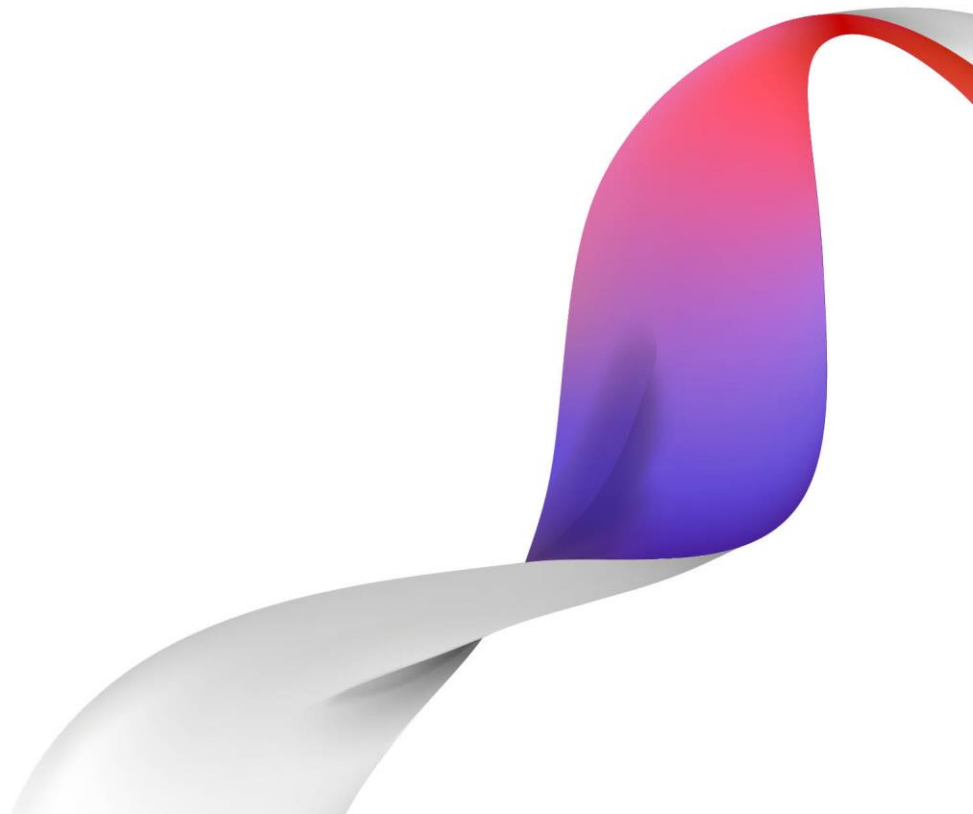


# Projet Annuel Big Data

3e année ingénierie du Big Data

**SIMON, Pierre**

juillet 24, 2018



# SOMMAIRE

<b>Introduction</b>	<b>3</b>
<b>Partie 1 : Recherche du plus court chemin</b>	<b>4</b>
Stack utilisé	4
Recherche locale naïve	5
Recuit simulé	6
Algorithme génétique	7
Dijkstra	8
A*	10
Dijkstra vs A*	11
<b>Partie 2 : Machine Learning</b>	<b>12</b>
Stack utilisé	12
Modèle linéaire : Perceptron	13
Radial Basis Function Network	15
Kernel machines choix du dataset	16
Framework de Machine Learning	17
Méthode de recherche du modèle	17
Evaluation d'un modèle	19
Critères de sélections	19
Modèle choisi	20
Comparaison avec notre lib	21
<b>Conclusion</b>	<b>22</b>

# Introduction

L'objectif du projet a été d'implémenter et d'adapter des algorithmes de résolution de problèmes d'optimisation. De réussir à implémenter et utiliser des modèles simples de Machine Learning et d'appliquer cela et les connaissances acquises en cours sur un jeu de données réel.

Le projet sera découpé en deux parties. La première partie sera d'implémenter des algorithmes de recherche du plus court chemin et de le démontrer sur des cas de tests. L'ensemble de ces algorithmes ont été implémentés en C# afin d'utiliser le moteur jeu Unity pour avoir une démonstration visuelle de la véracité des algorithmes. La seconde partie sera quant à elle d'implémenter des algorithmes et des modèles de Machine Learning. Toutes ces fonctions devront être réunies dans une librairie, ici rédigée en c++, afin de pouvoir l'utiliser dans plusieurs environnements. Dans un premier temps nous démontrerons le bon fonctionnement de chacun des modèles implémentés à l'aide de cas de tests dans Unity. Puis, nous les appliquerons sur un jeu de données réel et confronterons les résultats à des frameworks populaires de Machine Learning tel que Keras ou Scikit-learn.

Tout au long du projet nous avons utilisé Git afin de versionner l'avancement du projet, vous pourrez y trouver tout [le code sources](#).

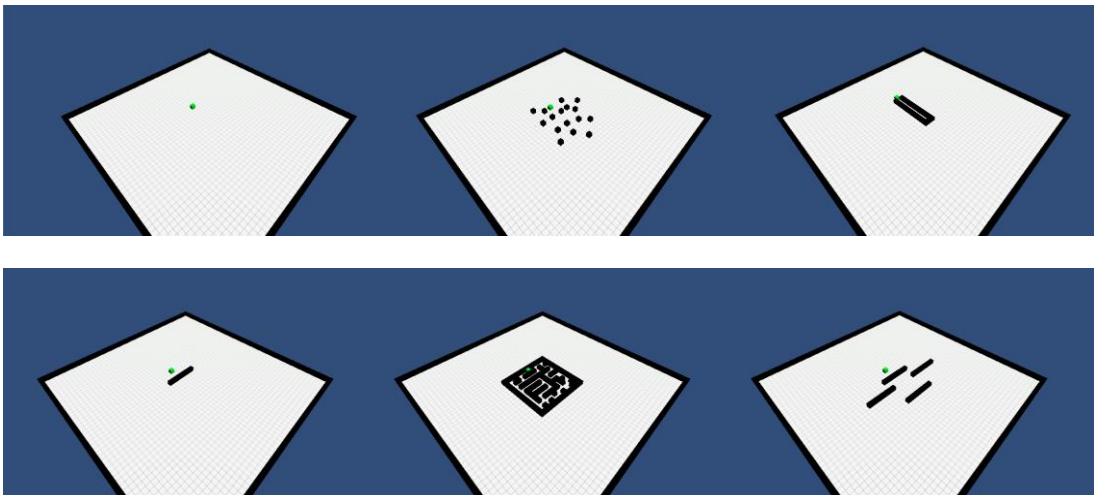
## Partie 1 : Recherche du plus court chemin

### Stack utilisé



Nous avons implémenté différents algorithmes de recherche du plus court chemin. Dans certains cas il s'appuie sur une heuristique. Ici nous avons pris le postulat d'utiliser la distance de Manhattan pour sa simplicité et son efficacité.<sup>1</sup>







Nous avons par la suite fait passer chacun des algorithmes sur six cas de tests :



---

<sup>1</sup> [Code source](#)

## Recherche locale naïve

PROBLEME	1	2	3	4	5	6
RESULTAT						

### Postulat

**Initialisation** : Génération d'un ensemble d'actions aléatoires qui forme un chemin.

**Modification** : Une action d'indices aléatoires par une nouvelle action aléatoire.







**Critère de sélection** : Le nouveau chemin est meilleur que l'ancien.

**Critère d'arrêt** : L'erreur est égale à l'erreur minimale.

### Valeur clé

**La taille du chemin** : c'est la valeur la plus influente, un chemin trop long va avoir du mal à tendre vers la solution et réciproquement. On obtient des résultats positifs à partir d'une taille de 10 et cela semble être optimal aux alentours de 50. Ce qui semble logique dû à la taille de la carte 50x50.

