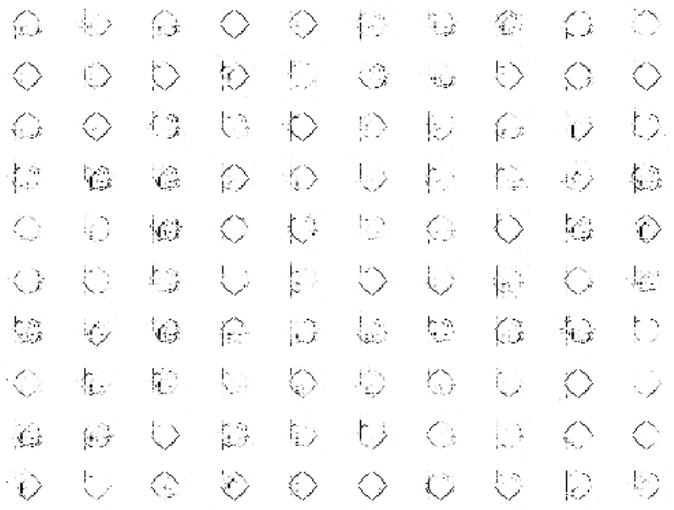
70ème itération



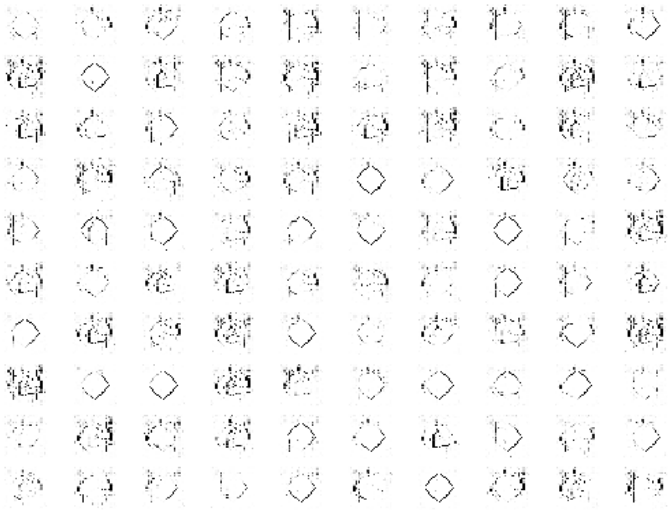
140ème itération



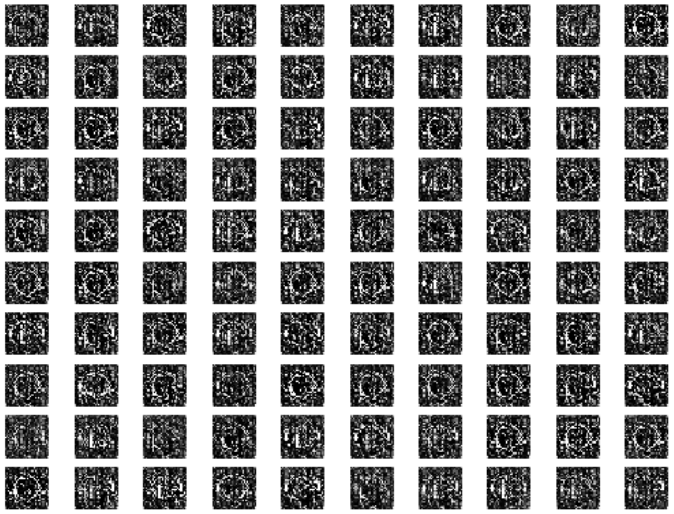
210ème itération



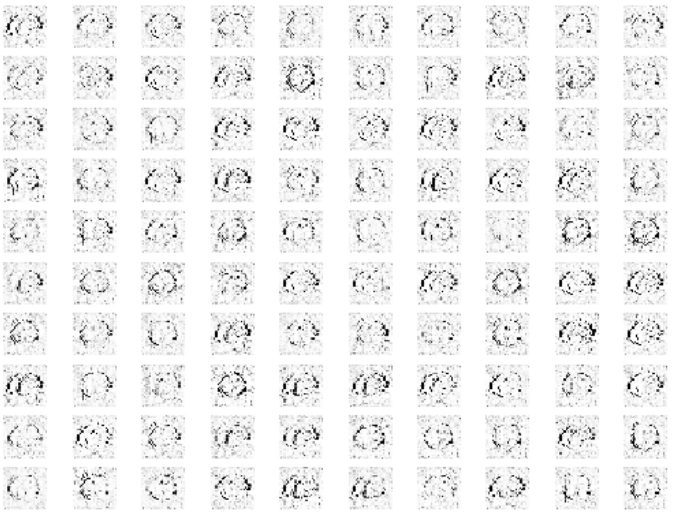
280ème itération



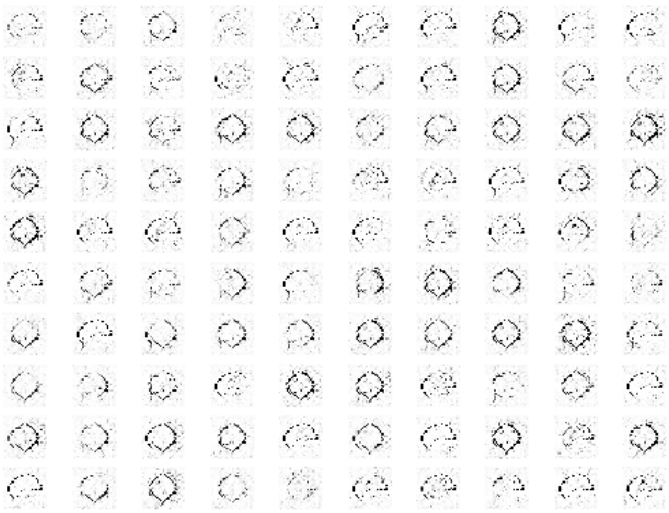
350ème itération



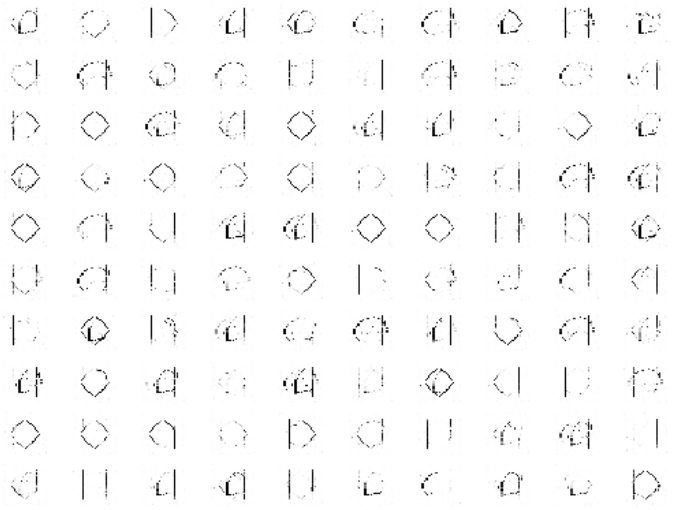
420ème itération



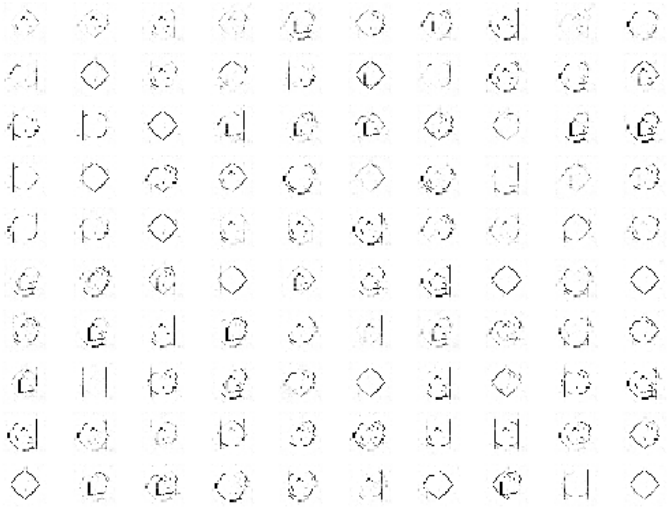
490ème itération



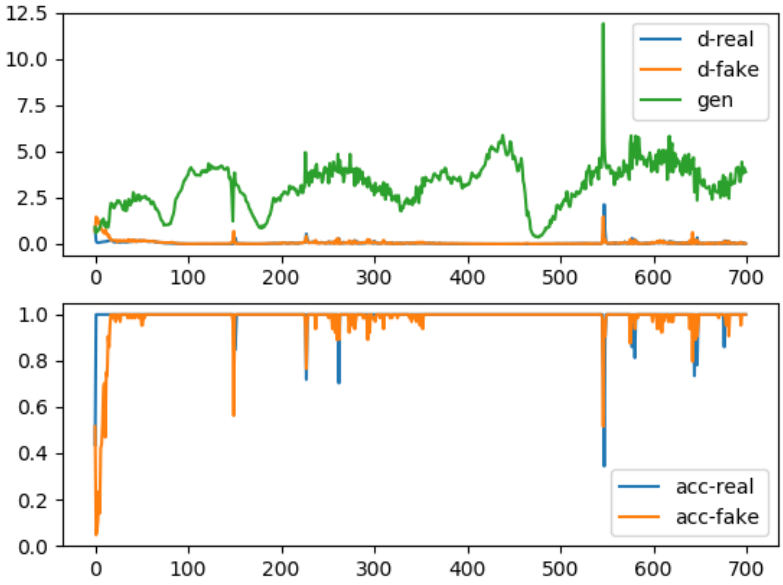
560ème itération



630ème itération

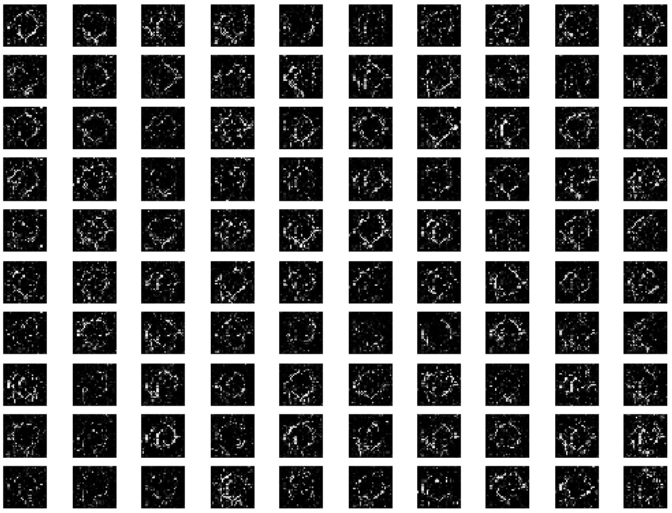


700ème itération

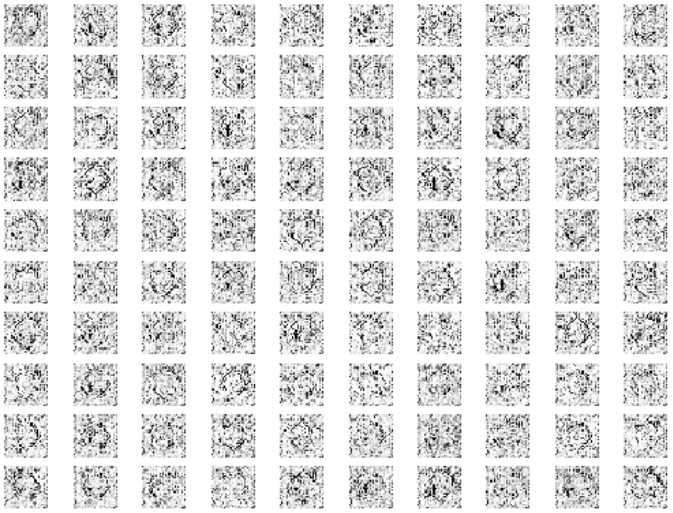



Courbes statistiques associées.