## Recuit simulé

PROBLEME	1	2	3	4	5	6
RESULTAT						

### Postulat

**Initialisation** : Génération d'un ensemble d'actions aléatoires qui forme un chemin.

**Modification** : Une action d'indice aléatoire par une nouvelle action aléatoire.







**Critère de sélection** : Si une valeur aléatoire entre 0f et 1f est inférieure au critère de Métropolis soit l'exponentiel de la différence entre l'erreur de l'état précédent et l'erreur de l'état courant divisé par la température alors on prend le nouveau chemin. Si l'erreur du nouveau chemin est toujours la même on incrémente la stagnation sinon on la remet à 0. Si la stagnation atteint un certain seuil on initialise la température à 6f par exemple et on remet la stagnation à 0, dans le but de sortir d'un minima local. Sachant que la température influe sur le critère de Métropolis. Et à chaque itération on décrémente la température.

**Critère d'arrêt** : L'erreur est égale à l'erreur minimale.

### Valeur clé

**La taille du chemin** : idem que la recherche locale naïve. La stagnation : car cela définit le seuil de tolérance à un minima local. Trop grande et on reste trop longtemps sur une même solution, et trop bas on quitte trop vite le minima trouvé qui pourrait être le bon. La température : influe sur la probabilité de bouger vers une meilleure nouvelle solution, et la probabilité de bouger vers une solution plus mauvaise est réduite en même temps que la température diminue.

# Algorithme génétique

PROBLEME	1	2	3	4	5	6
RESULTAT						

## Postulat

**Initialisation** : Génération d'un ensemble d'individus où un individu est un chemin de taille aléatoire.

**Evaluation** : Associe à chaque chemin un score.

**Sélection** : Sélectionne les meilleurs individus ~10% de la population.

**Croisement** : Deux parents aléatoires donnent un enfant qui a une action sur deux de chaque.

**Mutation** : On sélectionne aléatoirement une des actions et en génère une nouvelle.

**Critère d'arrêt** : Le score du meilleur fils est égal à l'erreur minimale.

## Valeurs clés






**La taille du chemin** : idem que pour les deux algorithmes précédents.

**La taille de la population** : une population trop faible tendra moins vite à trouver une solution, et une population trop forte sera trop lente à évaluer.

**Le critère de sélection** : s'il est trop grand il prendra des individus avec un score pouvant être très faible et impacter la génération suivante.

**Le pourcentage de mutation** : il ne doit pas être trop grand sinon cela impacte trop les individus fils et les dégrade

# Dijkstra

PROBLEME	1	2	3	4	5	6
RESULTAT						

## Postulat

**Initialisation** : Etiqueter tous les nœuds des graphes avec un score infini. Etiqueter le nœud de départ avec un score nul. Ajouter tous les nœuds à la liste L.

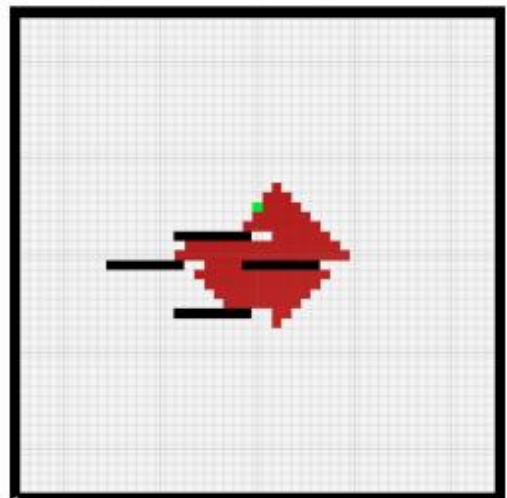
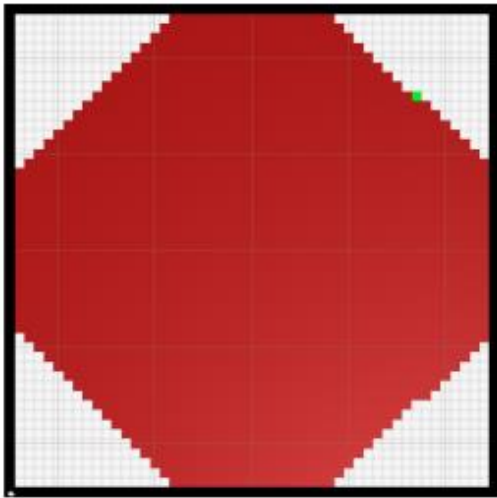
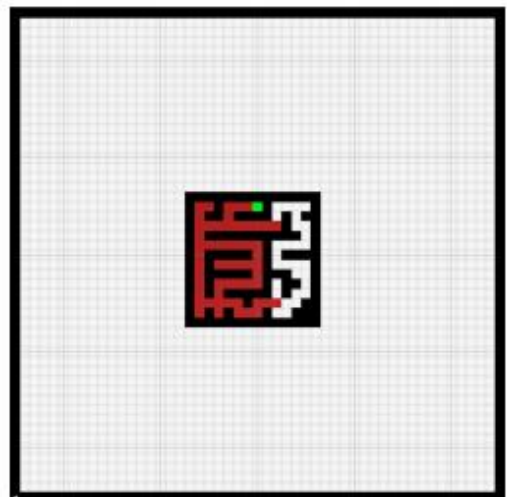
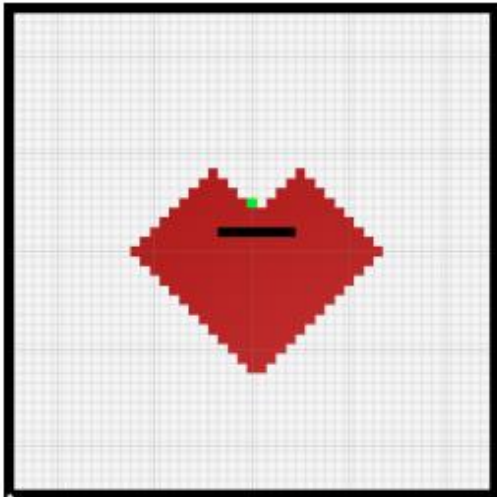
**Boucle** : Choisir le nœud de score le plus faible dans L : Nmin.

Si Nmin == Nœud destination -> fin et remontée.







Pour chaque fils de Nmin : Mettre à jour le score selon la formule  $\text{score}(\text{Nmin}) + \text{Cout}(\text{Nmin} \rightarrow \text{Voisin})$  à condition que le résultat soit plus faible que  $\text{score}(\text{Voisin})$ .