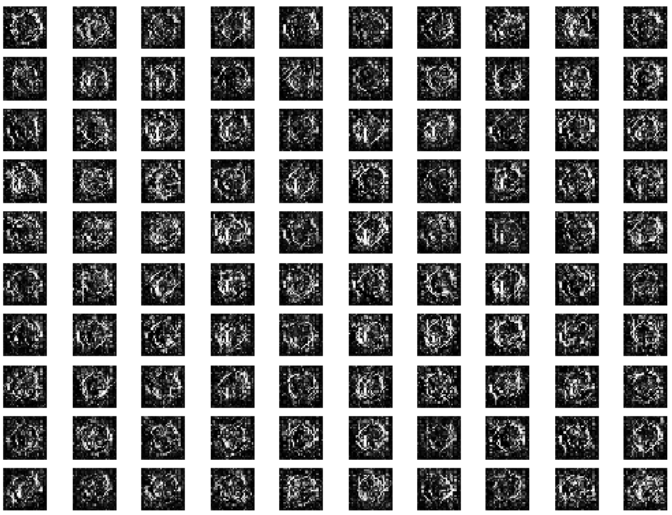
*9.10 Résultats du **GAN** sur les formes géométriques, blanc sur noir (B). (Source : auteur)*



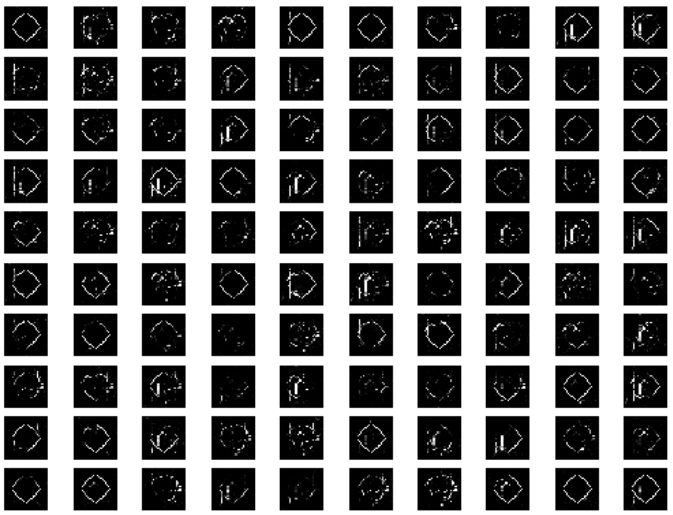
70ème itération



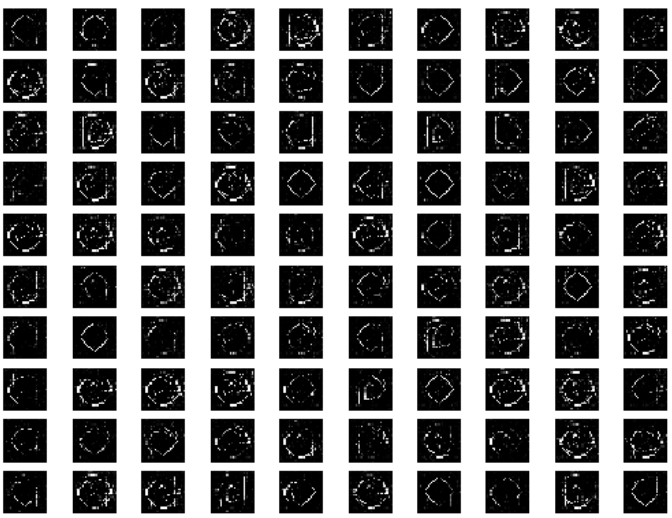
140ème itération



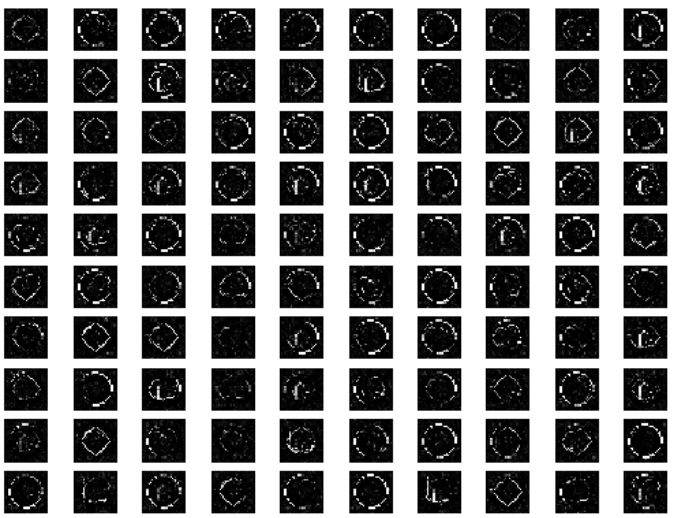
210ème itération



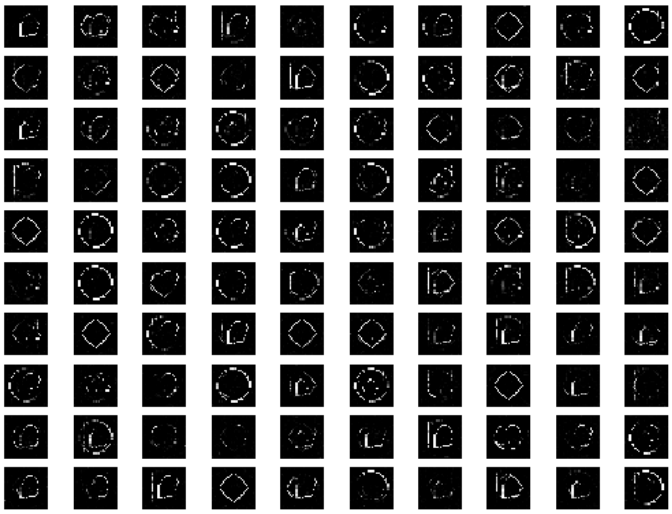
280ème itération



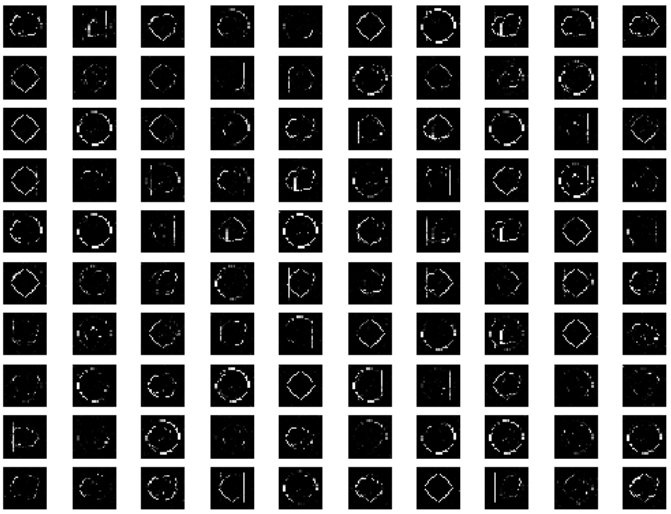