Retirer Nmin de L



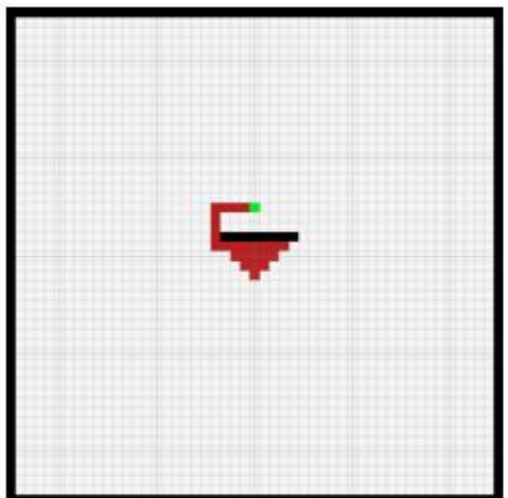
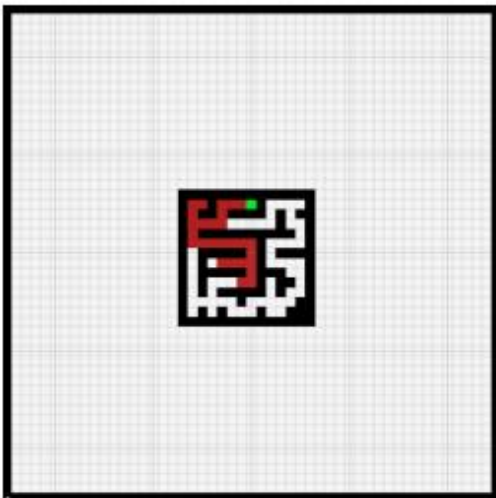
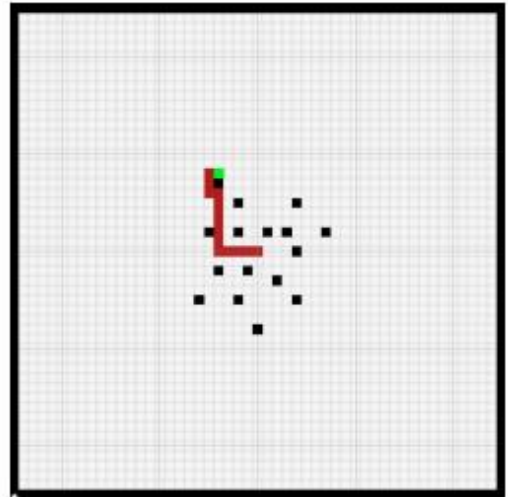
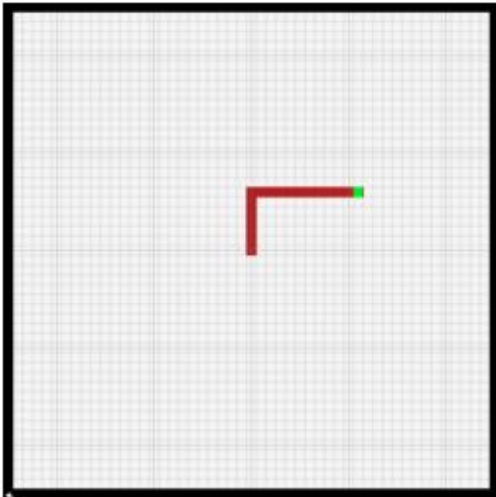


A\*

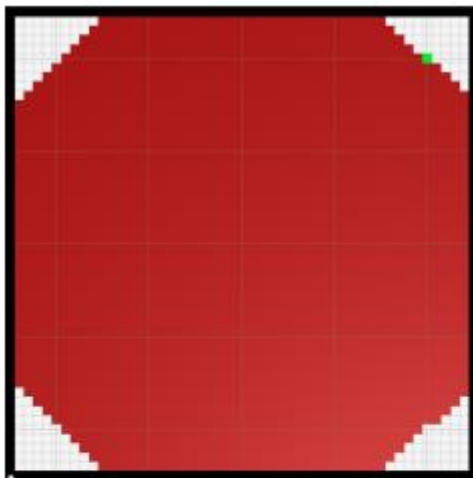
PROBLEME	1	2	3	4	5	6
RESULTAT						

### Postulat

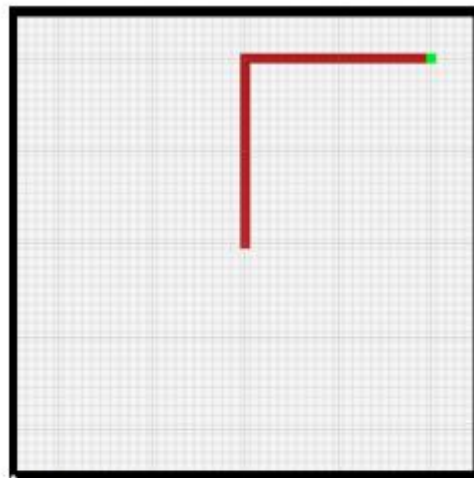
Identique que Dijkstra sauf qu'ici on ajoute la notion d'heuristique. On note chaque nœud avec celle-ci et on cherche le nœud nMin tel que  $\text{abs}(\text{score} - \text{heuristique score})$  soit le plus faible. L'heuristique choisie ici sera la Distance de Manhattan.



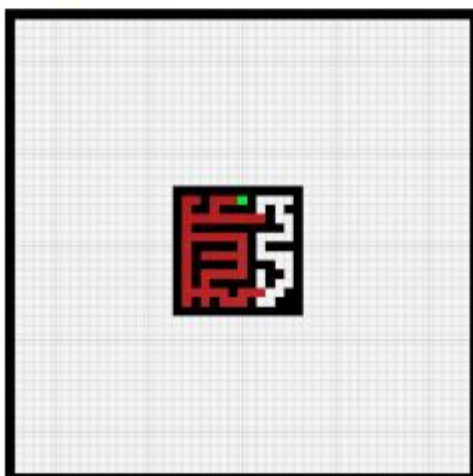
## Dijkstra vs A\*



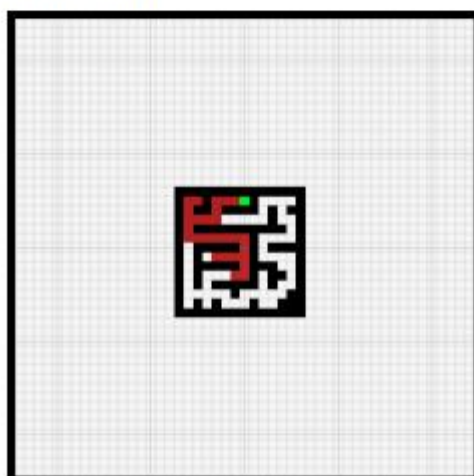
2229 itérations



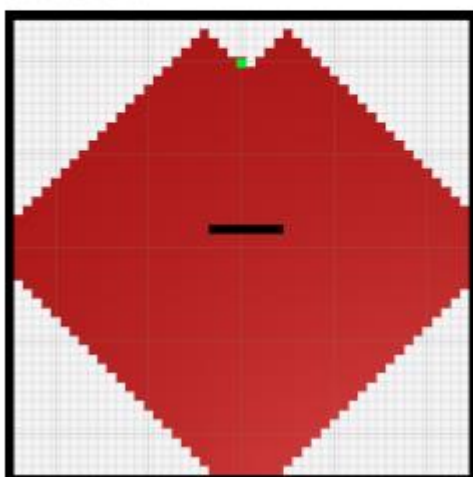
40 itérations



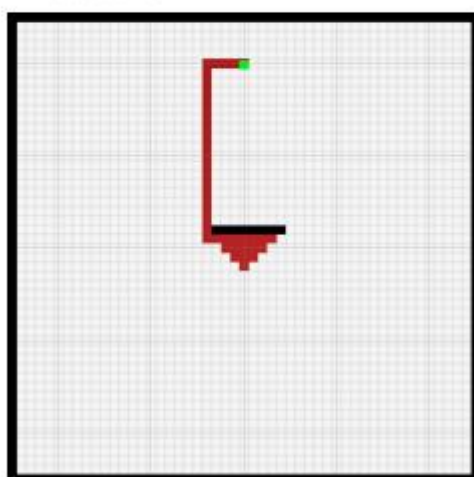
55 itérations



27 itérations



1482 itérations



39 itérations

## Partie 2 : Machine Learning

### Stack utilisé

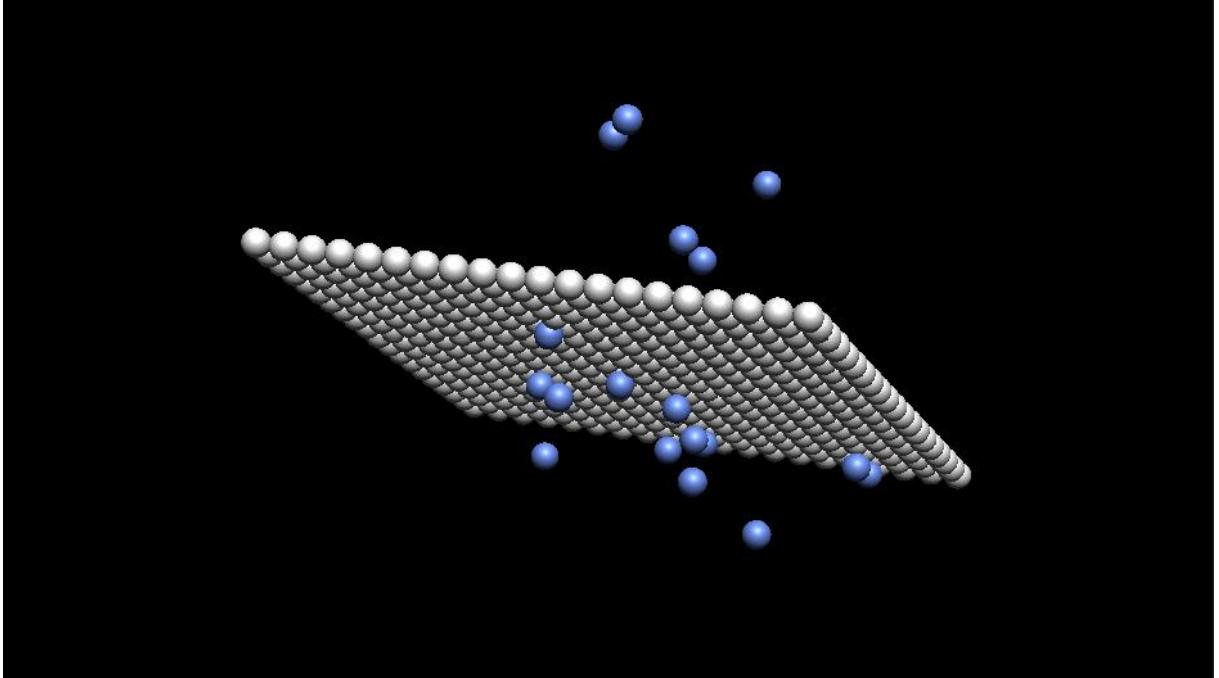


Ici nous allons démontrer le bon fonctionnement de nos algorithmes et modèles sur différents cas de test, toujours sur Unity afin d'avoir un résultat visuel.

## Modèle linéaire : Perceptron

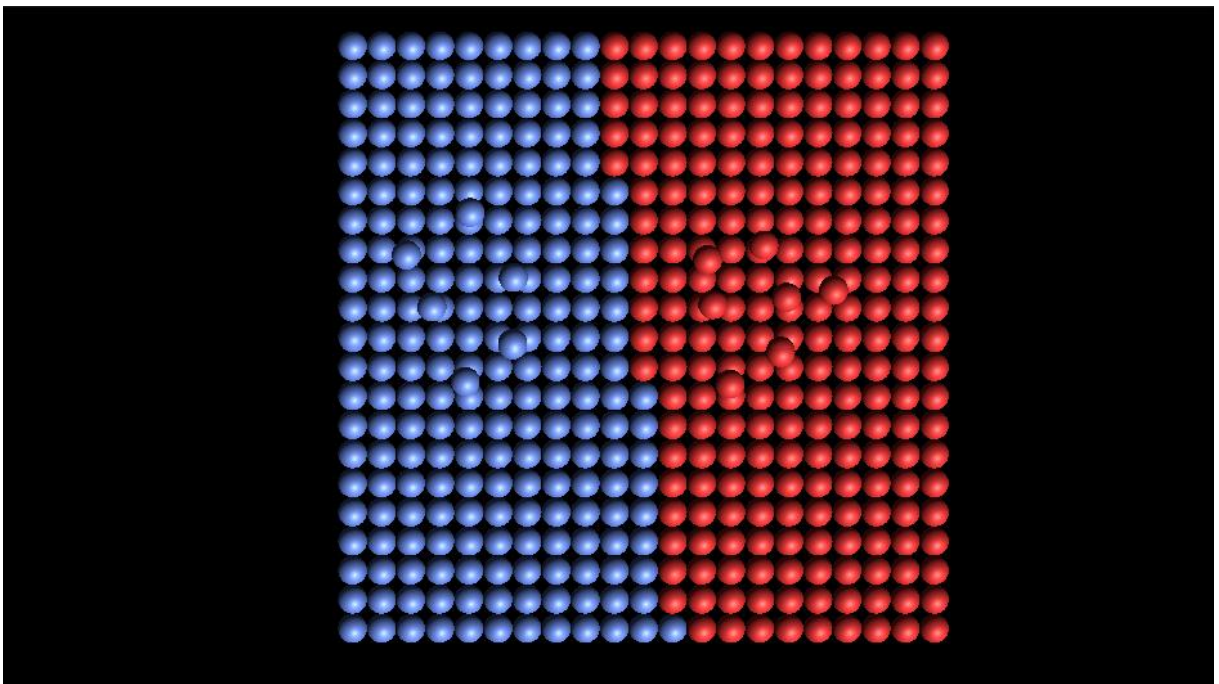
**Régression avec la Pseudo Inverse:** Etablir une relation entre les variables afin de pouvoir prédire de nouvelles données.

Possible lorsque les variables d'entrées ne sont pas trop éloignées.