350ème itération



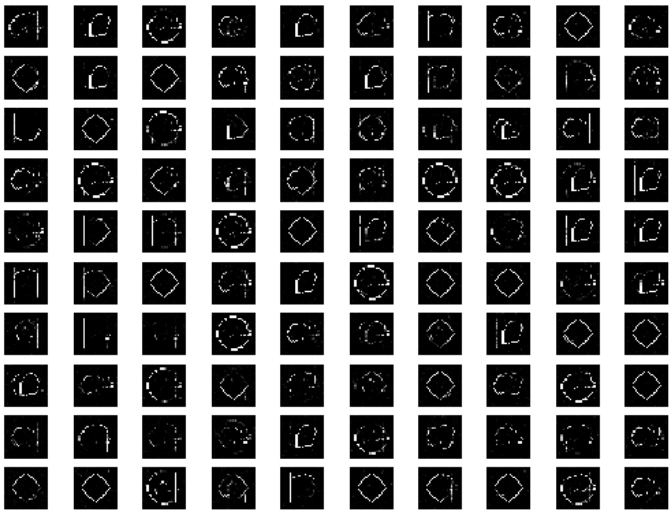
420ème itération



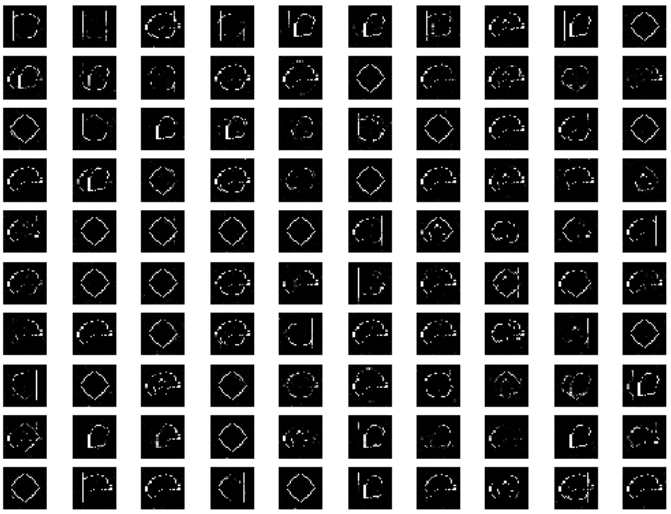
490ème itération



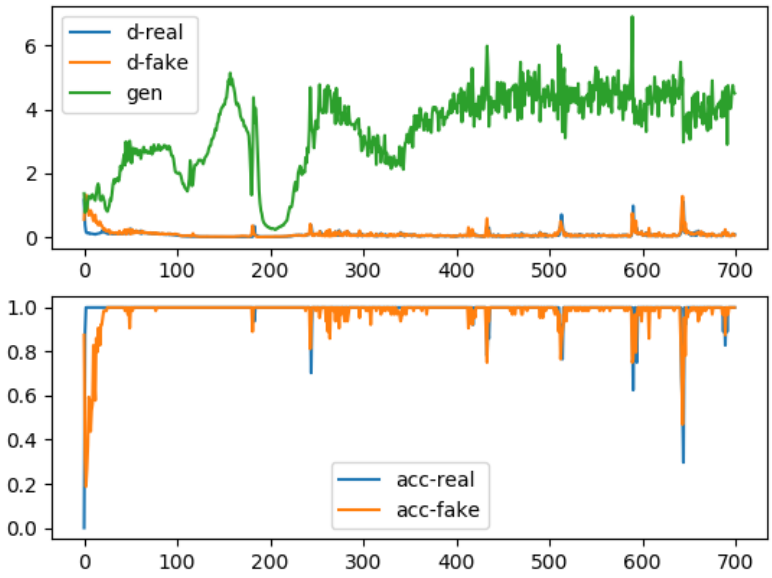
560ème itération



630ème itération

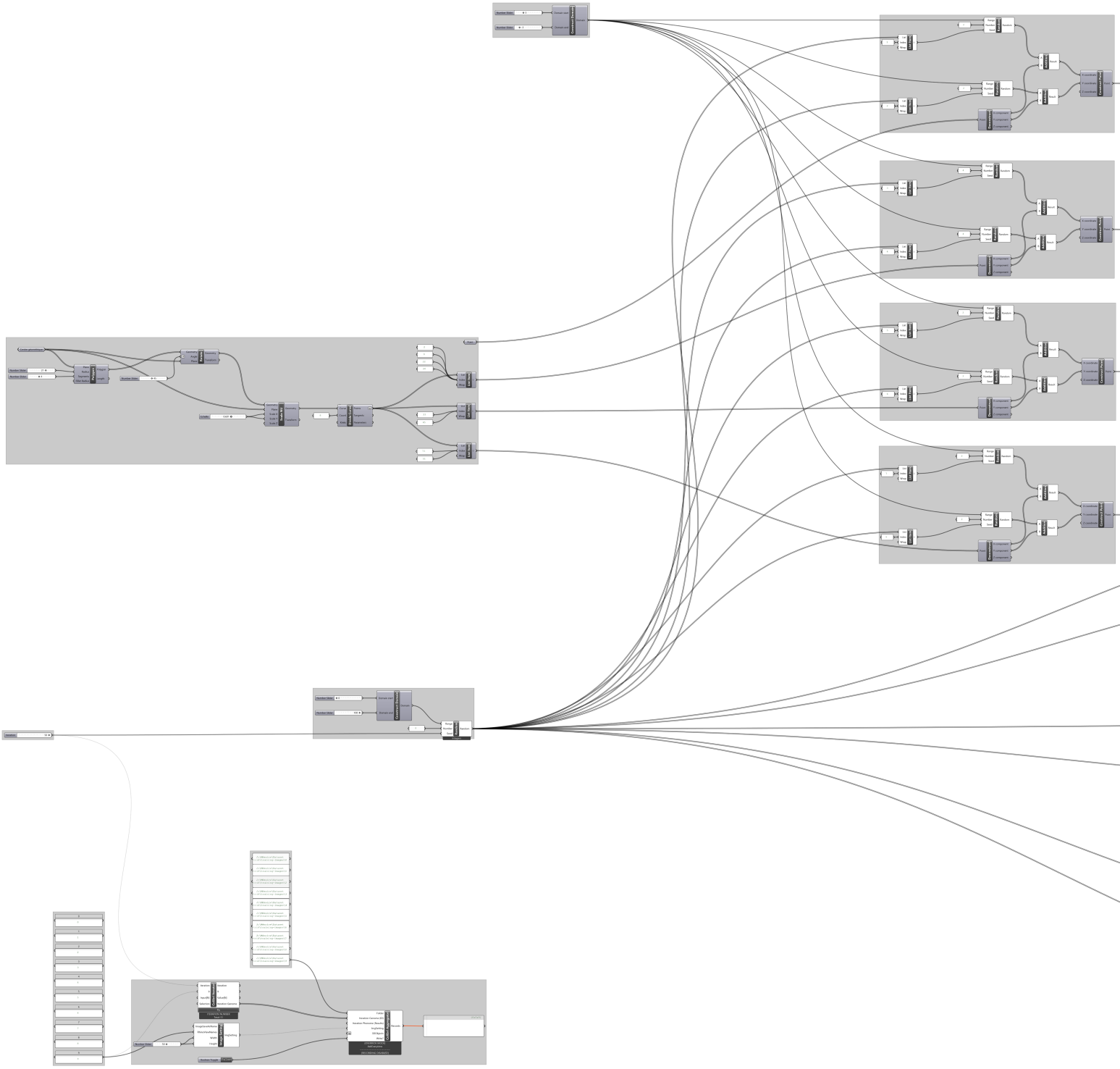


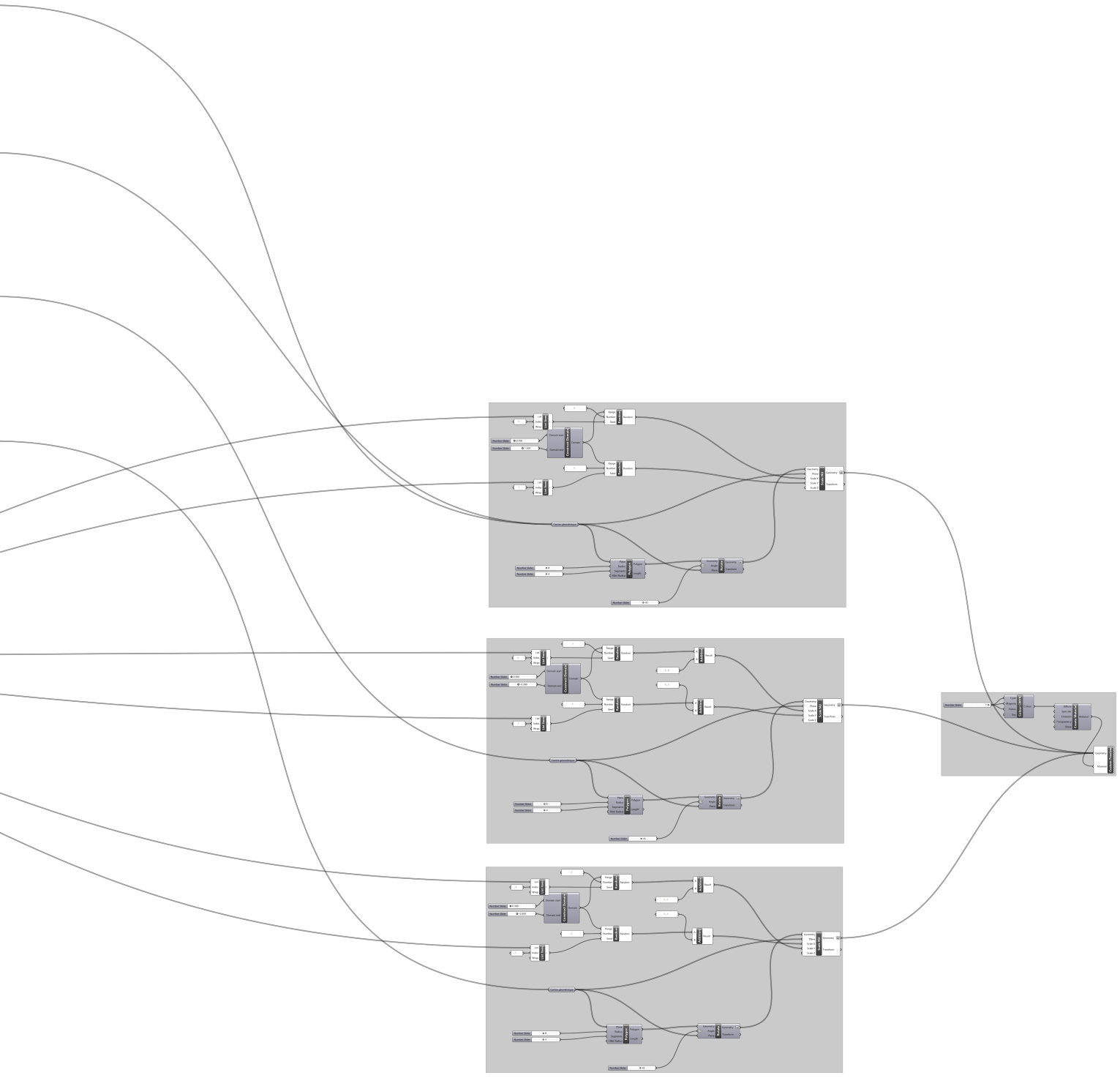
700ème itération



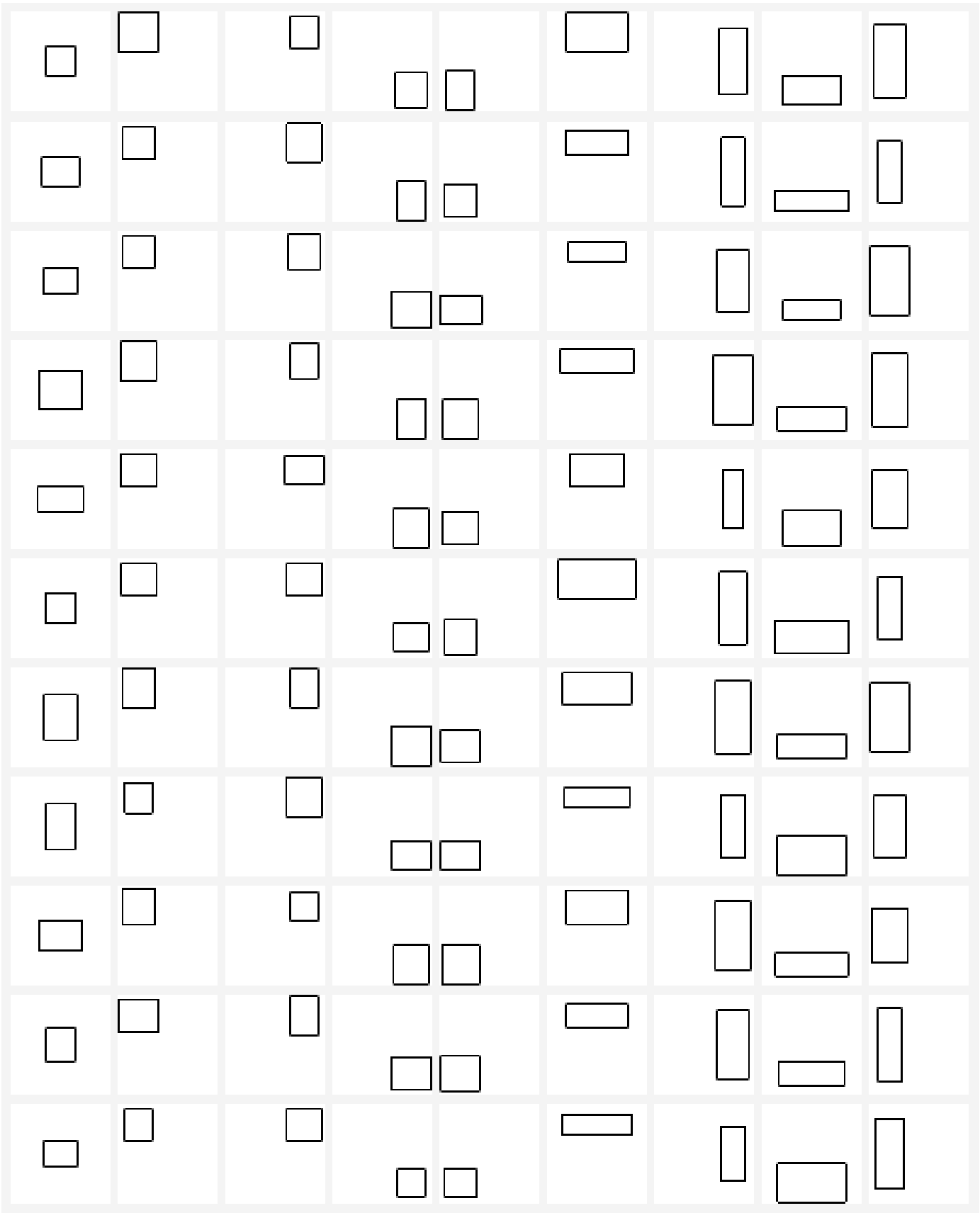

Courbes statistiques associées.