**Classification** à l'aide de la règle de Rosenblatt et le signe de la somme des entrées en sortie.

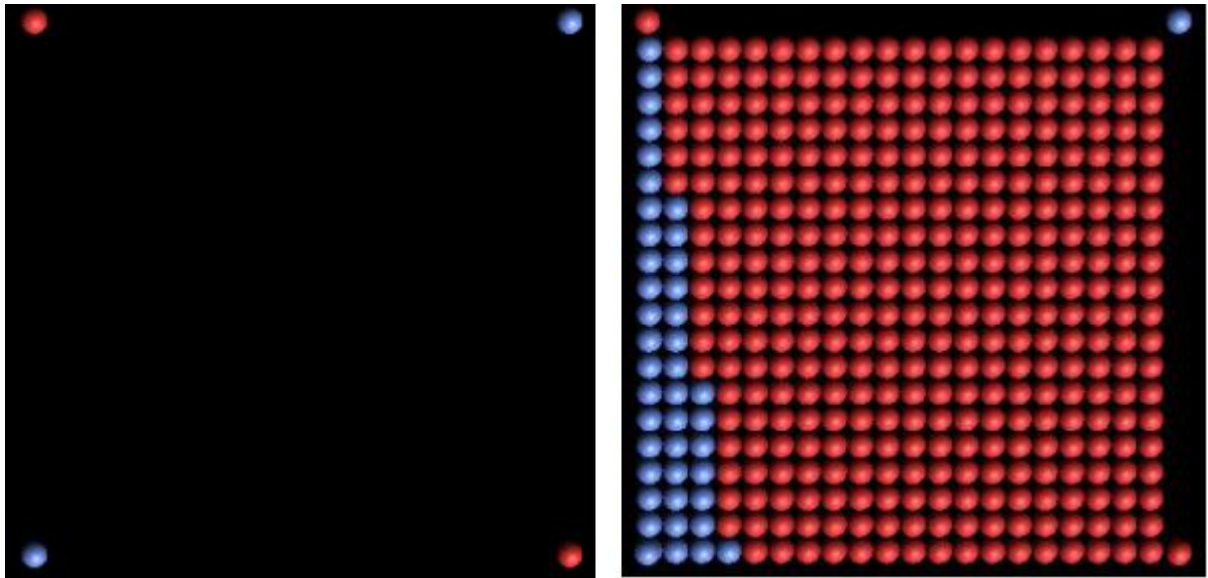
C'est un algorithme d'apprentissage supervisé de classificateurs binaires.





Cet algorithme ne fonctionne que dans le cas où il existe une séparation linéaire des données d'entrée.

Comme exemple de séparation non linéaire il y a le cas du XOR:

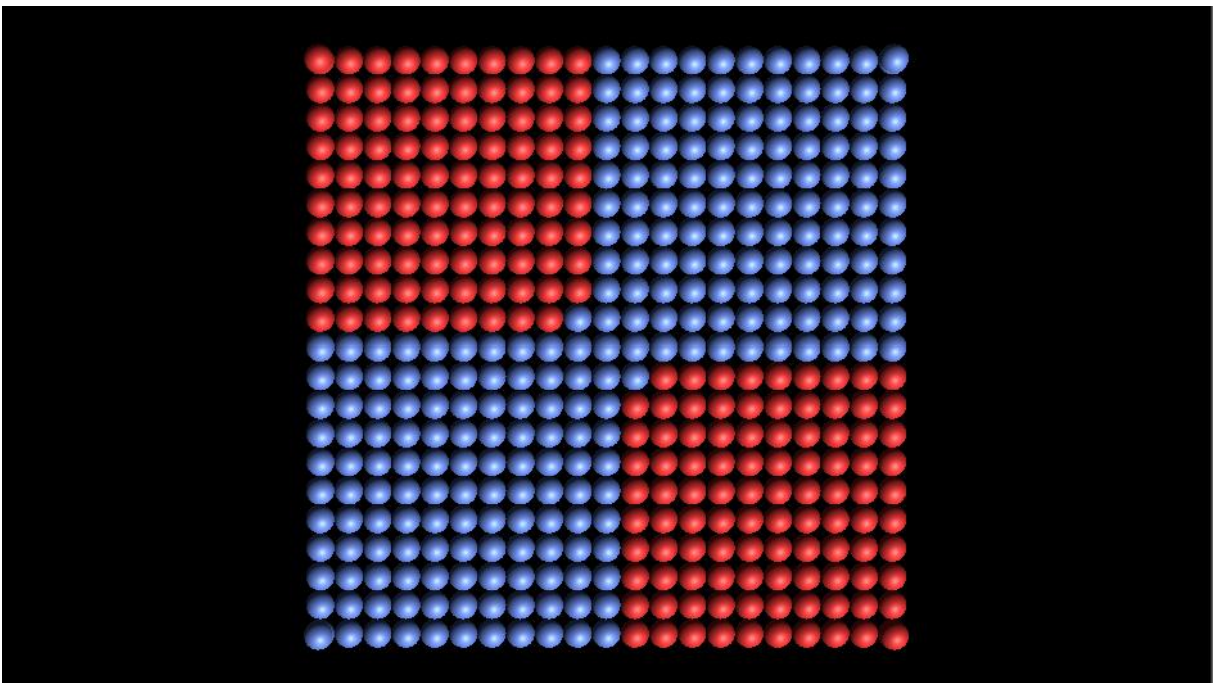


Méthode de transformation des entrées pour le cas du XOR

Ce cas est dit non linéairement séparable, mais dans un espace de dimension 2.