*9.11 Modèle paramétrique 3, générant des dispositions variables en totalité. (Source : auteur)*

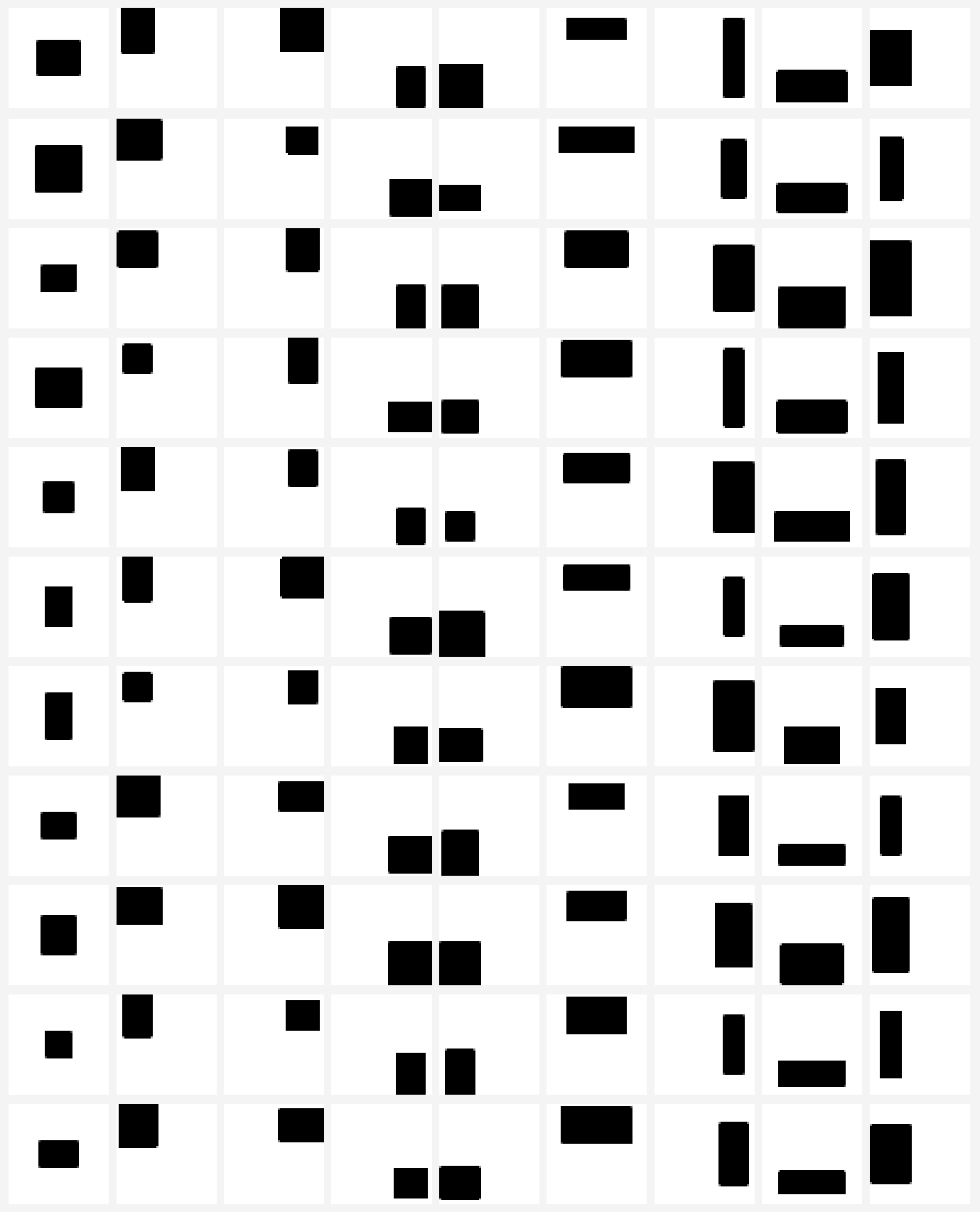




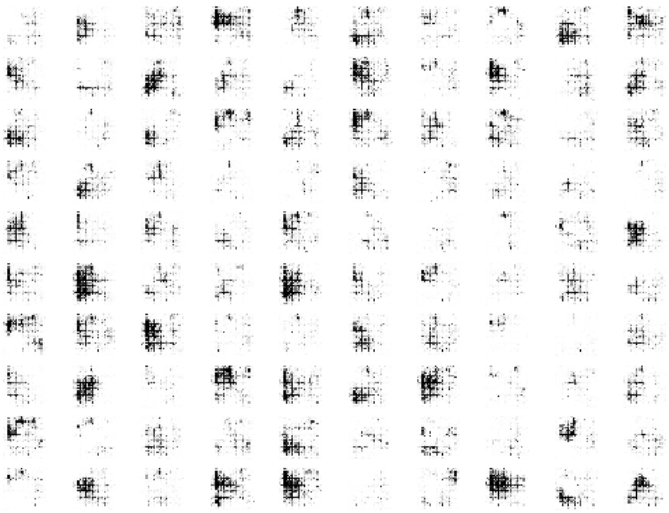
*9.12 Résultats du modèle paramétrique générant les dispositions, versions vides. (Source : auteur)*



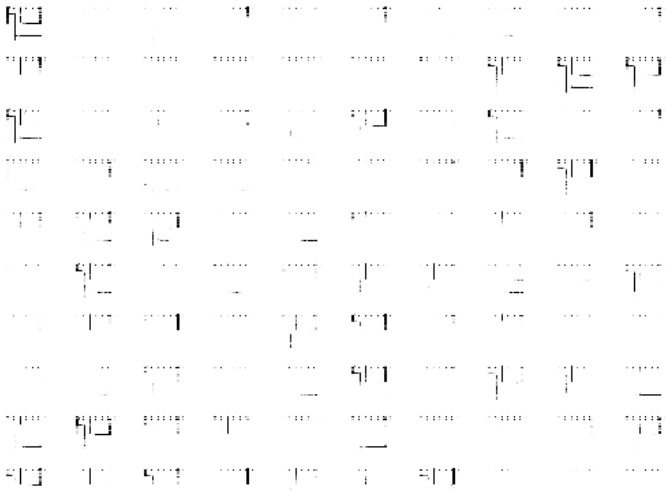
*9.13 Résultats du modèle paramétrique générant les dispositions, versions pleines. (Source : auteur)*



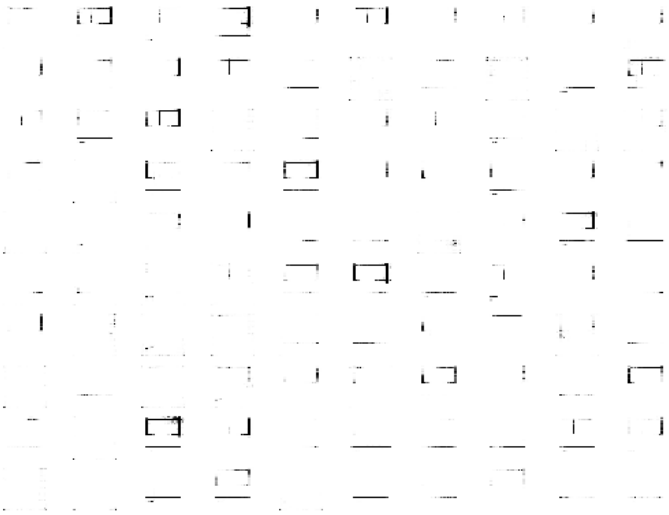
*9.14 Résultats du **GAN** sur les dispositions, formes vides noir sur blanc. (Source : auteur)*



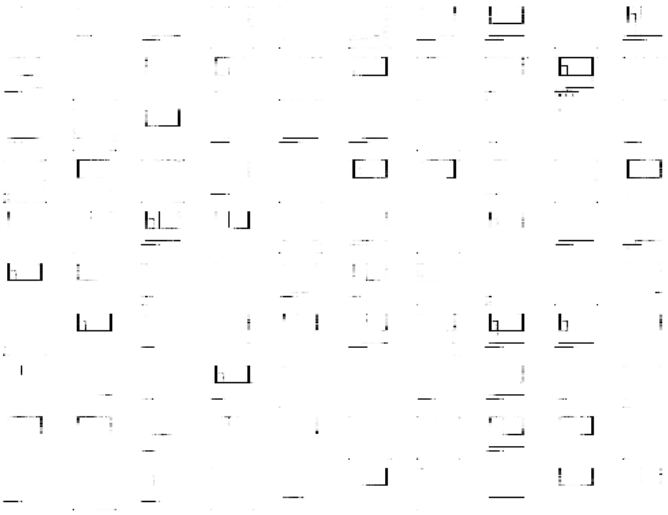
70ème itération



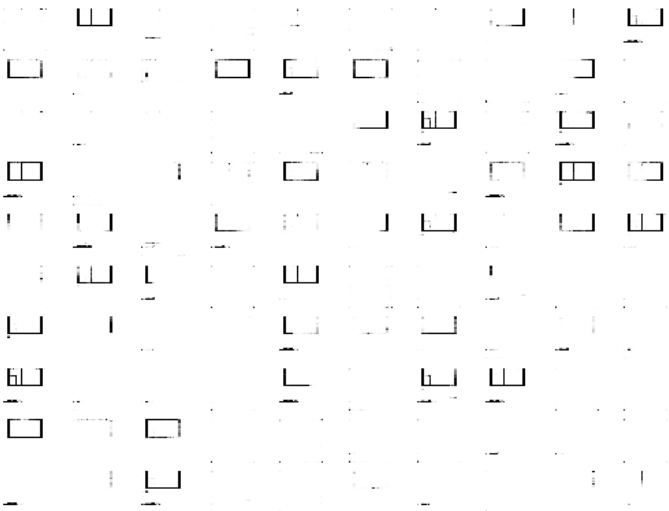
140ème itération



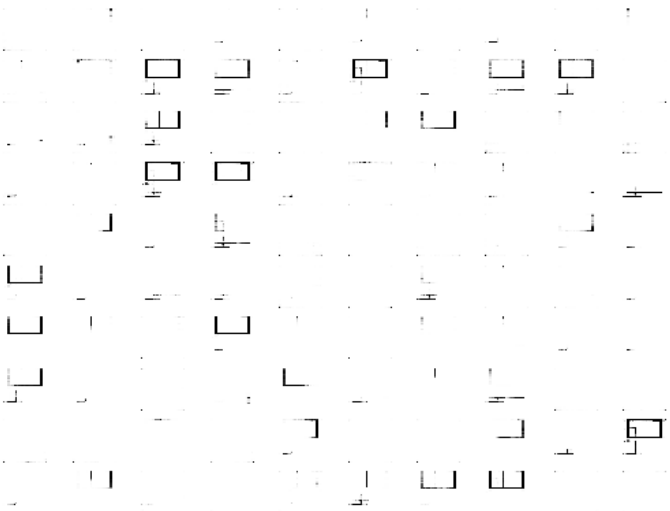
210ème itération



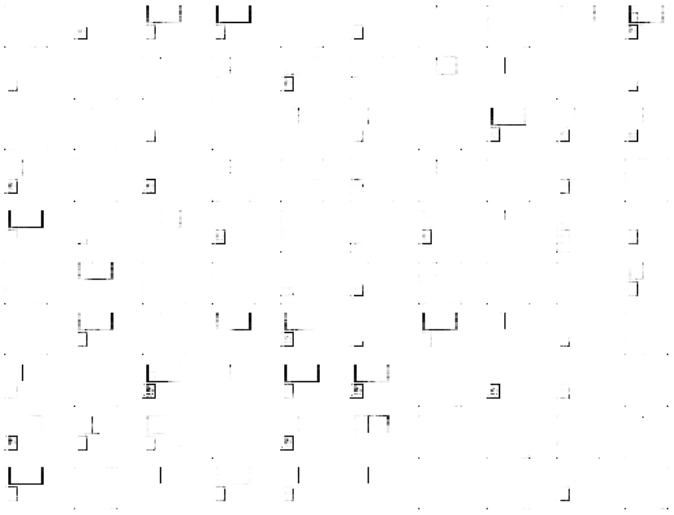
280ème itération



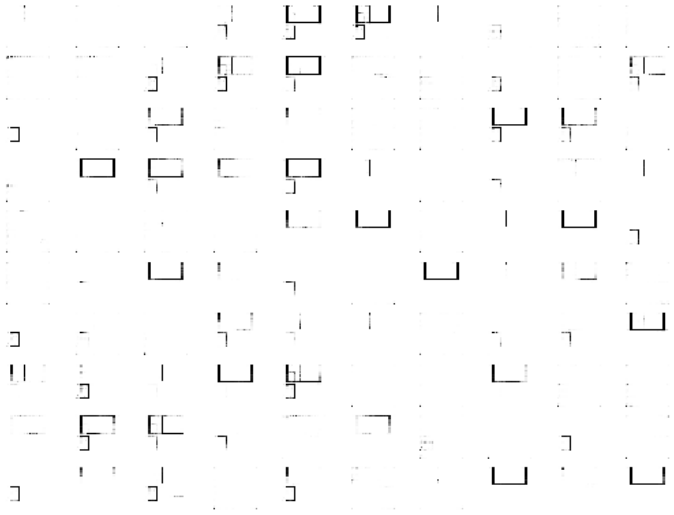
350ème itération



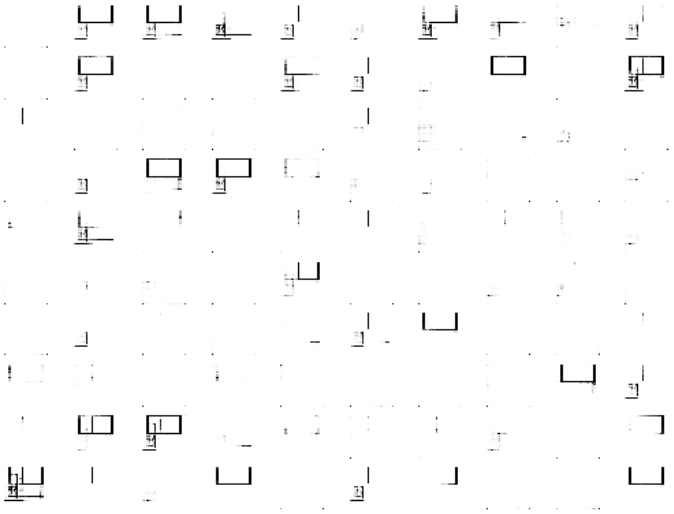
420ème itération



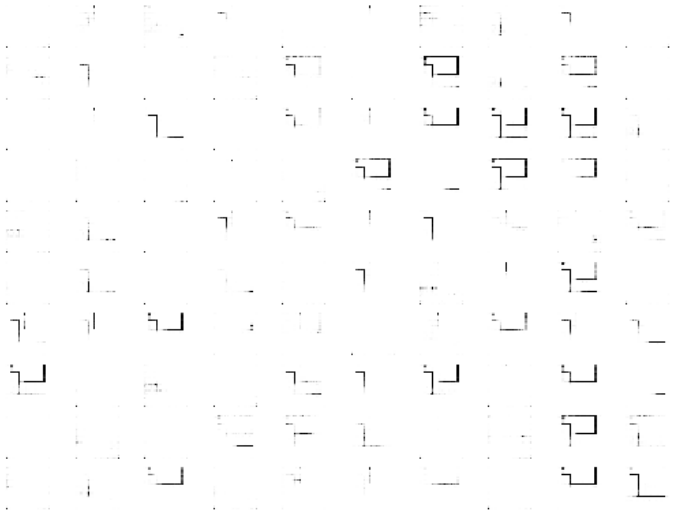
490ème itération



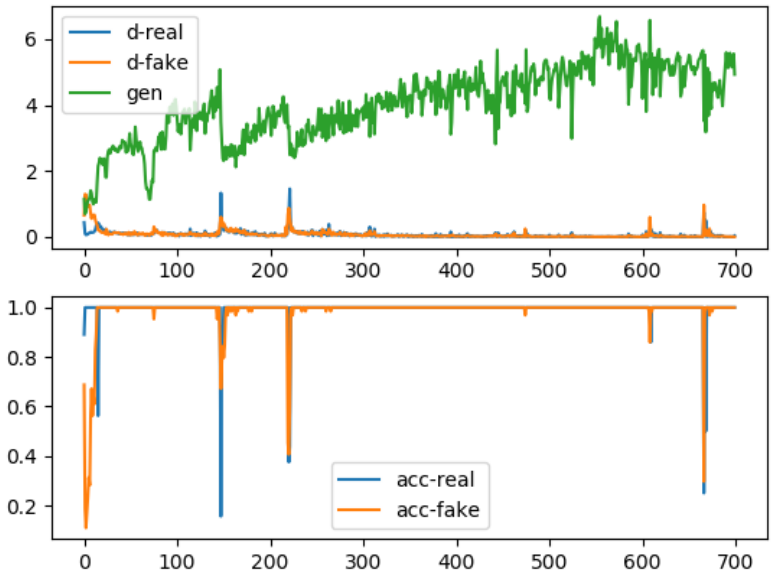
560ème itération



630ème itération

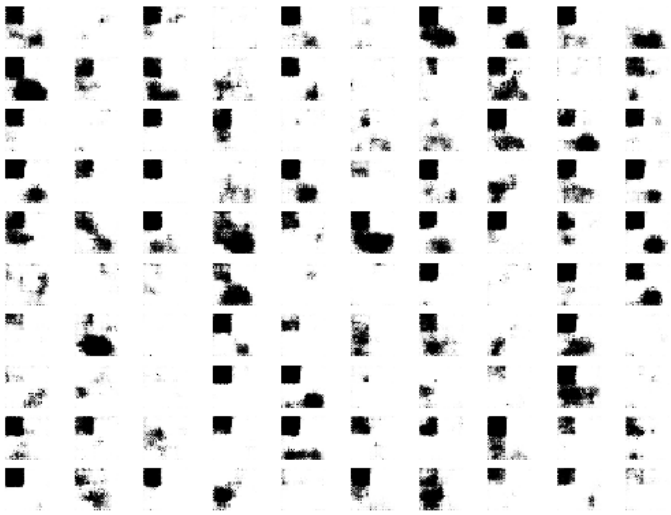


700ème itération

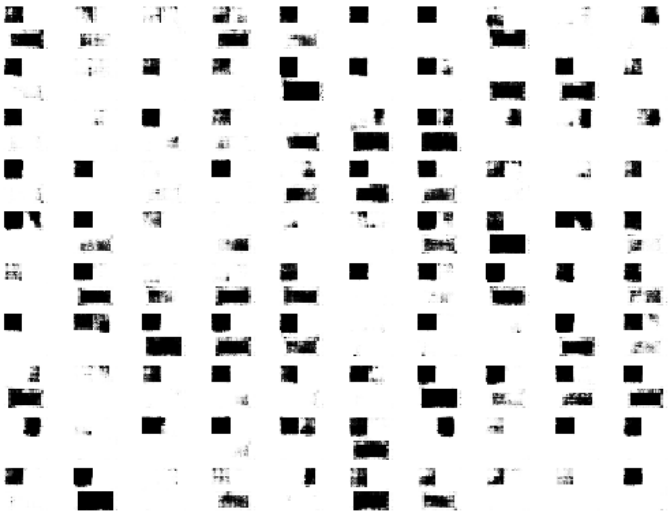



Courbes statistiques associées.

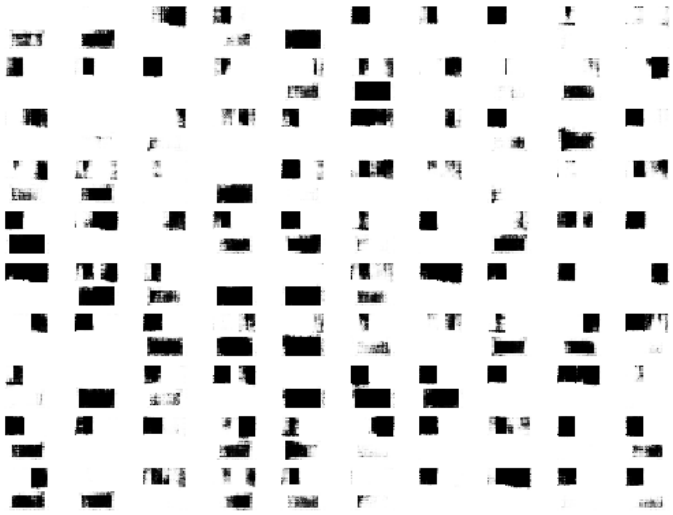
*9.15 Résultats du **GAN** sur les dispositions, formes pleines noir sur blanc. (Source : auteur)*