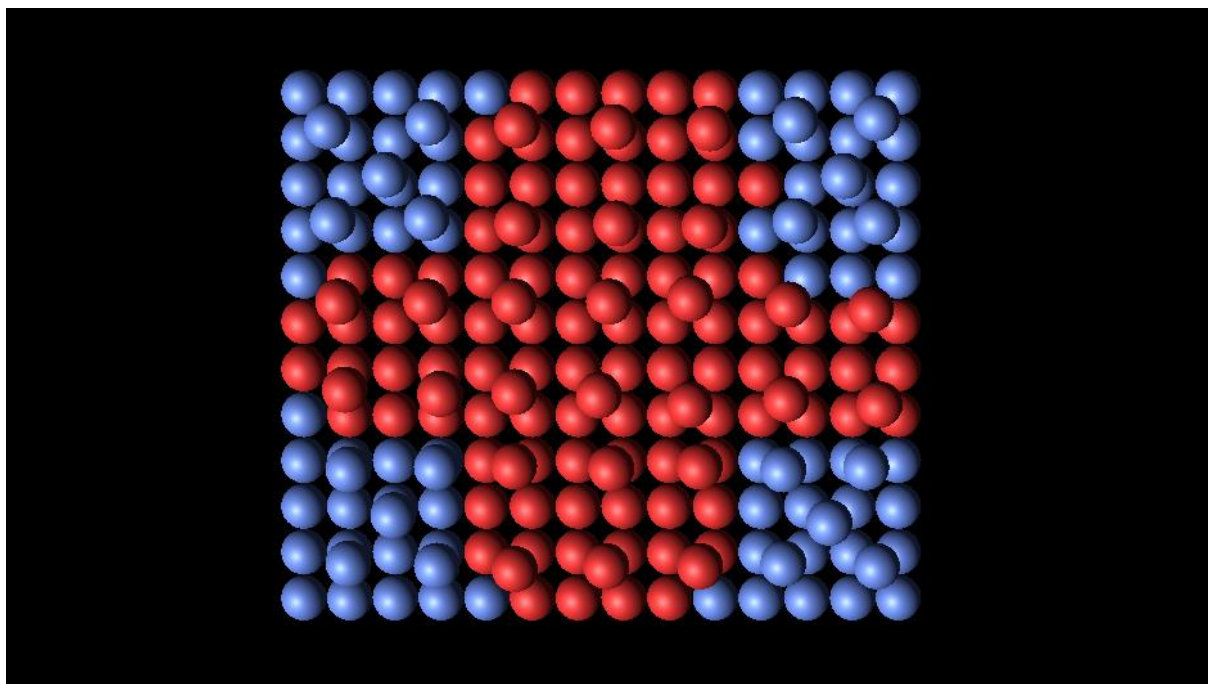
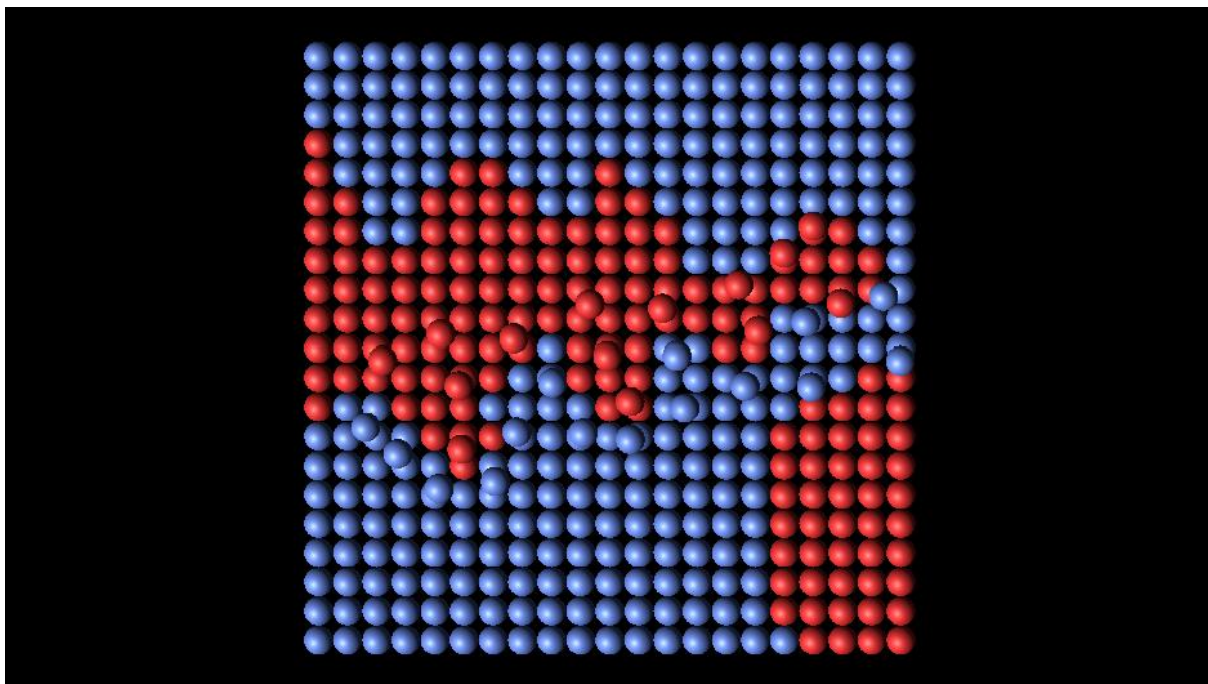
A l'aide de la fonction polynomiale  $(x, y) \rightarrow (x \times y)$  qui fait passer dans un espace de dimension

1 ce qui donne :

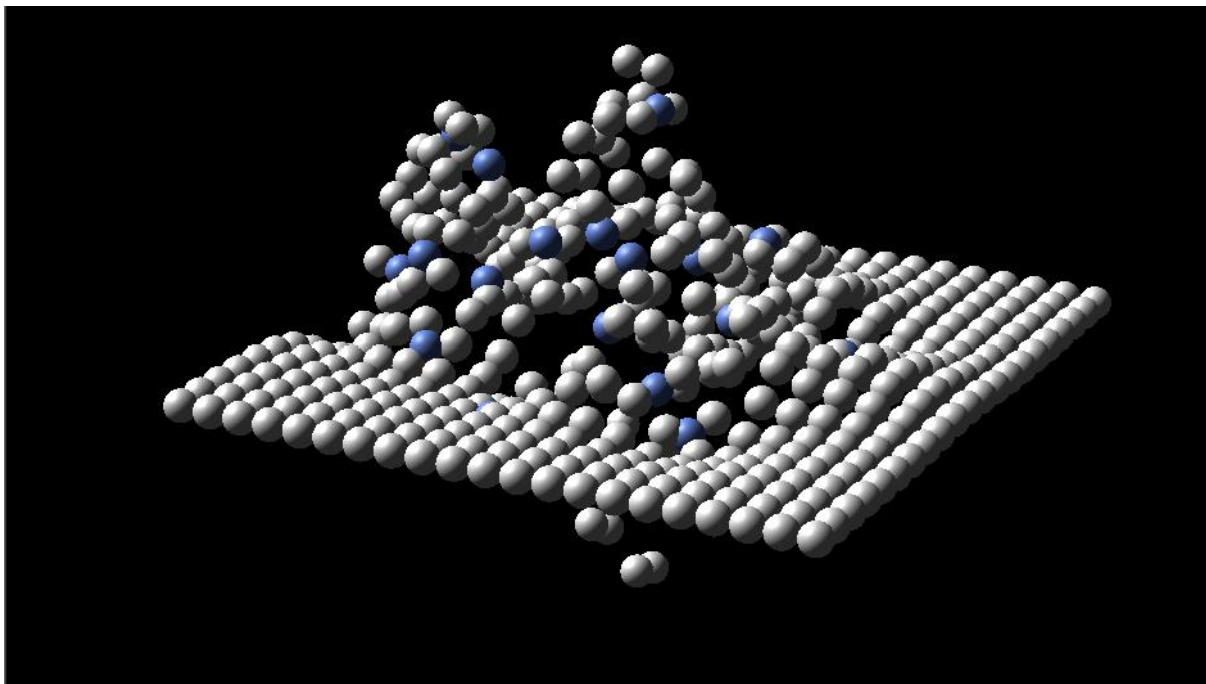


## Radial Basis Function Network

Classification:



## Régression:



## Kernel machines choix du dataset

Pour cette partie nous avons dû choisir deux jeux de données sur lesquelles nous allons faire tourner différents algorithmes de Machine Learning. Le premier porte sur la détection de fraude à la carte bancaire, il répondait aux critères de sélection qui sont un ratio de 10 entre le nombre de lignes et de colonnes. Le but était de prédire à partir des features la dernière colonne qui est 1 pour une fraude 0 si non. Le problème découvert lors de l'analyse préalable avant de lancer des algorithmes est que dans le jeu de données fournit la population de ligne positive à la fraude est inférieur à 0.02%.

Dans le doute de résultat cohérent et du temps restant nous avons préféré partir sur le second jeu de données.