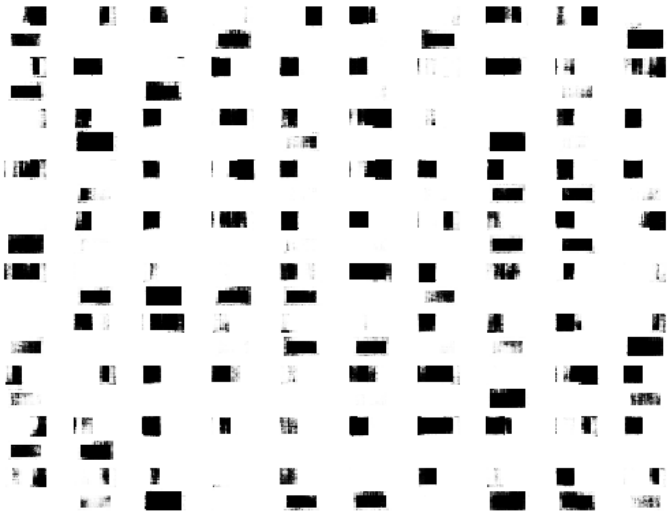
70ème itération



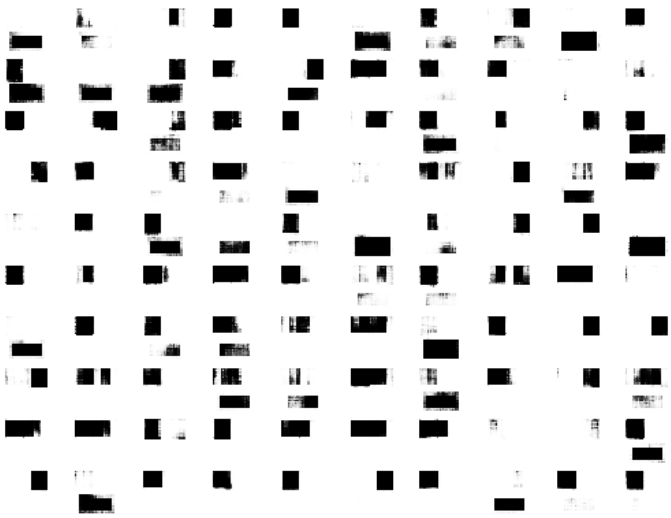
140ème itération



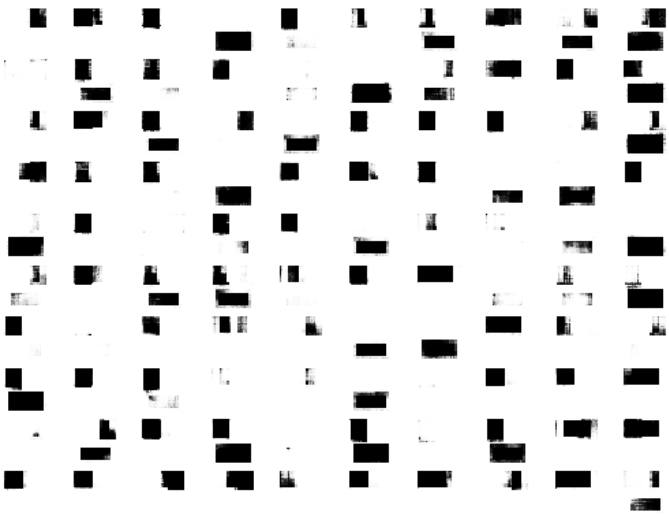
210ème itération



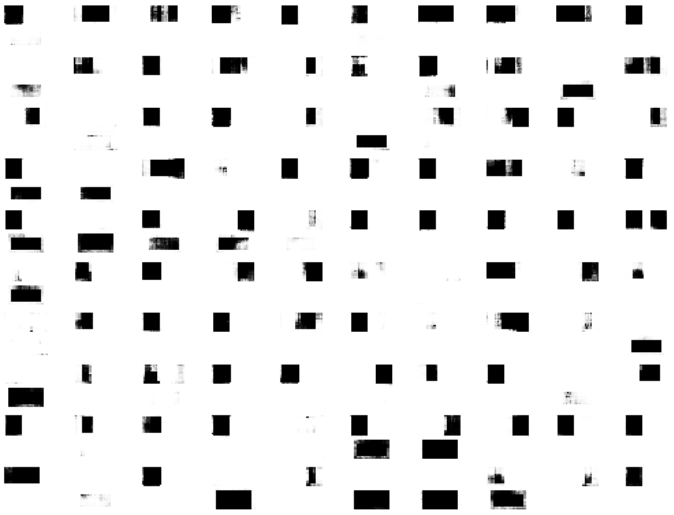
280ème itération



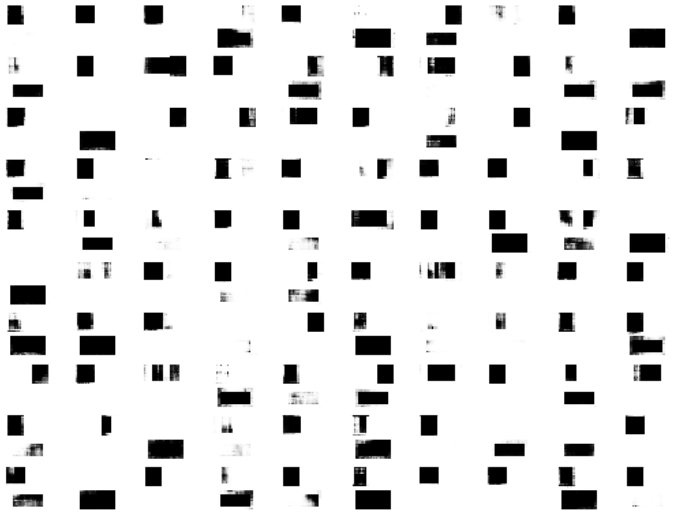
350ème itération



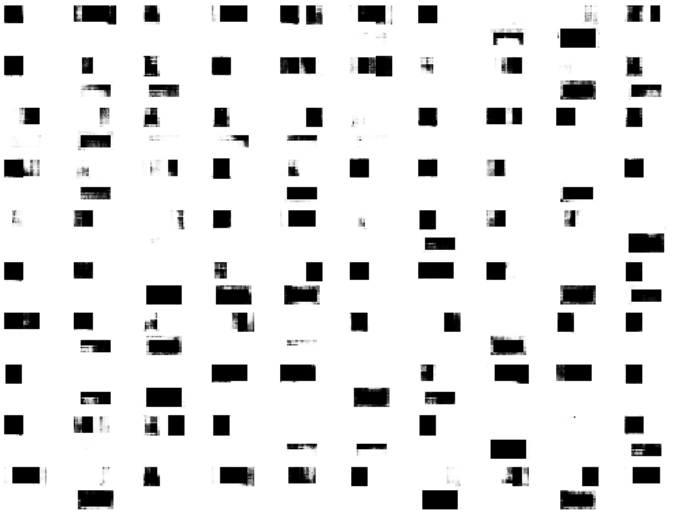
420ème itération



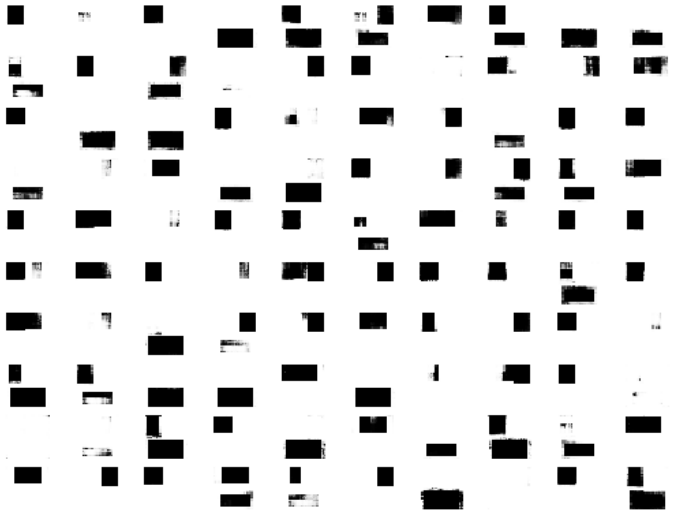
490ème itération



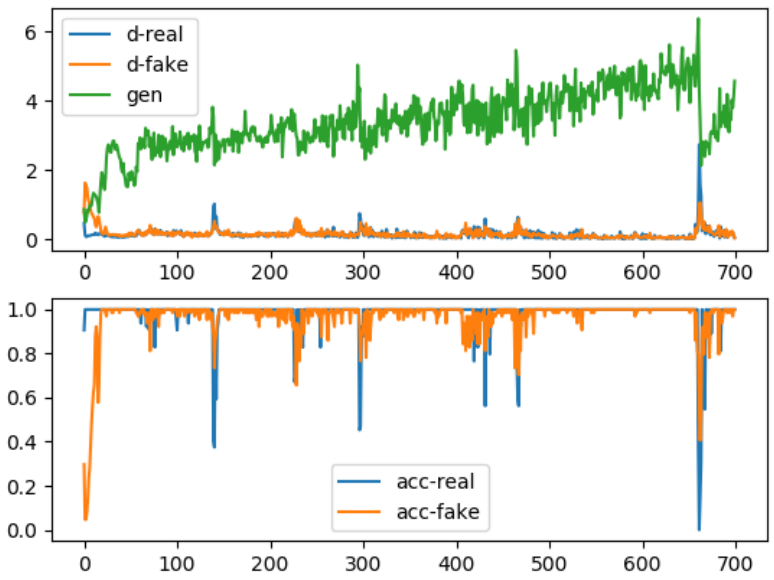
560ème itération



630ème itération

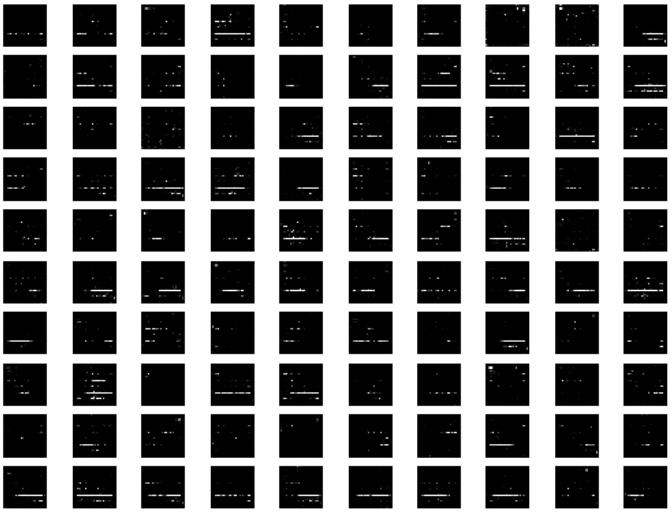


700ème itération

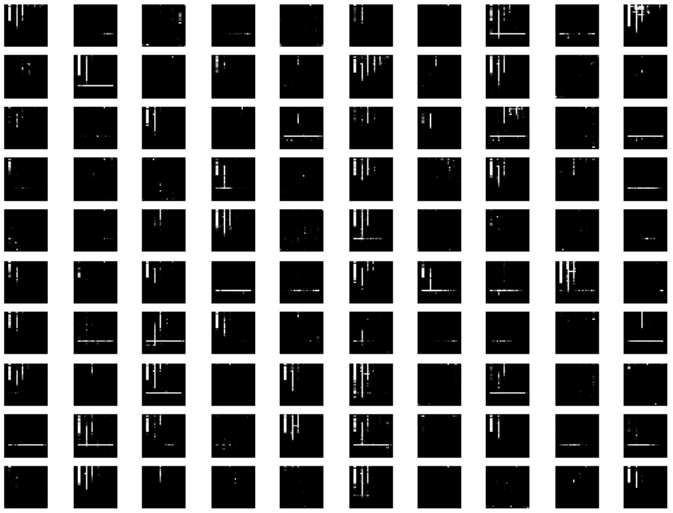



Courbes statistiques associées.

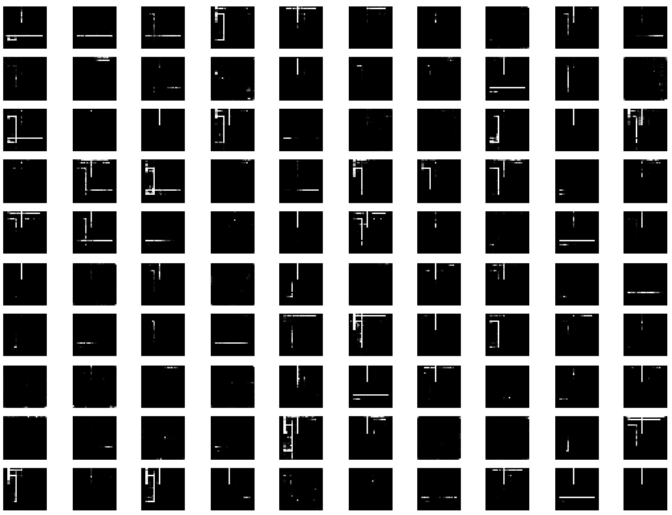
*9.16 Résultats du **GAN** sur les dispositions, formes vides blanc sur noir. (Source : auteur)*



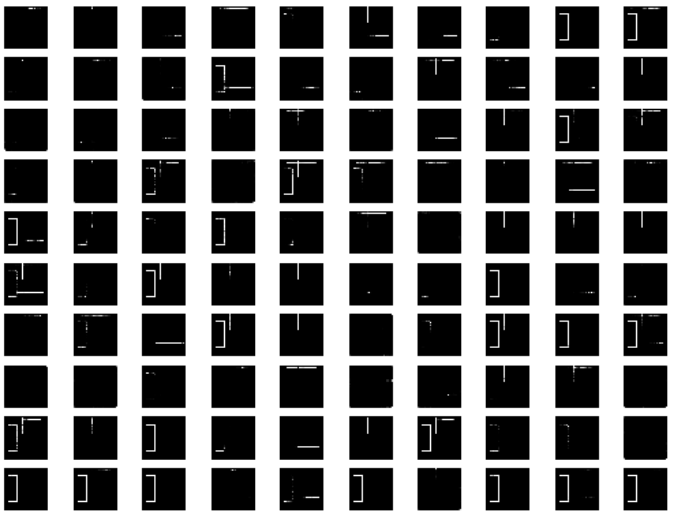
70ème itération



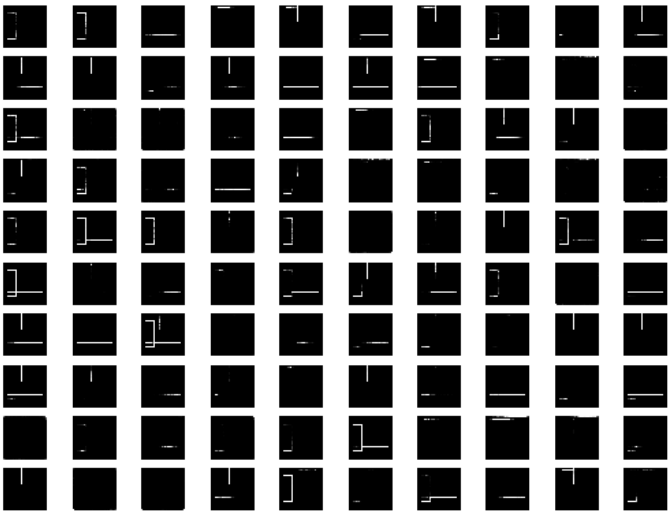
140ème itération



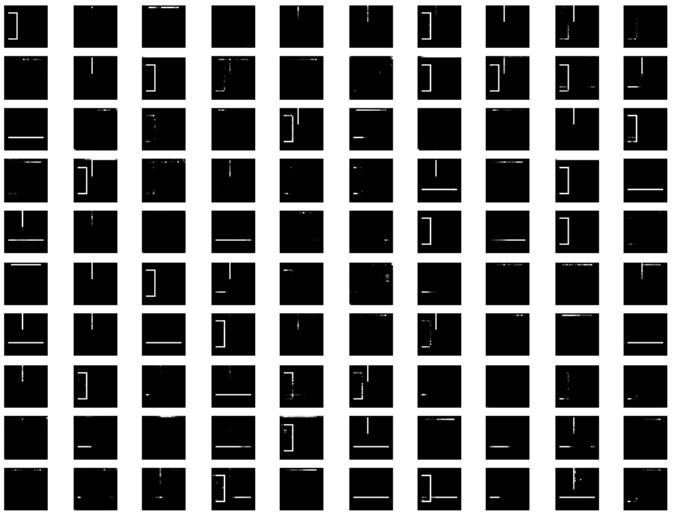
210ème itération



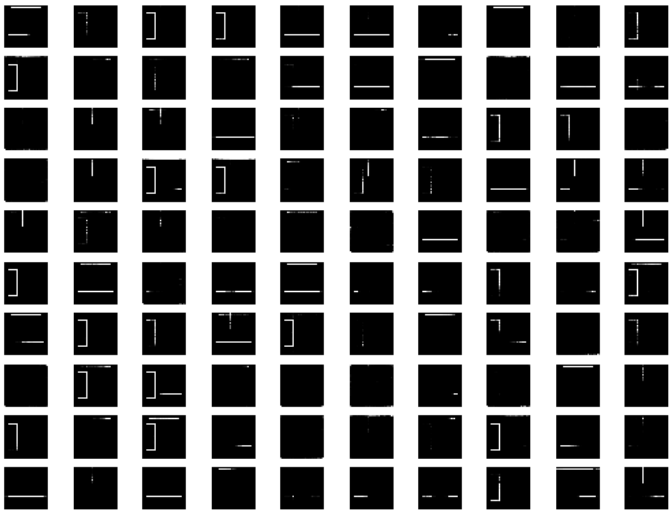
280ème itération



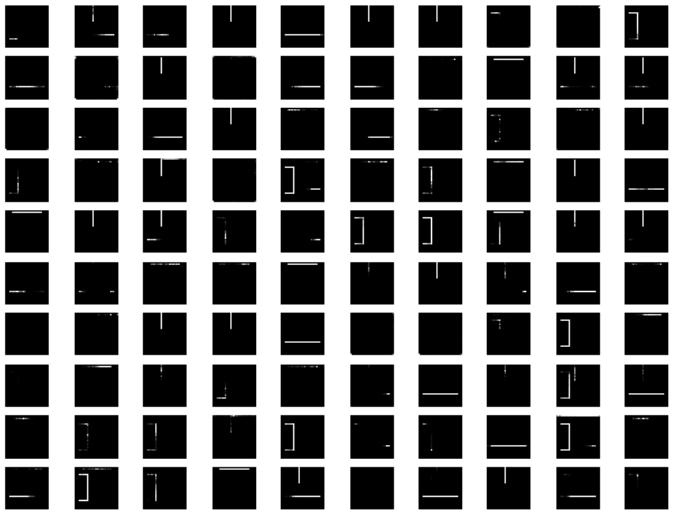
350ème itération



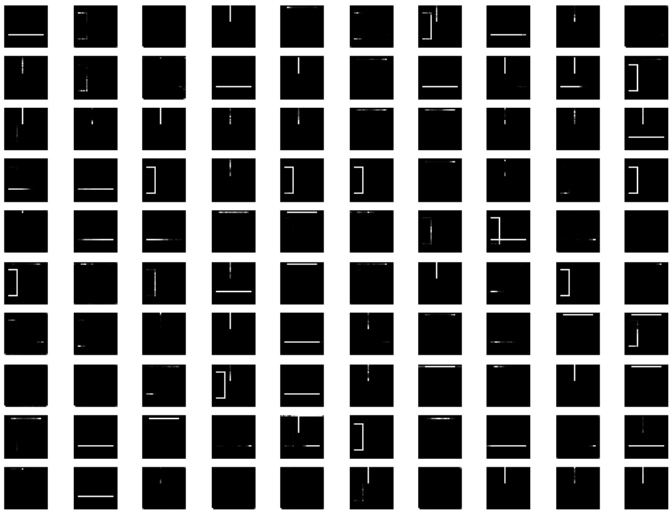
420ème itération



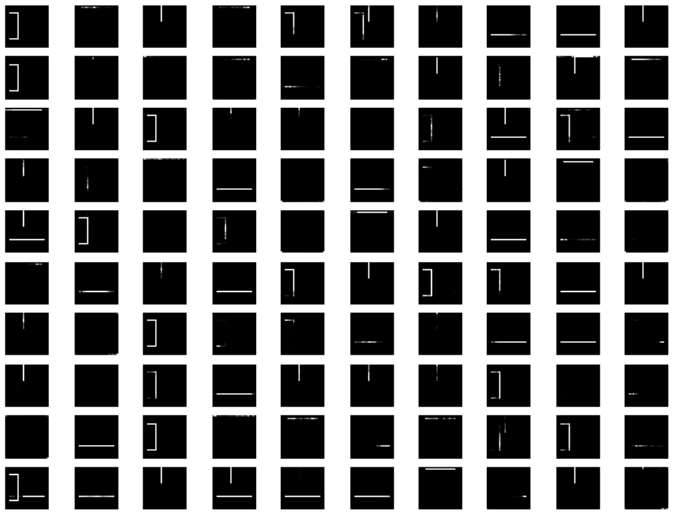
490ème itération



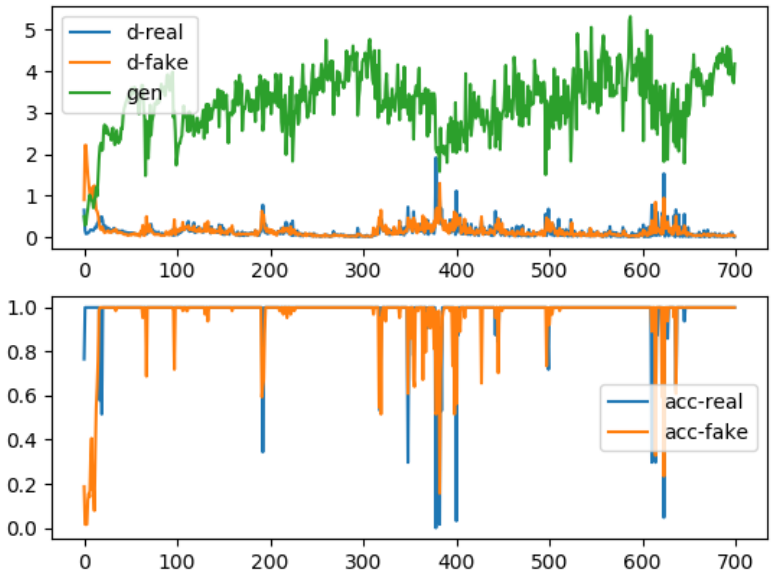
560ème itération



630ème itération

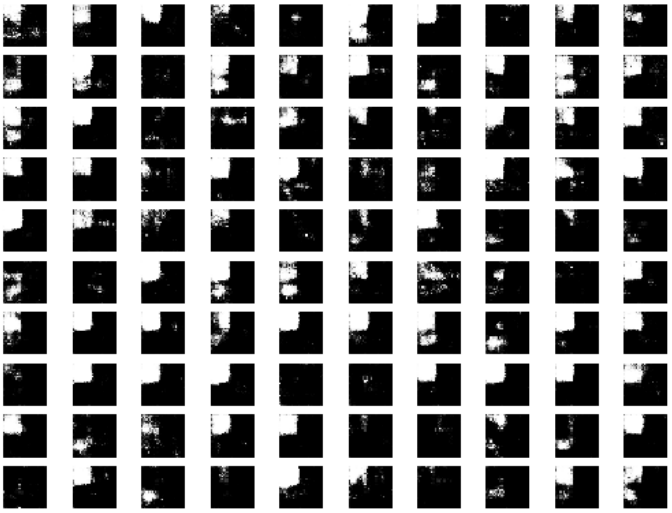


700ème itération

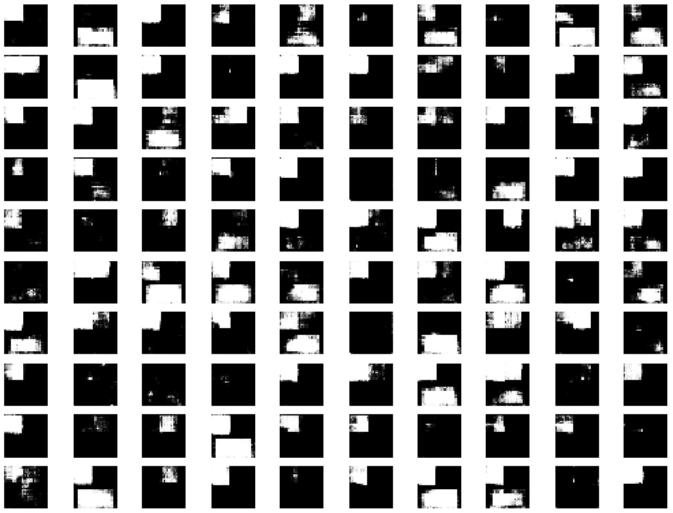



Courbes statistiques associées.