Celui-ci porte sur la prédiction du genre d'une personne à partir de mesures obtenues de la voix. C'est donc une classification binaire (dans le cas où il n'y a que 2 genres possible ...). Le jeu de données est équilibré 50% lignes sont des hommes et 50 % des femmes. Le jeu de données est composé uniquement de réel, il n'y a aucun blanc ou donnée manquante et est composé de 20 features dont le label homme/femme qui deviendra une colonne de binaire 1 et 0 respectivement pour homme et femme pour un total de 3168 lignes.<sup>2</sup>

---

<sup>2</sup> Pour plus d'information sur la méthode d'analyse du jeu de donnée voir le [notebook](#) associé



## Framework de Machine Learning

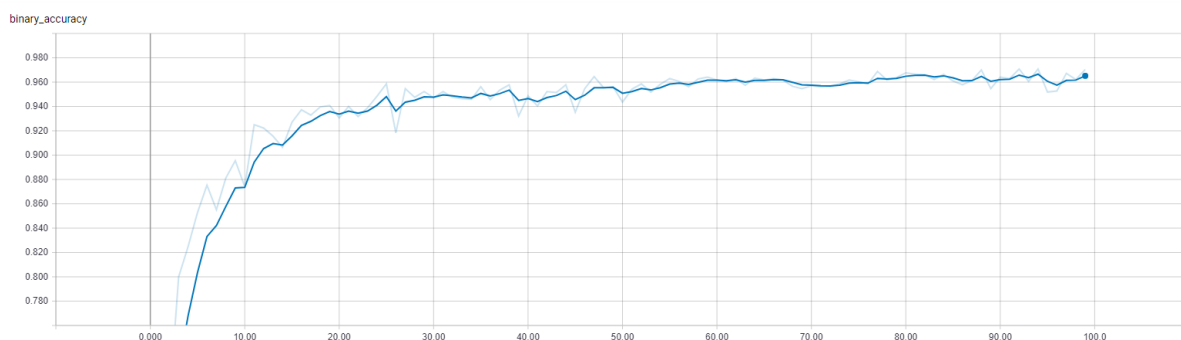
Nous avons utilisé un environnement virtuel à l'aide d'Anaconda sur lequel nous avons principalement utilisé Keras comme framework de Machine Learning, car il fournit une stack de modèle linéaire cohérent à celle implémenté dans notre lib. Comme backend nous avons utilisé Tensorflow. Cela nous permet de réaliser les calculs coûteux en terme de ressources sur GPU, d'avoir un monitoring de l'avancement de nos modèles sur Tensorboard. Nous avons également utilisé Jupyter pour fournir des livrables sous forme de notebook.

Ici nous parlerons du jeu de données<sup>3</sup>, du jeu d'entraînement qui représente 80% de celui-ci et du jeu de test qui représente les 20% restants. De plus par précaution nous avons mélangé le jeu de données avant de faire les séparations.

## Méthode de recherche du modèle

Dans un premier temps nous avons réalisé des tests à l'aveugle sur notre jeu de données.

Nous avons dans un premier temps fait varier les fonctions d'activation des couches



Ici nous utilisons la fonction d'activation « relu », et observons la binary accuracy qui représente l'évolution de notre modèle sur le jeu de test.



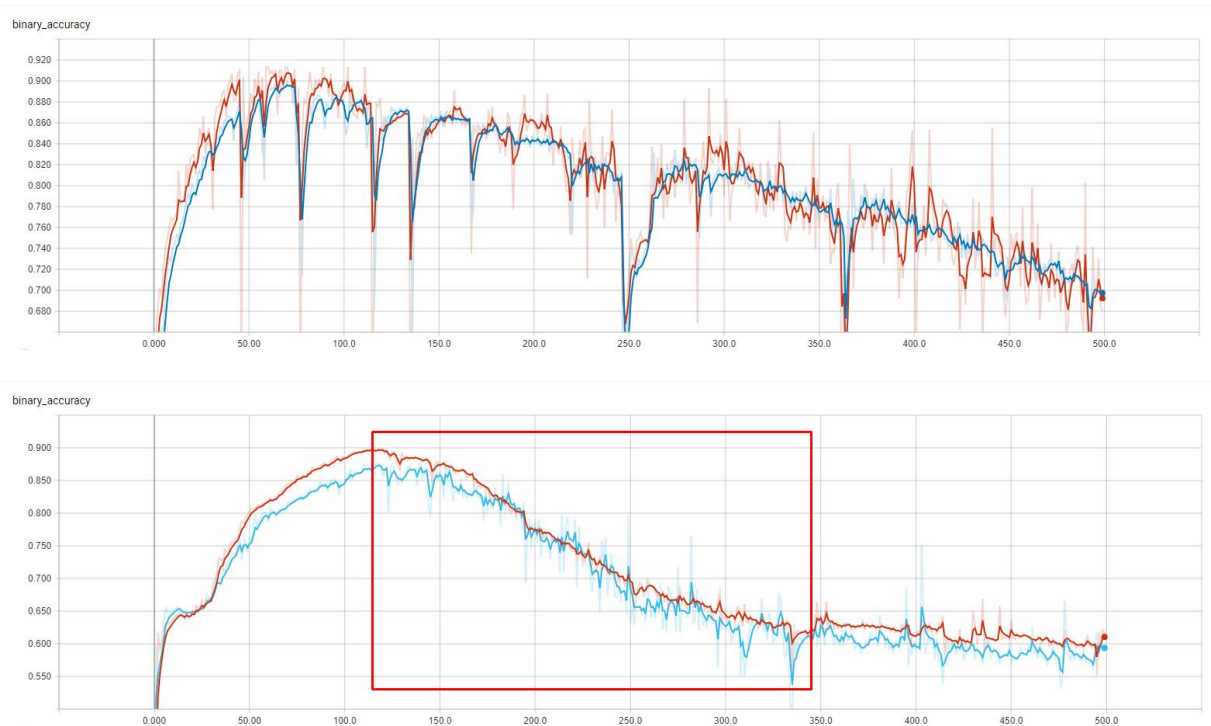
Ici nous utilisons la fonction d'activation « tanh ».

Nous pouvons observer ici deux choses : la vitesse à laquelle la courbe devient stable, et sa stabilité en général.

---

<sup>3</sup> <https://www.kaggle.com/primaryobjects/voicegender/home>

Puis nous avons fait varier le nombres de couches et le nombre de neurones par couche.



Avec un nombre de neurones trop faible nous avons du sous apprentissage.



Et inversement avec un nombre trop élevé de couche nous avons du surapprentissage.

A noté que comme vu lors de l'analyse de notre jeu de donnée, la répartition homme et femme est équilibrée. Alors même dans un cas de surapprentissage où la courbe représentant le jeu

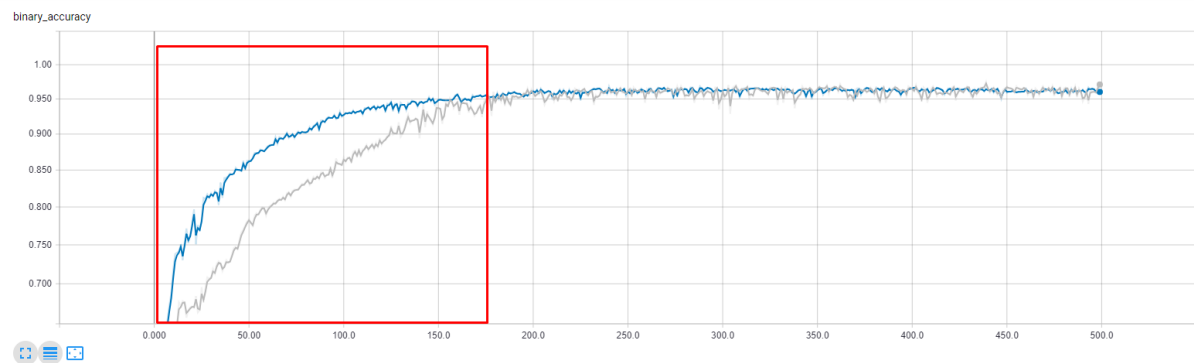
d'entraînement reste stable, la courbe représentant le jeu de test ne sera pas forcément décroissant mais plutôt instable et aléatoire.