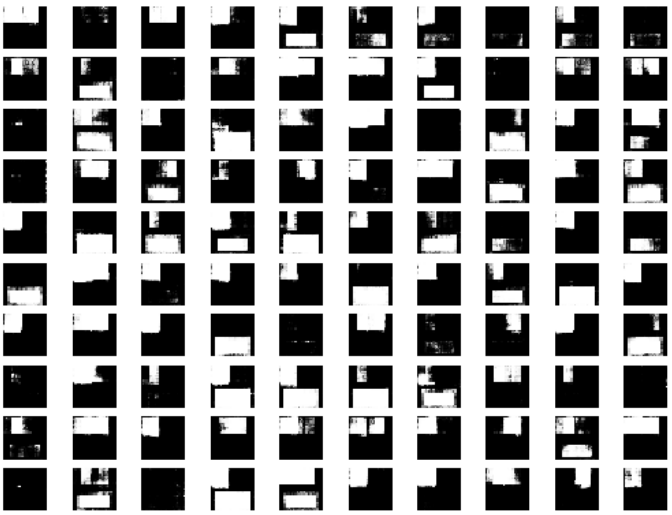
*9.17 Résultats du **GAN** sur les dispositions, formes pleines blanc sur noir. (Source : auteur)*



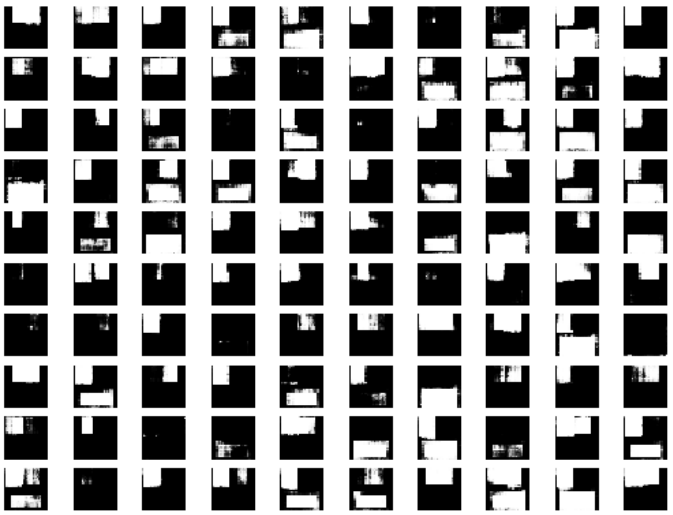
70ème itération



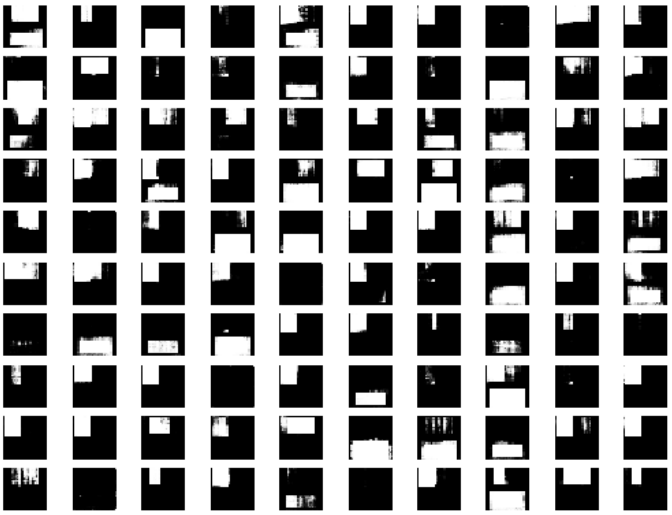
140ème itération



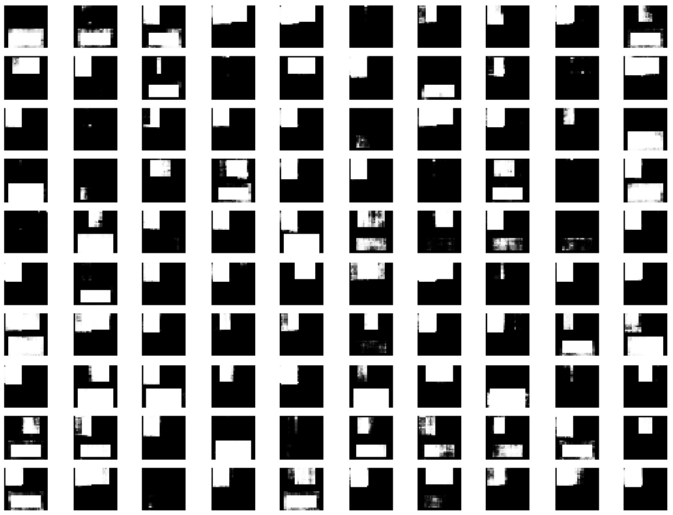
210ème itération



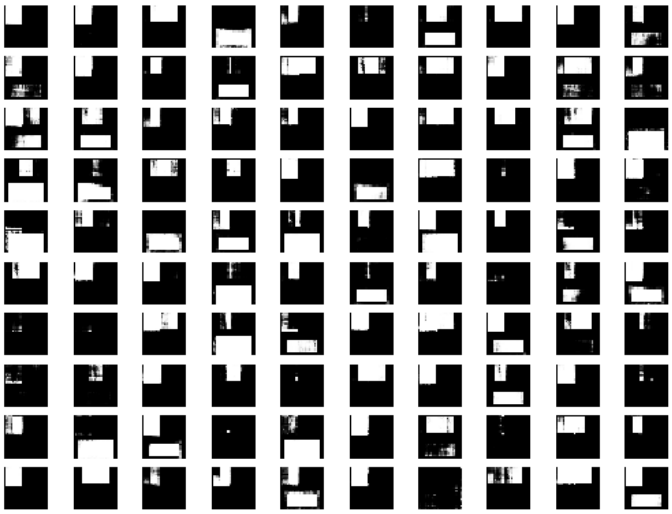
280ème itération



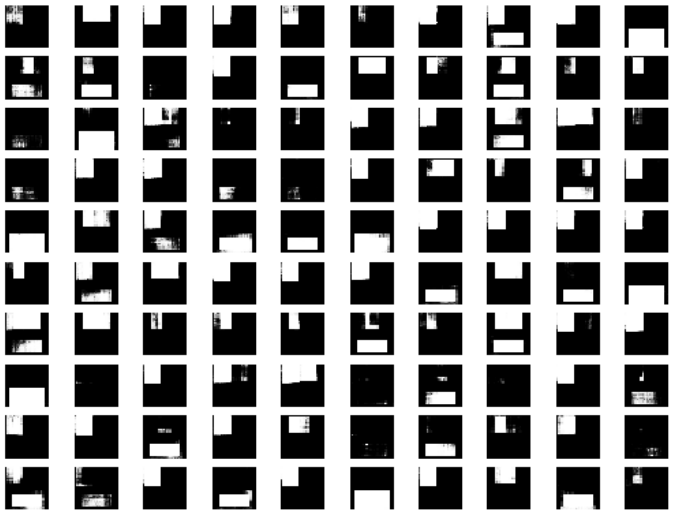
350ème itération



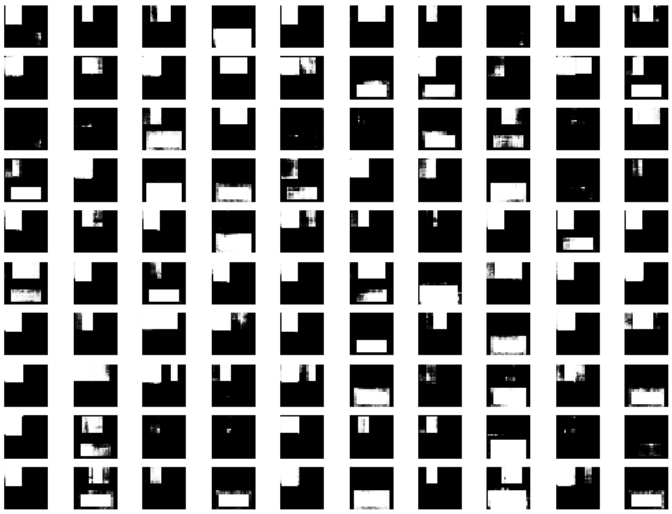
420ème itération



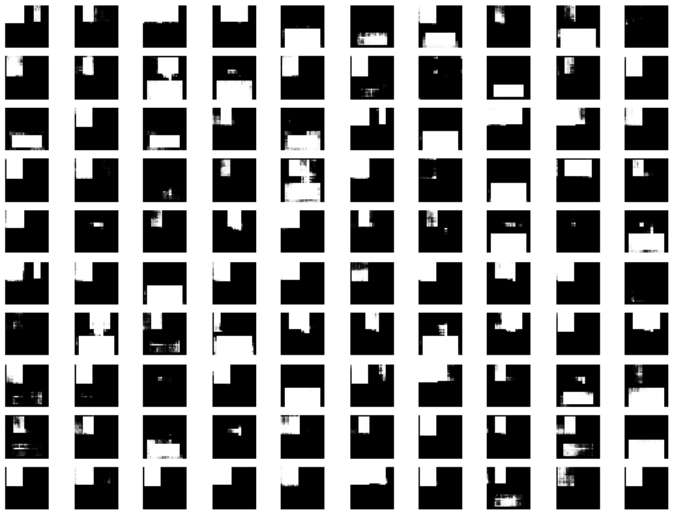
490ème itération



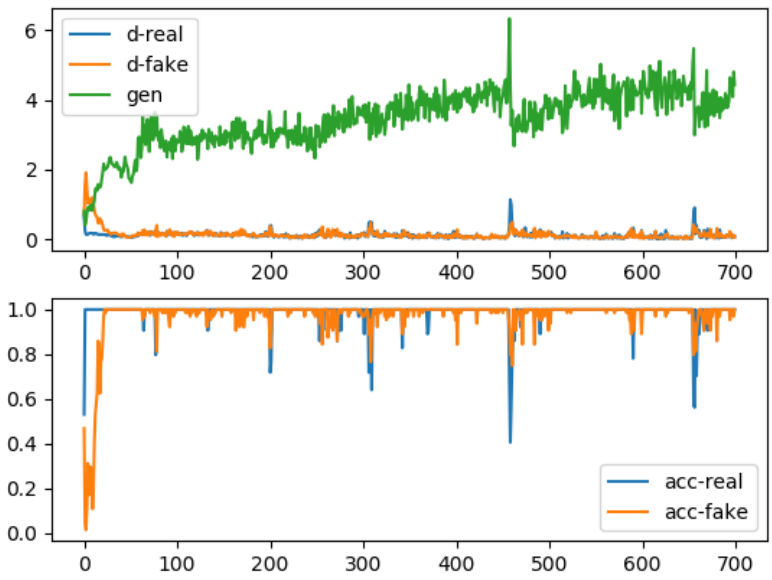
560ème itération



630ème itération



700ème itération




Courbes statistiques associées.

# RÉSUMÉ

Dans un contexte de déploiement de technologies de plus en plus lourdes dans le milieu architectural, ce mémoire touche à une des plus énigmatiques et magique d'entre elles, l'apprentissage machine. Reprenant les logiques d'apprentissage et d'adaptation du cerveau de l'Homme, cet outil présente un réel potentiel pour l'architecte comme pour l'architecture.

Malgré des avantages évidents, plusieurs limites justifient que son intégration est encore difficile, voire même controversée. Tout d'abord d'un point de vue fonctionnel, car l'apprentissage machine requiert de grande quantité de données pour fonctionner correctement, et le contexte actuel de la donnée en architecture est loin du libre-échange.

La question de l'éthique se pose également. Plus un outil devient puissant, plus la place de celui qui l'utilise se voit transformée, voir fragilisée. Garder le contrôle est essentiel, afin de préserver l'essence de l'architecture et de ses apôtres.

Ce mémoire approche ainsi une manière de recourir à l'apprentissage machine dans le processus de conception, par le biais de la création de données et de leur enrichissement via des réseaux de neurones antagonistes génératifs. Le système avancé est ouvert, et prêt à se nourrir des évolutions impétueuses de la technologie, de la société, et surtout de l'architecture.