## Evaluation d'un modèle

Afin d'évaluer un modèle nous avons utilisé une feature de scikit-learn<sup>4</sup> qui permet de faire tourner plusieurs fois notre modèle en faisant varier les hyperparamètres et de nous retourner la valeur moyenne des résultats sur le jeu de test.

## Critères de sélections

Comme observé précédemment, pour différencier et sélectionner le modèle qui serait le plus optimal nous avons utilisé deux critères de sélection qui sont la stabilité lors de l'entraînement et la vitesse à laquelle le modèle devient stable durant l'entraînement.



---

<sup>4</sup> Voir le [notebook](#) associé

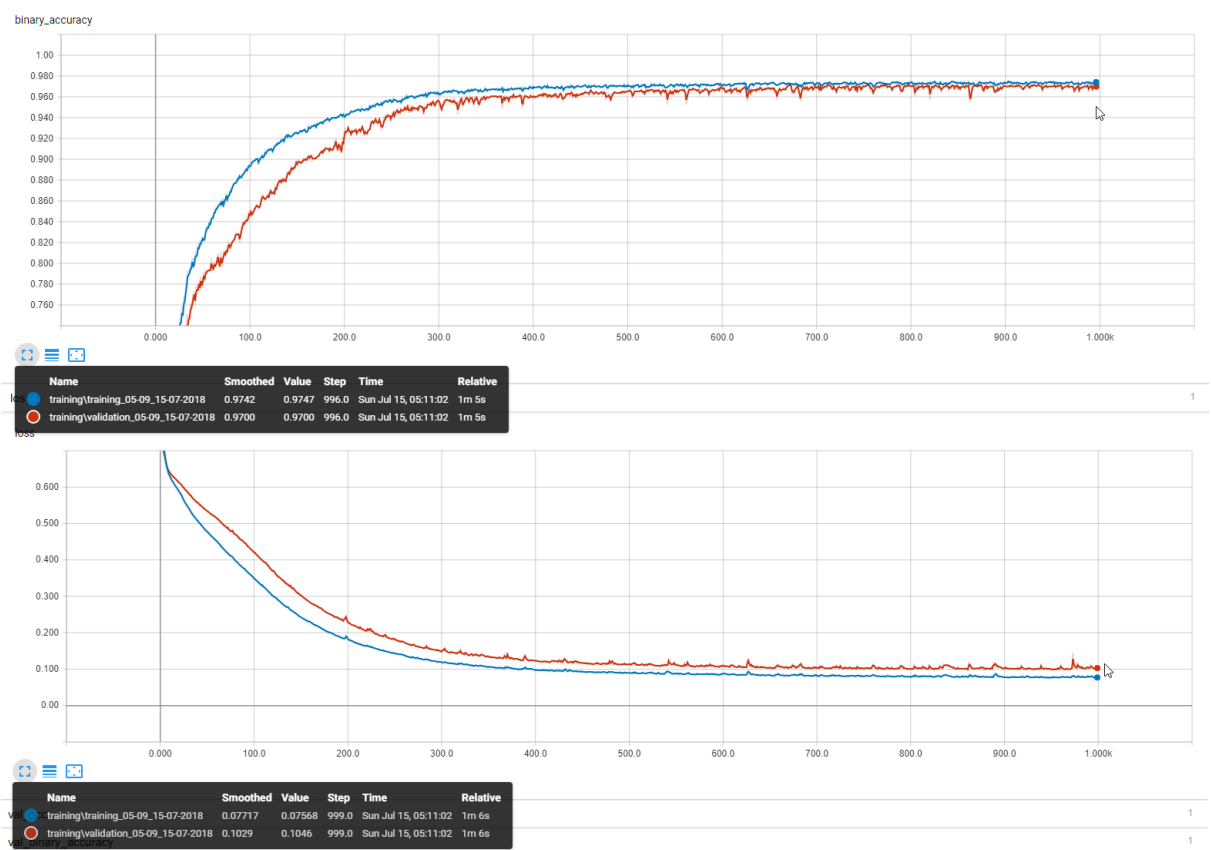
# Modèle choisi

Après qu'après quelques centaines d'essais à faire varier les fonctions d'activations, le nombre de couches et le nombre de neurones par couche et également les fonctions de loss et les optimizers. Nous avons réussi à obtenir un modèle cohérent et assez complexe pour correctement traiter les données d'apprentissage mais simple pour essayer de généraliser.

Vous pourrez le trouver plus en détail sur le [notebook](#).

Ce modèle est simplement constitué de 2 couches, la première avec autant de neurones que de dimension dans le jeu d'entraînement avec la fonction d'activation tanh, puis la couche de sortie binaire utilisant sigmoid.

Nous avons essayé d'utiliser les mêmes fonctionnalités que celle implémentée dans notre lib afin de pouvoir comparer les résultats.



## Comparaison avec notre lib

Comme dit plus haut nous avons essayé d'utiliser les mêmes fonctionnalités de Keras que celles présentent dans notre lib. Malheureusement pour des problèmes de temps nous n'avons pas pu implémenter le perceptron multicouche.

Nous ne nous sommes pas arrêté à cet échec et avons essayé plusieurs autres modèles présents dans notre lib :

### Classification Linéaire :

Avec notre lib nous obtenons un résultat d'environ 50% sur le jeu de test. Nous avons remarqué que le pipeline entre le python et le c++ est long et que le temps d'entraînement du modèle est lourd en ressource.

Nous avons comparé à la fonction `linear_model` de `sklearn`, et avec les hyperparamètres par défauts celle-ci nous donne 57% de résultats sur le jeu de test.

On peut alors en conclure que mise à part le temps d'exécution notre lib donne un résultat cohérent.

Vous pourrez trouver les notebooks associés ici :

- [Lib Notebook](#)
- [Sklearn Notebook](#)

### RBF Naïf pour une tâche de classification :

Vous pourrez trouver les notebooks associés ici :

- [Lib notebook](#)
- [Sklearn Notebook](#)

## Conclusion

Nous sommes assez satisfaits des résultats des différents algorithmes, que ce soit pour les algorithmes de recherche du plus court chemin ou ceux de Machine Learning. Nous n'avons pas eu le temps de réaliser l'intégralité des objectifs demandé. Notamment sur le perceptron multicouche ou l'implémentation d'un modèle de notre choix. Comme action à prendre pour les années à venir nous serons plus pragmatique et essaierons de ne pas reproduire ces erreurs. Malgré cela nous avons réussi à l'aide d'outils comme Keras et Tensorflow de nous essayer au Machine Learning. Nous avons pu avec une méthode incrémental définir quelle modèle permettait au mieux de généraliser notre jeu de donnée. Avec des résultats approximativement les 99% sur le jeu de test, nous ne pouvons qu'être positifs sur ce point.

Nous avons surtout joué avec les hyper paramètres des fonctions comme les fonctions d'activations. Mais nous n'avons pas eu le temps de faire des analyses plus poussé en amont sur notre jeu de donnée, comme des matrices de relation entre chaque feature. Ou encore essayer d'autre approche de Machine Learning comme le [TSNE](#).